TCL/TK ELECTRONIC REFERENCE

for Tcl /Tk version 8.0.x and [incr Tcl] version 3.0

Coverted to Adobe Acrobat Format (.pdf) by Charles Todd, Oct 1998.

ctodd@ball.com

Tcl	*Applications*	*Built-In C Library*
Tk	*Applications*	*Built-In C Library*
[incr Tcl]	*[incr Widgets]*	*[Incr Tk]*

NAME

tclsh − Simple shell containing Tcl interpreter

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 you create a Tcl script in a file whose first line is

#!/usr/local/bin/tclsh

then you can invoke the script file directly from your shell if you mark the file as executable. This assumes that **tclsh** has been installed in the default location in /usr/local/bin; if it's installed somewhere else then you'll have to modify the above line to match. Many UNIX systems do not allow the **#!** line to exceed about 30 characters in length, so be sure that the **tclsh** executable can be accessed with a short file name.

An even better approach is to start your script files with the following three lines:

```
#!/bin/sh
# the next line restarts using tclsh \
exec tclsh "$0" "$@"
```

This approach has three advantages over the approach in the previous paragraph. First, the location of the **tclsh** binary doesn't have to be hard-wired into the script: it can be anywhere in your shell search path. Second, it gets around the 30-character file name limit in the previous approach. Third, this approach will work even if **tclsh** is itself a shell script (this is done on some systems in order to handle multiple architectures or operating systems: the **tclsh** script selects one of several binaries to run). The three lines cause both **sh** and **tclsh** to process the script, but the **exec** is only executed by **sh**. **sh** processes the script first; it treats the second line as a comment and executes the third line. The **exec** statement cause the shell to stop processing and instead to start up **tclsh** to reprocess the entire script. When **tclsh** starts up, it treats all three lines as comments, since the backslash at the end of the second line causes the third line to be treated as part of the comment on the second line.

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 "**%** ". You 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 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 isn't yet complete; if **tcl_prompt2** isn't set then no prompt is output for incomplete commands.

KEYWORDS

argument, interpreter, prompt, script file, shell

NAME

wish − Simple windowing shell

SYNOPSIS

wish ?*fileName arg arg ...*?

OPTIONS

−**colormap** *new*	Specifies that the window should have a new private colormap instead of using the default colormap for the screen.
−**display** *display*	Display (and screen) on which to display window.
−**geometry** *geometry*	Initial geometry to use for window. If this option is specified, its value is stored in the **geometry** global variable of the application's Tcl interpreter.
−**name** *name*	Use *name* as the title to be displayed in the window, and as the name of the interpreter for **send** commands.
−**sync**	Execute all X server commands synchronously, so that errors are reported immediately. This will result in much slower execution, but it is useful for debugging.
−**use** *id*	Specifies that the main window for the application is to be embedded in the window whose identifier is *id*, instead of being created as an independent toplevel window. *Id* must be specified in the same way as the value for the −**use** option for toplevel widgets (i.e. it has a form like that returned by the **winfo id** command).
−**visual** *visual*	Specifies the visual to use for the window. *Visual* may have any of the forms supported by the **Tk_GetVisual** procedure.
− −	Pass all remaining arguments through to the script's **argv** variable without interpreting them. This provides a mechanism for passing arguments such as −**name** to a script instead of having **wish** interpret them.

DESCRIPTION

Wish is a simple program consisting of the Tcl command language, the Tk toolkit, and a main program that reads commands from standard input or from a file. It creates a main window and then processes Tcl commands. If **wish** is invoked with no arguments, or with a first argument that starts with "−", then it reads Tcl commands interactively from standard input. It will continue processing commands until all windows have been deleted or until end-of-file is reached on standard input. If there exists a file **.wishrc** in the home directory of the user, **wish** evaluates the file as a Tcl script just before reading the first command from standard input.

If **wish** is invoked with an initial *fileName* argument, then *fileName* is treated as the name of a script file. **Wish** will evaluate the script in *fileName* (which presumably creates a user interface), then it will respond to events until all windows have been deleted. Commands will not be read from standard input. There is no automatic evaluation of **.wishrc** in this case, but the script file can always **source** it if desired.

OPTIONS

Wish automatically processes all of the command-line options described in the **OPTIONS** summary above. Any other command-line arguments besides these are passed through to the application using the **argc** and **argv** variables described later.

APPLICATION NAME AND CLASS

The name of the application, which is used for purposes such as **send** commands, is taken from the −**name** option, if it is specified; otherwise it is taken from *fileName*, if it is specified, or from the command name

by which **wish** was invoked. In the last two cases, if the name contains a ‘‘/’’ character, then only the characters after the last slash are used as the application name.

The class of the application, which is used for purposes such as specifying options with a **RESOURCE_MANAGER** property or .Xdefaults file, is the same as its name except that the first letter is capitalized.

VARIABLES

Wish sets the following Tcl variables:

argc Contains a count of the number of *arg* arguments (0 if none), not including the options described above.

argv Contains a Tcl list whose elements are the *arg* arguments that follow a −− option or don’t match any of the options described in OPTIONS above, in order, or an empty string if there are no such arguments.

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

geometry If the −**geometry** option is specified, **wish** copies its value into this variable. If the variable still exists after *fileName* has been evaluated, **wish** uses the value of the variable in a **wm geometry** command to set the main window’s geometry.

tcl_interactive Contains 1 if **wish** is reading commands interactively (*fileName* was not specified and standard input is a terminal-like device), 0 otherwise.

SCRIPT FILES

If you create a Tcl script in a file whose first line is

#!/usr/local/bin/wish

then you can invoke the script file directly from your shell if you mark it as executable. This assumes that **wish** has been installed in the default location in /usr/local/bin; if it’s installed somewhere else then you’ll have to modify the above line to match. Many UNIX systems do not allow the **#!** line to exceed about 30 characters in length, so be sure that the **wish** executable can be accessed with a short file name.

An even better approach is to start your script files with the following three lines:

```
#!/bin/sh
# the next line restarts using wish \
exec wish "$0" "$@"
```

This approach has three advantages over the approach in the previous paragraph. First, the location of the **wish** binary doesn’t have to be hard-wired into the script: it can be anywhere in your shell search path. Second, it gets around the 30-character file name limit in the previous approach. Third, this approach will work even if **wish** is itself a shell script (this is done on some systems in order to handle multiple architectures or operating systems: the **wish** script selects one of several binaries to run). The three lines cause both **sh** and **wish** to process the script, but the **exec** is only executed by **sh**. **sh** processes the script first; it treats the second line as a comment and executes the third line. The **exec** statement cause the shell to stop processing and instead to start up **wish** to reprocess the entire script. When **wish** starts up, it treats all three lines as comments, since the backslash at the end of the second line causes the third line to be treated as part of the comment on the second line.

PROMPTS

When **wish** is invoked interactively it normally prompts for each command with ‘‘**%** ’’. You 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 to output a prompt; instead of outputting a prompt **wish** 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 isn’t yet complete; if **tcl_prompt2** isn’t set then no prompt is output for incomplete commands.

KEYWORDS

shell, toolkit

NAME

itclsh − Simple shell for [incr Tcl]

SYNOPSIS

itclsh ?*fileName arg arg ...*?

DESCRIPTION

itclsh is a shell-like application that reads Tcl commands from its standard input, or from a file, and evaluates them. It is just like **tclsh**, but includes the **[incr Tcl]** extensions for object-oriented programming.

See the **tclsh** man page for details concerning usage. See the **itcl** man page for an overview of **[incr Tcl]**.

KEYWORDS

Tcl, itcl, interpreter, script file, shell

NAME

itkwish − Simple windowing shell for [incr Tcl] / [incr Tk]

SYNOPSIS

itkwish ?*fileName arg arg ...*?

OPTIONS

−**display** *display*	Display (and screen) on which to display window.
−**geometry** *geometry*	Initial geometry to use for window. If this option is specified, its value is stored in the **geometry** global variable of the application's Tcl interpreter.
−**name** *name*	Use *name* as the title to be displayed in the window, and as the name of the interpreter for **send** commands.
−**sync**	Execute all X server commands synchronously, so that errors are reported immediately. This will result in much slower execution, but it is useful for debugging.
− −	Pass all remaining arguments through to the script's **argv** variable without interpreting them. This provides a mechanism for passing arguments such as −**name** to a script instead of having **itkwish** interpret them.

DESCRIPTION

itkwish is a simple program consisting of the Tcl command language, the Tk toolkit, the **[incr Tcl]** extension for object-oriented programming, and the **[incr Tk]** extension for building mega-widgets. The main program creates an interpreter, creates a main window, and then processes Tcl commands from standard input or from a file.

itkwish is just like **wish**, but includes the **[incr Tcl]** / **[incr Tk]** extensions.

See the **wish** man page for details concerning usage. See the **itcl** and **itk** man pages for an overview of **[incr Tcl]** / **[incr Tk]**.

KEYWORDS

Tcl, Tk, itcl, itk, interpreter, shell, toolkit

NAME

Tcl − Summary of Tcl language syntax.

DESCRIPTION

The following rules define the syntax and semantics of the 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 newlines, which are command separators).

[4] If the first character of a word is 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 into 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 *variable substitution*: the dollar-sign and the following characters are replaced in the word by the value of a variable. Variable substitution 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. The following table lists the backslash sequences that are handled specially, along with the value that replaces each sequence.

a Audible alert (bell) (0x7).

b Backspace (0x8).

f Form feed (0xc).

n Newline (0xa).

r Carriage-return (0xd).

t Tab (0x9).

v Vertical tab (0xb).

<newline>*whiteSpace*
A single space character replaces the backslash, newline, and all spaces and tabs 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.

x*hh* 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.

NAME

after − Execute a command after a time delay

SYNOPSIS

after *ms*

after *ms* ?*script script script ...*?

after cancel *id*

after cancel *script script script ...*

after idle ?*script script script ...*?

after info ?*id*?

DESCRIPTION

This command is used to delay execution of the program or to execute a command in background sometime in the future. It has several forms, depending on the first argument to the command:

after *ms*

Ms must be an integer giving a time in milliseconds. The command sleeps for *ms* milliseconds and then returns. While the command is sleeping the application does not respond to events.

after *ms* ?*script script script ...*?

In this form the command returns immediately, but it arranges for a Tcl command to be executed *ms* milliseconds later as an event handler. The command will be executed exactly once, at the given time. The delayed command is formed by concatenating all the *script* arguments in the same fashion as the **concat** command. The command will be executed at global level (outside the context of any Tcl procedure). If an error occurs while executing the delayed command then the **bgerror** mechanism is used to report the error. The **after** command returns an identifier that can be used to cancel the delayed command using **after cancel**.

after cancel *id*

Cancels the execution of a delayed command that was previously scheduled. *Id* indicates which command should be canceled; it must have been the return value from a previous **after** command. If the command given by *id* has already been executed then the **after cancel** command has no effect.

after cancel *script script ...*

This command also cancels the execution of a delayed command. The *script* arguments are concatenated together with space separators (just as in the **concat** command). If there is a pending command that matches the string, it is cancelled and will never be executed; if no such command is currently pending then the **after cancel** command has no effect.

after idle *script* ?*script script ...*?

Concatenates the *script* arguments together with space separators (just as in the **concat** command), and arranges for the resulting script to be evaluated later as an idle callback. The script will be run exactly once, the next time the event loop is entered and there are no events to process. The command returns an identifier that can be used to cancel the delayed command using **after cancel**. If an error occurs while executing the script then the **bgerror** mechanism is used to report the error.

after info ?*id*?

This command returns information about existing event handlers. If no *id* argument is supplied,

the command returns a list of the identifiers for all existing event handlers created by the **after** command for this interpreter. If *id* is supplied, it specifies an existing handler; *id* must have been the return value from some previous call to **after** and it must not have triggered yet or been cancelled. In this case the command returns a list with two elements. The first element of the list is the script associated with *id*, and the second element is either **idle** or **timer** to indicate what kind of event handler it is.

The **after** *ms* and **after idle** forms of the command assume that the application is event driven: the delayed commands will not be executed unless the application enters the event loop. In applications that are not normally event-driven, such as **tclsh**, the event loop can be entered with the **vwait** and **update** commands.

SEE ALSO

bgerror

KEYWORDS

cancel, delay, idle callback, sleep, time

NAME

append − Append to variable

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.

KEYWORDS

append, variable

NAME

array − Manipulate array variables

SYNOPSIS

array *option arrayName* ?*arg arg ...*?

DESCRIPTION

This command performs one of several operations on the variable given by *arrayName*. Unless otherwise specified for individual commands below, *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 have been the return value from a previous invocation of **array startsearch**. 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 state associated with that search. *SearchId* indicates which search on *arrayName* to destroy, and must have been the return value from a previous invocation of **array startsearch**. Returns an empty string.

array exists *arrayName*

Returns 1 if *arrayName* is an array variable, 0 if there is no variable by that name or if it is a scalar variable.

array get *arrayName* ?*pattern*?

Returns a list containing pairs of elements. The first element in each pair is the name of an element in *arrayName* and the second element of each pair is the value of the array element. The order of the pairs is undefined. If *pattern* is not specified, then all of the elements of the array are included in the result. If *pattern* is specified, then only those elements whose names match *pattern* (using the glob-style matching rules of **string match**) are included. If *arrayName* isn't the name of an array variable, or if the array contains no elements, then an empty list is returned.

array names *arrayName* ?*pattern*?

Returns a list containing the names of all of the elements in the array that match *pattern* (using the glob-style matching rules of **string match**). If *pattern* is omitted then the command returns all of the element names in the array. If there are no (matching) elements in the array, or if *arrayName* isn't the name of an array variable, 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 have been 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 set *arrayName list*

Sets the values of one or more elements in *arrayName*. *list* must have a form like that returned by **array get**, consisting of an even number of elements. Each odd-numbered element in *list* is treated as an element name within *arrayName*, and the following element in *list* is used as a new value for that array element. If the variable *arrayName* does not already exist and *list* is empty,

arrayName is created with an empty array value.

array size *arrayName*
Returns a decimal string giving the number of elements in the array. If *arrayName* isn't the name of an array then 0 is returned.

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 to be underway simultaneously for the same array.

KEYWORDS
array, element names, search

NAME

bgerror − Command invoked to process background errors

SYNOPSIS

bgerror *message*

DESCRIPTION

The **bgerror** command doesn't exist as built-in part of Tcl. Instead, individual applications or users can define a **bgerror** command (e.g. as a Tcl procedure) if they wish to handle background errors.

A background error is one that occurs in an event handler or some other command that didn't originate with the application. For example, if an error occurs while executing a command specified with the **after** command, then it is a background error. For a non-background error, the error can simply be returned up through nested Tcl command evaluations until it reaches the top-level code in the application; then the application can report the error in whatever way it wishes. When a background error occurs, the unwinding ends in the Tcl library and there is no obvious way for Tcl to report the error.

When Tcl detects a background error, it saves information about the error and invokes the **bgerror** command later as an idle event handler. Before invoking **bgerror**, Tcl restores the **errorInfo** and **errorCode** variables to their values at the time the error occurred, then it invokes **bgerror** with the error message as its only argument. Tcl assumes that the application has implemented the **bgerror** command, and that the command will report the error in a way that makes sense for the application. Tcl will ignore any result returned by the **bgerror** command as long as no error is generated.

If another Tcl error occurs within the **bgerror** command (for example, because no **bgerror** command has been defined) then Tcl reports the error itself by writing a message to stderr.

If several background errors accumulate before **bgerror** is invoked to process them, **bgerror** will be invoked once for each error, in the order they occurred. However, if **bgerror** returns with a break exception, then any remaining errors are skipped without calling **bgerror**.

Tcl has no default implementation for **bgerror**. However, in applications using Tk there is a default **bgerror** procedure which posts a dialog box containing the error message and offers the user a chance to see a stack trace showing where the error occurred.

KEYWORDS

background error, reporting

NAME

binary − Insert and extract fields from binary strings

SYNOPSIS

binary format *formatString* ?*arg arg* ...?
binary scan *string formatString* ?*varName varName* ...?

DESCRIPTION

This command provides facilities for manipulating binary data. The first form, **binary format**, creates a binary string from normal Tcl values. For example, given the values 16 and 22, it might produce an 8-byte binary string consisting of two 4-byte integers, one for each of the numbers. The second form of the command, **binary scan**, does the opposite: it extracts data from a binary string and returns it as ordinary Tcl string values.

BINARY FORMAT

The **binary format** command generates a binary string whose layout is specified by the *formatString* and whose contents come from the additional arguments. The resulting binary value is returned.

The *formatString* consists of a sequence of zero or more field specifiers separated by zero or more spaces. Each field specifier is a single type character followed by an optional numeric *count*. Most field specifiers consume one argument to obtain the value to be formatted. The type character specifies how the value is to be formatted. The *count* typically indicates how many items of the specified type are taken from the value. If present, the *count* is a non-negative decimal integer or ∗, which normally indicates that all of the items in the value are to be used. If the number of arguments does not match the number of fields in the format string that consume arguments, then an error is generated.

Each type-count pair moves an imaginary cursor through the binary data, storing bytes at the current position and advancing the cursor to just after the last byte stored. The cursor is initially at position 0 at the beginning of the data. The type may be any one of the following characters:

a Stores a character string of length *count* in the output string. If *arg* has fewer than *count* bytes, then additional zero bytes are used to pad out the field. If *arg* is longer than the specified length, the extra characters will be ignored. If *count* is ∗, then all of the bytes in *arg* will be formatted. If *count* is omitted, then one character will be formatted. For example,

binary format a7a∗a alpha bravo charlie

will return a string equivalent to **alpha\000\000bravoc**.

A This form is the same as **a** except that spaces are used for padding instead of nulls. For example,

binary format A6A∗A alpha bravo charlie

will return **alpha bravoc**.

b Stores a string of *count* binary digits in low-to-high order within each byte in the output string. *Arg* must contain a sequence of **1** and **0** characters. The resulting bytes are emitted in first to last order with the bits being formatted in low-to-high order within each byte. If *arg* has fewer than *count* digits, then zeros will be used for the remaining bits. If *arg* has more than the specified number of digits, the extra digits will be ignored. If *count* is ∗, then all of the digits in *arg* will be formatted. If *count* is omitted, then one digit will be formatted. If the number of bits formatted does not end at a byte boundary, the remaining bits of the last byte will be zeros. For example,

binary format b5b∗ 11100 111000011010

will return a string equivalent to **\x07\x87\x05**.

B This form is the same as **b** except that the bits are stored in high-to-low order within each byte. For example,

binary format B5B∗ 11100 111000011010

will return a string equivalent to **\xe0\xe1\xa0**.

h Stores a string of *count* hexadecimal digits in low-to-high within each byte in the output string. *Arg* must contain a sequence of characters in the set "0123456789abcdefABCDEF". The resulting bytes are emitted in first to last order with the hex digits being formatted in low-to-high order within each byte. If *arg* has fewer than *count* digits, then zeros will be used for the remaining digits. If *arg* has more than the specified number of digits, the extra digits will be ignored. If *count* is *, then all of the digits in *arg* will be formatted. If *count* is omitted, then one digit will be formatted. If the number of digits formatted does not end at a byte boundary, the remaining bits of the last byte will be zeros. For example,

binary format h3h* AB def

will return a string equivalent to **\xba\xed\x0f**.

H This form is the same as **h** except that the digits are stored in high-to-low order within each byte. For example,

binary format H3H* ab DEF

will return a string equivalent to **\xab\xde\xf0**.

c Stores one or more 8-bit integer values in the output string. If no *count* is specified, then *arg* must consist of an integer value; otherwise *arg* must consist of a list containing at least *count* integer elements. The low-order 8 bits of each integer are stored as a one-byte value at the cursor position. If *count* is *, then all of the integers in the list are formatted. If the number of elements in the list is fewer than *count*, then an error is generated. If the number of elements in the list is greater than *count*, then the extra elements are ignored. For example,

binary format c3cc* {3 -3 128 1} 257 {2 5}

will return a string equivalent to **\x03\xfd\x80\x01\x02\x05**, whereas

binary format c {2 5}

will generate an error.

s This form is the same as **c** except that it stores one or more 16-bit integers in little-endian byte order in the output string. The low-order 16-bits of each integer are stored as a two-byte value at the cursor position with the least significant byte stored first. For example,

binary format s3 {3 -3 258 1}

will return a string equivalent to **\x03\x00\xfd\xff\x02\x01**.

S This form is the same as **s** except that it stores one or more 16-bit integers in big-endian byte order in the output string. For example,

binary format S3 {3 -3 258 1}

will return a string equivalent to **\x00\x03\xff\xfd\x01\x02**.

i This form is the same as **c** except that it stores one or more 32-bit integers in little-endian byte order in the output string. The low-order 32-bits of each integer are stored as a four-byte value at the cursor position with the least significant byte stored first. For example,

binary format i3 {3 -3 65536 1}

will return a string equivalent to **\x03\x00\x00\x00\xfd\xff\xff\xff\x00\x00\x10\x00**.

I This form is the same as **i** except that it stores one or more one or more 32-bit integers in big-endian byte order in the output string. For example,

binary format I3 {3 -3 65536 1}

will return a string equivalent to **\x00\x00\x00\x03\xff\xff\xff\xfd\x00\x10\x00\x00**.

f This form is the same as **c** except that it stores one or more one or more single-precision floating in the machine's native representation in the output string. This representation is not portable across architectures, so it should not be used to communicate floating point numbers across the network. The size of a floating point number may vary across architectures, so the number of bytes that are generated may vary. If the value overflows the machine's native representation, then the value of

FLT_MAX as defined by the system will be used instead. Because Tcl uses double-precision floating-point numbers internally, there may be some loss of precision in the conversion to single-precision. For example, on a Windows system running on an Intel Pentium processor,

binary format f2 {1.6 3.4}

will return a string equivalent to **\xcd\xcc\xcc\x3f\x9a\x99\x59\x40**.

d This form is the same as **f** except that it stores one or more one or more double-precision floating in the machine's native representation in the output string. For example, on a Windows system running on an Intel Pentium processor,

binary format d1 {1.6}

will return a string equivalent to **\x9a\x99\x99\x99\x99\x99\xf9\x3f**.

x Stores *count* null bytes in the output string. If *count* is not specified, stores one null byte. If *count* is *, generates an error. This type does not consume an argument. For example,

binary format a3xa3x2a3 abc def ghi

will return a string equivalent to **abc\000def\000\000ghi**.

X Moves the cursor back *count* bytes in the output string. If *count* is * or is larger than the current cursor position, then the cursor is positioned at location 0 so that the next byte stored will be the first byte in the result string. If *count* is omitted then the cursor is moved back one byte. This type does not consume an argument. For example,

binary format a3X*a3X2a3 abc def ghi

will return **dghi**.

@ Moves the cursor to the absolute location in the output string specified by *count*. Position 0 refers to the first byte in the output string. If *count* refers to a position beyond the last byte stored so far, then null bytes will be placed in the unitialized locations and the cursor will be placed at the specified location. If *count* is *, then the cursor is moved to the current end of the output string. If *count* is omitted, then an error will be generated. This type does not consume an argument. For example,

binary format a5@2a1@*a3@10a1 abcde f ghi j

will return **abfdeghi\000\000j**.

BINARY SCAN

The **binary scan** command parses fields from a binary string, returning the number of conversions performed. *String* gives the input to be parsed and *formatString* indicates how to parse it. Each *varName* gives the name of a variable; when a field is scanned from *string* the result is assigned to the corresponding variable.

As with **binary format**, the *formatString* consists of a sequence of zero or more field specifiers separated by zero or more spaces. Each field specifier is a single type character followed by an optional numeric *count*. Most field specifiers consume one argument to obtain the variable into which the scanned values should be placed. The type character specifies how the binary data is to be interpreted. The *count* typically indicates how many items of the specified type are taken from the data. If present, the *count* is a non-negative decimal integer or *, which normally indicates that all of the remaining items in the data are to be used. If there are not enough bytes left after the current cursor position to satisfy the current field specifier, then the corresponding variable is left untouched and **binary scan** returns immediately with the number of variables that were set. If there are not enough arguments for all of the fields in the format string that consume arguments, then an error is generated.

Each type-count pair moves an imaginary cursor through the binary data, reading bytes from the current position. The cursor is initially at position 0 at the beginning of the data. The type may be any one of the following characters:

a The data is a character string of length *count*. If *count* is *, then all of the remaining bytes in *string* will be scanned into the variable. If *count* is omitted, then one character will be scanned. For

example,

binary scan abcde\000fghi a6a10 var1 var2

will return **1** with the string equivalent to **abcde\000** stored in **var1** and **var2** left unmodified.

A This form is the same as **a**, except trailing blanks and nulls are stripped from the scanned value before it is stored in the variable. For example,

binary scan "abc efghi \000" a* var1

will return **1** with **abc efghi** stored in **var1**.

b The data is turned into a string of *count* binary digits in low-to-high order represented as a sequence of ''1'' and ''0'' characters. The data bytes are scanned in first to last order with the bits being taken in low-to-high order within each byte. Any extra bits in the last byte are ignored. If *count* is *, then all of the remaining bits in **string** will be scanned. If *count* is omitted, then one bit will be scanned. For example,

binary scan \x07\x87\x05 b5b* var1 var2

will return **2** with **11100** stored in **var1** and **1110000110100000** stored in **var2**.

B This form is the same as **B**, except the bits are taken in high-to-low order within each byte. For example,

binary scan \x70\x87\x05 b5b* var1 var2

will return **2** with **01110** stored in **var1** and **1000011100000101** stored in **var2**.

h The data is turned into a string of *count* hexadecimal digits in low-to-high order represented as a sequence of characters in the set ''0123456789abcdef''. The data bytes are scanned in first to last order with the hex digits being taken in low-to-high order within each byte. Any extra bits in the last byte are ignored. If *count* is *, then all of the remaining hex digits in **string** will be scanned. If *count* is omitted, then one hex digit will be scanned. For example,

binary scan \x07\x86\x05 h3h* var1 var2

will return **2** with **706** stored in **var1** and **50** stored in **var2**.

H This form is the same as **h**, except the digits are taken in low-to-high order within each byte. For example,

binary scan \x07\x86\x05 H3H* var1 var2

will return **2** with **078** stored in **var1** and **05** stored in **var2**.

c The data is turned into *count* 8-bit signed integers and stored in the corresponding variable as a list. If *count* is *, then all of the remaining bytes in **string** will be scanned. If *count* is omitted, then one 8-bit integer will be scanned. For example,

binary scan \x07\x86\x05 c2c* var1 var2

will return **2** with **7 -122** stored in **var1** and **5** stored in **var2**. Note that the integers returned are signed, but they can be converted to unsigned 8-bit quantities using an expression like:

expr ($num + 0x100) % 0x100

s The data is interpreted as *count* 16-bit signed integers represented in little-endian byte order. The integers are stored in the corresponding variable as a list. If *count* is *, then all of the remaining bytes in **string** will be scanned. If *count* is omitted, then one 16-bit integer will be scanned. For example,

binary scan \x05\x00\x07\x00\xf0\xff s2s* var1 var2

will return **2** with **5 7** stored in **var1** and **-16** stored in **var2**. Note that the integers returned are signed, but they can be converted to unsigned 16-bit quantities using an expression like:

expr ($num + 0x10000) % 0x10000

S This form is the same as **s** except that the data is interpreted as *count* 16-bit signed integers represented in big-endian byte order. For example,

binary scan \x00\x05\x00\x07\xff\xf0 S2S* var1 var2

will return **2** with **5 7** stored in **var1** and **-16** stored in **var2**.

i The data is interpreted as *count* 32-bit signed integers represented in little-endian byte order. The integers are stored in the corresponding variable as a list. If *count* is ∗, then all of the remaining bytes in **string** will be scanned. If *count* is omitted, then one 32-bit integer will be scanned. For example,

binary scan \x05\x00\x00\x00\x07\x00\x00\x00\xf0\xff\xff\xff i2i∗ var1 var2

will return **2** with **5 7** stored in **var1** and **-16** stored in **var2**. Note that the integers returned are signed and cannot be represented by Tcl as unsigned values.

I This form is the same as **I** except that the data is interpreted as *count* 32-bit signed integers represented in big-endian byte order. For example,

binary \x00\x00\x00\x05\x00\x00\x00\x07\xff\xff\xff\xf0 I2I∗ var1 var2

will return **2** with **5 7** stored in **var1** and **-16** stored in **var2**.

f The data is interpreted as *count* single-precision floating point numbers in the machine's native representation. The floating point numbers are stored in the corresponding variable as a list. If *count* is ∗, then all of the remaining bytes in **string** will be scanned. If *count* is omitted, then one single-precision floating point number will be scanned. The size of a floating point number may vary across architectures, so the number of bytes that are scanned may vary. If the data does not represent a valid floating point number, the resulting value is undefined and compiler dependent. For example, on a Windows system running on an Intel Pentium processor,

binary scan \x3f\xcc\xcc\xcd f var1

will return **1** with **1.6000000238418579** stored in **var1**.

d This form is the same as **f** except that the data is interpreted as *count* double-precision floating point numbers in the machine's native representation. For example, on a Windows system running on an Intel Pentium processor,

binary scan \x9a\x99\x99\x99\x99\x99\xf9\x3f d var1

will return **1** with **1.6000000000000001** stored in **var1**.

x Moves the cursor forward *count* bytes in *string*. If *count* is ∗ or is larger than the number of bytes after the current cursor cursor position, then the cursor is positioned after the last byte in *string*. If *count* is omitted, then the cursor is moved forward one byte. Note that this type does not consume an argument. For example,

binary scan \x01\x02\x03\x04 x2H∗ var1

will return **1** with **0304** stored in **var1**.

X Moves the cursor back *count* bytes in *string*. If *count* is ∗ or is larger than the current cursor position, then the cursor is positioned at location 0 so that the next byte scanned will be the first byte in *string*. If *count* is omitted then the cursor is moved back one byte. Note that this type does not consume an argument. For example,

binary scan \x01\x02\x03\x04 c2XH∗ var1 var2

will return **2** with **1 2** stored in **var1** and **020304** stored in **var2**.

@ Moves the cursor to the absolute location in the data string specified by *count*. Note that position 0 refers to the first byte in *string*. If *count* refers to a position beyond the end of *string*, then the cursor is positioned after the last byte. If *count* is omitted, then an error will be generated. For example,

binary scan \x01\x02\x03\x04 c2@1H∗ var1 var2

will return **2** with **1 2** stored in **var1** and **020304** stored in **var2**.

PLATFORM ISSUES

Sometimes it is desirable to format or scan integer values in the native byte order for the machine. Refer to the **byteOrder** element of the **tcl_platform** array to decide which type character to use when formatting or scanning integers.

SEE ALSO
format, scan, tclvars

KEYWORDS
binary, format, scan

NAME

break − Abort looping command

SYNOPSIS

break

DESCRIPTION

This command is typically invoked inside the body of a looping command such as **for** or **foreach** or **while**. It returns a TCL_BREAK code, which causes a break exception to occur. The exception causes the current script to be aborted out to the innermost containing loop command, which then aborts its execution and returns normally. Break exceptions are also handled in a few other situations, such as the **catch** command, Tk event bindings, and the outermost scripts of procedure bodies.

KEYWORDS

abort, break, loop

NAME

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

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. You should use the* **switch** *command instead.*

The **case** command matches *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 *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 wildcards described under **string match**. If a *patList* argument is **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 than the first form in some cases.

KEYWORDS

case, match, regular expression

NAME

catch − Evaluate script and trap exceptional returns

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).

Note that **catch** catches all exceptions, including those generated by **break** and **continue** as well as errors.

KEYWORDS

catch, error

NAME

cd − Change working directory

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. Returns an empty string.

KEYWORDS

working directory

NAME

clock − Obtain and manipulate time

SYNOPSIS

clock *option* ?*arg arg ...*?

DESCRIPTION

This command performs one of several operations that may obtain or manipulate strings or values that represent some notion of time. The *option* argument determines what action is carried out by the command. The legal *options* (which may be abbreviated) are:

clock clicks

Return a high-resolution time value as a system-dependent integer value. The unit of the value is system-dependent but should be the highest resolution clock available on the system such as a CPU cycle counter. This value should only be used for the relative measurement of elapsed time.

clock format *clockValue* ?**−format** *string*? ?**−gmt** *boolean*?

Converts an integer time value, typically returned by **clock seconds**, **clock scan**, or the **atime**, **mtime**, or **ctime** options of the **file** command, to human-readable form. If the **−format** argument is present the next argument is a string that describes how the date and time are to be formatted. Field descriptors consist of a **%** followed by a field descriptor character. All other characters are copied into the result. Valid field descriptors are:

%% Insert a %.

%a Abbreviated weekday name (Mon, Tue, etc.).

%A Full weekday name (Monday, Tuesday, etc.).

%b Abbreviated month name (Jan, Feb, etc.).

%B Full month name.

%c Locale specific date and time.

%d Day of month (01 - 31).

%H Hour in 24-hour format (00 - 23).

%I Hour in 12-hour format (00 - 12).

%j Day of year (001 - 366).

%m Month number (01 - 12).

%M Minute (00 - 59).

%p AM/PM indicator.

%S Seconds (00 - 59).

%U Week of year (01 - 52), Sunday is the first day of the week.

%w Weekday number (Sunday = 0).

%W Week of year (01 - 52), Monday is the first day of the week.

%x Locale specific date format.

%X Locale specific time format.

%y Year without century (00 - 99).

%Y Year with century (e.g. 1990)

%Z Time zone name.

In addition, the following field descriptors may be supported on some systems (e.g. Unix but not Windows):

%D Date as %m/%d/%y.

%e Day of month (1 - 31), no leading zeros.

%h Abbreviated month name.

%n Insert a newline.

%r Time as %I:%M:%S %p.

%R Time as %H:%M.

%t Insert a tab.

%T Time as %H:%M:%S.

If the **−format** argument is not specified, the format string "**%a %b %d %H:%M:%S %Z %Y**" is used. If the **−gmt** argument is present the next argument must be a boolean which if true specifies that the time will be formatted as Greenwich Mean Time. If false then the local timezone will be used as defined by the operating environment.

clock scan *dateString* ?**−base** *clockVal*? ?**−gmt** *boolean*?

Convert *dateString* to an integer clock value (see **clock seconds**). This command can parse and convert virtually any standard date and/or time string, which can include standard time zone mnemonics. If only a time is specified, the current date is assumed. If the string does not contain a time zone mnemonic, the local time zone is assumed, unless the **−gmt** argument is true, in which case the clock value is calculated assuming that the specified time is relative to Greenwich Mean Time.

If the **−base** flag is specified, the next argument should contain an integer clock value. Only the date in this value is used, not the time. This is useful for determining the time on a specific day or doing other date-relative conversions.

The *dateString* consists of zero or more specifications of the following form:

time A time of day, which is of the form: *hh*?*:mm*?*:ss*?? ?*meridian*? ?*zone*? or *hhmm* ?*meridian*? ?*zone*?. If no meridian is specified, *hh* is interpreted on a 24-hour clock.

date A specific month and day with optional year. The acceptable formats are *mm/dd*?*/yy*?, *monthname dd* ?, *yy*?, *dd monthname* ?*yy*? and *day, dd monthname yy*. The default year is the current year. If the year is less than 100, we treat the years 00-68 as 2000-2068 and the years 69-99 as 1969-1999. Not all platforms can represent the years 38-70, so an error may result if these years are used.

relative time

A specification relative to the current time. The format is *number unit* acceptable units are **year**, **fortnight**, **month**, **week**, **day**, **hour**, **minute** (or **min**), and **second** (or **sec**). The unit can be specified as a singular or plural, as in **3 weeks**. These modifiers may also be specified: **tomorrow**, **yesterday**, **today**, **now**, **last**, **this**, **next**, **ago**.

The actual date is calculated according to the following steps. First, any absolute date and/or time is processed and converted. Using that time as the base, day-of-week specifications are added. Next, relative specifications are used. If a date or day is specified, and no absolute or relative time is given, midnight is used. Finally, a correction is applied so that the correct hour of the day is

produced after allowing for daylight savings time differences and the correct date is given when going from the end of a long month to a short month.

clock seconds

Return the current date and time as a system-dependent integer value. The unit of the value is seconds, allowing it to be used for relative time calculations. The value is usually defined as total elapsed time from an ‘‘epoch’’. You shouldn’t assume the value of the epoch.

KEYWORDS

clock, date, time

NAME

close − Close an open channel.

SYNOPSIS

close *channelId*

DESCRIPTION

Closes the channel given by *channelId*. *ChannelId* must be a channel identifier such as the return value from a previous **open** or **socket** command. All buffered output is flushed to the channel's output device, any buffered input is discarded, the underlying file or device is closed, and *channelId* becomes unavailable for use.

If the channel is blocking, the command does not return until all output is flushed. If the channel is nonblocking and there is unflushed output, the channel remains open and the command returns immediately; output will be flushed in the background and the channel will be closed when all the flushing is complete.

If *channelId* is a blocking channel for a command pipeline then **close** waits for the child processes to complete.

If the channel is shared between interpreters, then **close** makes *channelId* unavailable in the invoking interpreter but has no other effect until all of the sharing interpreters have closed the channel. When the last interpreter in which the channel is registered invokes **close**, the cleanup actions described above occur. See the **interp** command for a description of channel sharing.

Channels are automatically closed when an interpreter is destroyed and when the process exits. Channels are switched to blocking mode, to ensure that all output is correctly flushed before the process exits.

The command returns an empty string, and may generate an error if an error occurs while flushing output.

KEYWORDS

blocking, channel, close, nonblocking

NAME

concat − Join lists together

SYNOPSIS

concat ?*arg arg ...*?

DESCRIPTION

This command treats each argument as a list and concatenates them into a single list. It also eliminates leading and trailing spaces in the *arg*'s and adds a single separator space between *arg*'s. It permits any number of arguments. For example, the command

concat a b {c d e} {f {g h}}

will return

a b c d e f {g h}

as its result.

If no *arg*s are supplied, the result is an empty string.

KEYWORDS

concatenate, join, lists

NAME

continue − Skip to the next iteration of a loop

SYNOPSIS

continue

DESCRIPTION

This command is typically invoked inside the body of a looping command such as **for** or **foreach** or **while**. It returns a TCL_CONTINUE code, which causes a continue exception to occur. The exception causes the current script to be aborted out to the innermost containing loop command, which then continues with the next iteration of the loop. Catch exceptions are also handled in a few other situations, such as the **catch** command and the outermost scripts of procedure bodies.

KEYWORDS

continue, iteration, loop

NAME

eof − Check for end of file condition on channel

SYNOPSIS

eof *channelId*

DESCRIPTION

Returns 1 if an end of file condition occurred during the most recent input operation on *channelId* (such as **gets**), 0 otherwise.

KEYWORDS

channel, end of file

NAME

error − Generate an error

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 reflecting 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 **tclvars** manual page 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.

KEYWORDS

error, errorCode, errorInfo

NAME

eval − Evaluate a Tcl script

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).

KEYWORDS

concatenate, evaluate, script

NAME

exec − Invoke subprocess(es)

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 *arg*'s) 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.

2> *fileName* Standard error from all commands in the pipeline is redirected to the file named *fileName*, overwriting its previous contents.

>& *fileName* Both standard output from the last command and standard error 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.

2>> *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 error 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.

2>@ *fileId* — *FileId* must be the identifier for an open file, such as the return value from a previous call to **open**. Standard error from all commands in the pipeline is 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 error 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 don't 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 input 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.

PORTABILITY ISSUES

Windows (all versions)

Reading from or writing to a socket, using the ''***@ fileId***'' notation, does not work. When reading from a socket, a 16-bit DOS application will hang and a 32-bit application will return immediately with end-of-file. When either type of application writes to a socket, the information is instead sent to the console, if one is present, or is discarded.

The Tk console text widget does not provide real standard IO capabilities. Under Tk, when redirecting from standard input, all applications will see an immediate end-of-file; information redirected to standard output or standard error will be discarded.

Either forward or backward slashes are accepted as path separators for arguments to Tcl commands. When executing an application, the path name specified for the application may also contain forward or backward slashes as path separators. Bear in mind, however, that most Windows applications accept arguments with forward slashes only as option delimiters and backslashes only in paths. Any arguments to an application that specify a path name with forward slashes will not

automatically be converted to use the backslash character. If an argument contains forward slashes as the path separator, it may or may not be recognized as a path name, depending on the program.

Additionally, when calling a 16-bit DOS or Windows 3.X application, all path names must use the short, cryptic, path format (e.g., using ‘‘applba~1.def’’ instead of ‘‘applbakery.default’’).

Two or more forward or backward slashes in a row in a path refer to a network path. For example, a simple concatenation of the root directory **c:/** with a subdirectory **/windows/system** will yield **c://windows/system** (two slashes together), which refers to the directory **/system** on the machine **windows** (and the **c:/** is ignored), and is not equivalent to **c:/windows/system**, which describes a directory on the current computer.

Windows NT

When attempting to execute an application, **exec** first searches for the name as it was specified. Then, in order, **.com**, **.exe**, and **.bat** are appended to the end of the specified name and it searches for the longer name. If a directory name was not specified as part of the application name, the following directories are automatically searched in order when attempting to locate the application:

The directory from which the Tcl executable was loaded.
The current directory.
The Windows NT 32-bit system directory.
The Windows NT 16-bit system directory.
The Windows NT home directory.
The directories listed in the path.

In order to execute the shell builtin commands like **dir** and **copy**, the caller must prepend ‘‘**cmd.exe /c** ’’ to the desired command.

Windows 95

When attempting to execute an application, **exec** first searches for the name as it was specified. Then, in order, **.com**, **.exe**, and **.bat** are appended to the end of the specified name and it searches for the longer name. If a directory name was not specified as part of the application name, the following directories are automatically searched in order when attempting to locate the application:

The directory from which the Tcl executable was loaded.
The current directory.
The Windows 95 system directory.
The Windows 95 home directory.
The directories listed in the path.

In order to execute the shell builtin commands like **dir** and **copy**, the caller must prepend ‘‘**command.com /c** ’’ to the desired command.

Once a 16-bit DOS application has read standard input from a console and then quit, all subsequently run 16-bit DOS applications will see the standard input as already closed. 32-bit applications do not have this problem and will run correctly even after a 16-bit DOS application thinks that standard input is closed. There is no known workaround for this bug at this time.

Redirection between the **NUL:** device and a 16-bit application does not always work. When redirecting from **NUL:**, some applications may hang, others will get an infinite stream of ‘‘0x01’’ bytes, and some will actually correctly get an immediate end-of-file; the behavior seems to depend upon something compiled into the application itself. When redirecting greater than 4K or so to

NUL:, some applications will hang. The above problems do not happen with 32-bit applications.

All DOS 16-bit applications are run synchronously. All standard input from a pipe to a 16-bit DOS application is collected into a temporary file; the other end of the pipe must be closed before the 16-bit DOS application begins executing. All standard output or error from a 16-bit DOS application to a pipe is collected into temporary files; the application must terminate before the temporary files are redirected to the next stage of the pipeline. This is due to a workaround for a Windows 95 bug in the implementation of pipes, and is how the Windows 95 command line interpreter handles pipes itself.

Certain applications, such as **command.com**, should not be executed interactively. Applications which directly access the console window, rather than reading from their standard input and writing to their standard output may fail, hang Tcl, or even hang the system if their own private console window is not available to them.

Windows 3.X

When attempting to execute an application, **exec** first searches for the name as it was specified. Then, in order, **.com**, **.exe**, and **.bat** are appended to the end of the specified name and it searches for the longer name. If a directory name was not specified as part of the application name, the following directories are automatically searched in order when attempting to locate the application:

The directory from which the Tcl executable was loaded.
The current directory.
The Windows 3.X system directory.
The Windows 3.X home directory.
The directories listed in the path.

In order to execute the shell builtin commands like **dir** and **copy**, the caller must prepend ''**command.com /c** '' to the desired command.

16-bit and 32-bit DOS and Windows applications may be executed. However, redirection and piping of standard IO only works with 16-bit DOS applications. 32-bit applications always see standard input as already closed, and any standard output or error is discarded, no matter where in the pipeline the application occurs or what redirection symbols are used by the caller. Additionally, for 16-bit applications, standard error is always sent to the same place as standard output; it cannot be redirected to a separate location. In order to achieve pseudo-redirection for 32-bit applications, the 32-bit application must instead be written to take command line arguments that specify the files that it should read from and write to and open those files itself.

All applications, both 16-bit and 32-bit, run synchronously; each application runs to completion before the next one in the pipeline starts. Temporary files are used to simulate piping between applications. The **exec** command cannot be used to start an application in the background.

When standard input is redirected from an open file using the ''**@** *fileId*'' notation, the open file is completely read up to its end. This is slightly different than under Windows 95 or NT, where the child application consumes from the open file only as much as it wants. Redirecting to an open file is supported as normal.

Macintosh

The **exec** command is not implemented and does not exist under Macintosh.

Unix

The **exec** command is fully functional and works as described.

SEE ALSO

open(n)

KEYWORDS

execute, pipeline, redirection, subprocess

NAME

exit − End the application

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.

KEYWORDS

exit, process

NAME

expr − Evaluate an expression

SYNOPSIS

expr *arg* ?*arg arg ...*?

DESCRIPTION

Concatenates *arg*'s (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's instructions. 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 an 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's instructions. 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, unary plus, bit-wise NOT, logical NOT. None of these operands may be applied to string operands, and bit-wise NOT may be applied only to integers.
∗ / %	Multiply, divide, remainder. None of these operands may be applied to string operands, 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. A right shift always propagates the sign bit.
< > <= >=	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 boolean and numeric (integers or floating-point) operands only.
||	Logical OR. Produces a 0 result if both operands are zero, 1 otherwise. Valid for boolean and numeric (integers or floating-point) operands only.
*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 conversion between integers and floating-point numbers and the generation of random 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, returns *arg*, otherwise converts *arg* to floating and returns the converted value.

int(*arg***)** If *arg* is an integer value, returns *arg*, otherwise converts *arg* to integer by truncation and returns the converted value.

rand() Returns a floating point number from zero to just less than one or, in mathematical terms, the range [0,1). The seed comes from the internal clock of the machine or may be set manual with the srand function.

round(*arg***)**
If *arg* is an integer value, returns *arg*, otherwise converts *arg* to integer by rounding and returns the converted value.

srand(*arg***)**
The *arg*, which must be an integer, is used to reset the seed for the random number generator. Returns the first random number from that seed. Each interpreter has it's own seed.

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 arithmetic computations, integers are used until some floating-point number is introduced, after which floating-point is used. For example,

expr 5 / 4

returns 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**.

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 specifier **%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 using string comparison after the second operand is converted to the string **18**. Because of Tcl's tendency to treat values as numbers whenever possible, it isn't generally a good idea to use operators like == when you really want string comparison and the values of the operands could be arbitrary; it's better in these cases to use the **string compare** command instead.

PERFORMANCE CONSIDERATIONS

Enclose expressions in braces for the best speed and the smallest storage requirements. This allows the Tcl bytecode compiler to generate the best code.

As mentioned above, expressions are substituted twice: once by the Tcl parser and once by the **expr** command. For example, the commands

set a 3
set b {$a + 2}
expr $b*4

return 11, not a multiple of 4. This is because the Tcl parser will first substitute **$a + 2** for the variable **b**, then the **expr** command will evaluate the expression **$a + 2*4**.

Most expressions do not require a second round of substitutions. Either they are enclosed in braces or, if not, their variable and command substitutions yield numbers or strings that don't themselves require substitutions. However, because a few unbraced expressions need two rounds of substitutions, the bytecode compiler must emit additional instructions to handle this situation. The most expensive code is required for unbraced expressions that contain command substitutions. These expressions must be implemented by generating new code each time the expression is executed.

KEYWORDS

arithmetic, boolean, compare, expression, fuzzy comparison

NAME

fblocked − Test whether the last input operation exhausted all available input

SYNOPSIS

fblocked *channelId*

DESCRIPTION

The **fblocked** command returns 1 if the most recent input operation on *channelId* returned less information than requested because all available input was exhausted. For example, if **gets** is invoked when there are only three characters available for input and no end-of-line sequence, **gets** returns an empty string and a subsequent call to **fblocked** will return 1.

SEE ALSO

gets(n), read(n)

KEYWORDS

blocking, nonblocking

NAME

fconfigure − Set and get options on a channel

SYNOPSIS

fconfigure *channelId*
fconfigure *channelId name*
fconfigure *channelId name value* ?*name value* ...?

DESCRIPTION

The **fconfigure** command sets and retrieves options for channels. *ChannelId* identifies the channel for which to set or query an option. If no *name* or *value* arguments are supplied, the command returns a list containing alternating option names and values for the channel. If *name* is supplied but no *value* then the command returns the current value of the given option. If one or more pairs of *name* and *value* are supplied, the command sets each of the named options to the corresponding *value*; in this case the return value is an empty string.

The options described below are supported for all channels. In addition, each channel type may add options that only it supports. See the manual entry for the command that creates each type of channels for the options that that specific type of channel supports. For example, see the manual entry for the **socket** command for its additional options.

−blocking *boolean*

The **−blocking** option determines whether I/O operations on the channel can cause the process to block indefinitely. The value of the option must be a proper boolean value. Channels are normally in blocking mode; if a channel is placed into nonblocking mode it will affect the operation of the **gets**, **read**, **puts**, **flush**, and **close** commands; see the documentation for those commands for details. For nonblocking mode to work correctly, the application must be using the Tcl event loop (e.g. by calling **Tcl_DoOneEvent** or invoking the **vwait** command).

−buffering *newValue*

If *newValue* is **full** then the I/O system will buffer output until its internal buffer is full or until the **flush** command is invoked. If *newValue* is **line**, then the I/O system will automatically flush output for the channel whenever a newline character is output. If *newValue* is **none**, the I/O system will flush automatically after every output operation. The default is for **−buffering** to be set to **full** except for channels that connect to terminal-like devices; for these channels the initial setting is **line**.

−buffersize *newSize*

Newvalue must be an integer; its value is used to set the size of buffers, in bytes, subsequently allocated for this channel to store input or output. *Newvalue* must be between ten and one million, allowing buffers of ten to one million bytes in size.

−eofchar *char*

−eofchar {*inChar outChar***}**

This option supports DOS file systems that use Control-z (\x1a) as an end of file marker. If *char* is not an empty string, then this character signals end of file when it is encountered during input. For output, the end of file character is output when the channel is closed. If *char* is the empty string, then there is no special end of file character marker. For read-write channels, a two-element list specifies the end of file marker for input and output, respectively. As a convenience, when setting the end-of-file character for a read-write channel you can specify a single value that will apply to both reading and writing. When querying the end-of-file character of a read-write channel, a two-element list will always be returned. The default value for **−eofchar** is the empty string in all cases except for files under Windows. In that case the **−eofchar** is Control-z (\x1a) for reading

and the empty string for writing.

−translation *mode*

−translation {*inMode outMode***}**

In Tcl scripts the end of a line is always represented using a single newline character (\n). However, in actual files and devices the end of a line may be represented differently on different platforms, or even for different devices on the same platform. For example, under UNIX newlines are used in files, whereas carriage-return-linefeed sequences are normally used in network connections. On input (i.e., with **gets** and **read**) the Tcl I/O system automatically translates the external end-of-line representation into newline characters. Upon output (i.e., with **puts**), the I/O system translates newlines to the external end-of-line representation. The default translation mode, **auto**, handles all the common cases automatically, but the **−translation** option provides explicit control over the end of line translations.

The value associated with **−translation** is a single item for read-only and write-only channels. The value is a two-element list for read-write channels; the read translation mode is the first element of the list, and the write translation mode is the second element. As a convenience, when setting the translation mode for a read-write channel you can specify a single value that will apply to both reading and writing. When querying the translation mode of a read-write channel, a two-element list will always be returned. The following values are currently supported:

auto As the input translation mode, **auto** treats any of newline (**lf**), carriage return (**cr**), or carriage return followed by a newline (**crlf**) as the end of line representation. The end of line representation can even change from line-to-line, and all cases are translated to a newline. As the output translation mode, **auto** chooses a platform specific representation; for sockets on all platforms Tcl chooses **crlf**, for all Unix flavors, it chooses **lf**, for the Macintosh platform it chooses **cr** and for the various flavors of Windows it chooses **crlf**. The default setting for **−translation** is **auto** for both input and output.

binary No end-of-line translations are performed. This is nearly identical to **lf** mode, except that in addition **binary** mode also sets the end of file character to the empty string, which disables it. See the description of **−eofchar** for more information.

cr The end of a line in the underlying file or device is represented by a single carriage return character. As the input translation mode, **cr** mode converts carriage returns to newline characters. As the output translation mode, **cr** mode translates newline characters to carriage returns. This mode is typically used on Macintosh platforms.

crlf The end of a line in the underlying file or device is represented by a carriage return character followed by a linefeed character. As the input translation mode, **crlf** mode converts carriage-return-linefeed sequences to newline characters. As the output translation mode, **crlf** mode translates newline characters to carriage-return-linefeed sequences. This mode is typically used on Windows platforms and for network connections.

lf The end of a line in the underlying file or device is represented by a single newline (linefeed) character. In this mode no translations occur during either input or output. This mode is typically used on UNIX platforms.

SEE ALSO

close(n), flush(n), gets(n), puts(n), read(n), socket(n)

KEYWORDS

blocking, buffering, carriage return, end of line, flushing, linemode, newline, nonblocking, platform, translation

NAME

fcopy − Copy data from one channel to another.

SYNOPSIS

fcopy *inchan outchan* ?**−size** *size*? ?**−command** *callback*?

DESCRIPTION

The **fcopy** command copies data from one I/O channel, *inchan* to another I/O channel, *outchan*. The **fcopy** command leverages the buffering in the Tcl I/O system to avoid extra copies and to avoid buffering too much data in main memory when copying large files to slow destinations like network sockets.

The **fcopy** command transfers data from *inchan* until end of file or *size* bytes have been transferred. If no **−size** argument is given, then the copy goes until end of file. All the data read from *inchan* is copied to *outchan*. Without the **−command** option, **fcopy** blocks until the copy is complete and returns the number of bytes written to *outchan*.

The **−command** argument makes **fcopy** work in the background. In this case it returns immediately and the *callback* is invoked later when the copy completes. The *callback* is called with one or two additional arguments that indicates how many bytes were written to *outchan*. If an error occurred during the background copy, the second argument is the error string associated with the error. With a background copy, it is not necessary to put *inchan* or *outchan* into non-blocking mode; the **fcopy** command takes care of that automatically. However, it is necessary to enter the event loop by using the **vwait** command or by using Tk.

You are not allowed to do other I/O operations with *inchan* or *outchan* during a background fcopy. If either *inchan* or *outchan* get closed while the copy is in progress, the current copy is stopped and the command callback is *not* made. If *inchan* is closed, then all data already queued for *outchan* is written out.

Note that *inchan* can become readable during a background copy. You should turn off any **fileevent** handlers during a background copy so those handlers do not interfere with the copy. Any I/O attempted by a **fileevent** handler will get a "channel busy" error.

Fcopy translates end-of-line sequences in *inchan* and *outchan* according to the **−translation** option for these channels. See the manual entry for **fconfigure** for details on the **−translation** option. The translations mean that the number of bytes read from *inchan* can be different than the number of bytes written to *outchan*. Only the number of bytes written to *outchan* is reported, either as the return value of a synchronous **fcopy** or as the argument to the callback for an asynchronous **fcopy**.

EXAMPLE

This first example shows how the callback gets passed the number of bytes transferred. It also uses vwait to put the application into the event loop. Of course, this simplified example could be done without the command callback.

```
proc Cleanup {in out bytes {error {}}} {
   global total
   set total $bytes
   close $in
   close $out
   if {[string length $error] != 0} {
         # error occurred during the copy
   }
}
set in [open $file1]
```

```
set out [socket $server $port]
fcopy $in $out -command [list Cleanup $in $out]
vwait total
```

The second example copies in chunks and tests for end of file in the command callback

```
proc CopyMore {in out chunk bytes {error {}}} {
   global total done
   incr total $bytes
   if {([string length $error] != 0) || [eof $in] {
         set done $total
         close $in
         close $out
   } else {
         fcopy $in $out -command [list CopyMore $in $out $chunk] \
            -size $chunk
   }
}
set in [open $file1]
set out [socket $server $port]
set chunk 1024
set total 0
fcopy $in $out -command [list CopyMore $in $out $chunk] -size $chunk
vwait done
```

SEE ALSO

eof(n), fblocked(n), fconfigure(n)

KEYWORDS

blocking, channel, end of line, end of file, nonblocking, read, translation

NAME

file − Manipulate file names and attributes

SYNOPSIS

file *option name* ?*arg arg ...*?

DESCRIPTION

This command provides several 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 done before executing the command (see the manual entry for **filename** 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 attributes *name*

file attributes *name* ?**option**?
file attributes *name* ?**option value option value...**?
This subcommand returns or sets platform specific values associated with a file. The first form returns a list of the platform specific flags and their values. The second form returns the value for the specific option. The third form sets one or more of the values. The values are as follows:

On Unix, **-group** gets or sets the group name for the file. A group id can be given to the command, but it returns a group name. **-owner** gets or sets the user name of the owner of the file. The command returns the owner name, but the numerical id can be passed when setting the owner. **-permissions** sets or retrieves the octal code that chmod(1) uses. This command does not support the symbolic attributes for chmod(1) at this time.

On Windows, **-archive** gives the value or sets or clears the archive attribute of the file. **-hidden** gives the value or sets or clears the hidden attribute of the file. **-longname** will expand each path element to its long version. This attribute cannot be set. **-readonly** gives the value or sets or clears the readonly attribute of the file. **-shortname** gives a string where every path element is replaced with its short (8.3) version of the name. This attribute cannot be set. **-system** gives or sets or clears the value of the system attribute of the file.

On Macintosh, **-creator** gives or sets the Finder creator type of the file. **-hidden** gives or sets or clears the hidden attribute of the file. **-readonly** gives or sets or clears the readonly attribute of the file. Note that directories can only be locked if File Sharing is turned on. **-type** gives or sets the Finder file type for the file.

file copy ?−**force**? ?− −? *source target*
file copy ?−**force**? ?− −? *source* ?*source ...*? *targetDir*

The first form makes a copy of the file or directory *source* under the pathname *target*. If *target* is an existing directory, then the second form is used. The second form makes a copy inside *targetDir* of each *source* file listed. If a directory is specified as a *source*, then the contents of the directory will be recursively copied into *targetDir*. Existing files will not be overwritten unless the −**force** option is specified. Trying to overwrite a non-empty directory, overwrite a directory with a file, or a file with a directory will all result in errors even if −*force* was specified. Arguments are processed in the order specified, halting at the first error, if any. A − − marks the end of switches; the argument following the − − will be treated as a *source* even if it starts with a −.

file delete ?−**force**? ?− −? *pathname* ?*pathname ...* ?

Removes the file or directory specified by each *pathname* argument. Non-empty directories will

be removed only if the **−force** option is specified. Trying to delete a non-existant file is not considered an error. Trying to delete a read-only file will cause the file to be deleted, even if the **−force** flags is not specified. Arguments are processed in the order specified, halting at the first error, if any. A **− −** marks the end of switches; the argument following the **− −** will be treated as a *pathname* even if it starts with a **−**.

file dirname *name*

Returns a name comprised of all of the path components in *name* excluding the last element. If *name* is a relative file name and only contains one path element, then returns "**.**" (or "**:**" on the Macintosh). If *name* refers to a root directory, then the root directory is returned. For example,

file dirname c:/

returns **c:/**.

Note that tilde substitution will only be performed if it is necessary to complete the command. For example,

file dirname ~/src/foo.c

returns **~/src**, whereas

file dirname ~

returns **/home** (or something similar).

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 the last element of *name*. If there is no dot in the last element of *name* then returns the empty string.

file isdirectory *name*

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

file isfile *name*

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

file join *name* ?*name* ...?

Takes one or more file names and combines them, using the correct path separator for the current platform. If a particular *name* is relative, then it will be joined to the previous file name argument. Otherwise, any earlier arguments will be discarded, and joining will proceed from the current argument. For example,

file join a b /foo bar

returns **/foo/bar**.

Note that any of the names can contain separators, and that the result is always canonical for the current platform: **/** for Unix and Windows, and **:** for Macintosh.

file lstat *name varName*

Same as **stat** option (see below) except 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 **stat** option.

file mkdir *dir* ?*dir* ...?

Creates each directory specified. For each pathname *dir* specified, this command will create all non-existing parent directories as well as *dir* itself. If an existing directory is specified, then no action is taken and no error is returned. Trying to overwrite an existing file with a directory will

result in an error. Arguments are processed in the order specified, halting at the first error, if any.

file mtime *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 nativename *name*
Returns the platform-specific name of the file. This is useful if the filename is needed to pass to a platform-specific call, such as exec under Windows or AppleScript on the Macintosh.

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

file pathtype *name*
Returns one of **absolute**, **relative**, **volumerelative**. If *name* refers to a specific file on a specific volume, the path type will be **absolute**. If *name* refers to a file relative to the current working directory, then the path type will be **relative**. If *name* refers to a file relative to the current working directory on a specified volume, or to a specific file on the current working volume, then the file type is **volumerelative**.

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 it points 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 rename ?**−force**? ?**− −**? *source target*
file rename ?**−force**? ?**− −**? *source* ?*source ...*? *targetDir*
The first form takes the file or directory specified by pathname *source* and renames it to *target*, moving the file if the pathname *target* specifies a name in a different directory. If *target* is an existing directory, then the second form is used. The second form moves each *source* file or directory into the directory *targetDir*. Existing files will not be overwritten unless the **−force** option is specified. Trying to overwrite a non-empty directory, overwrite a directory with a file, or a file with a directory will all result in errors. Arguments are processed in the order specified, halting at the first error, if any. A **− −** marks the end of switches; the argument following the **− −** will be treated as a *source* even if it starts with a **−**.

file rootname *name*
Returns all of the characters in *name* up to but not including the last ''.'' character in the last component of name. If the last component of *name* doesn't contain a dot, then 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 split *name*
Returns a list whose elements are the path components in *name*. The first element of the list will have the same path type as *name*. All other elements will be relative. Path separators will be discarded unless they are needed ensure that an element is unambiguously relative. For example, under Unix

file split /foo/~bar/baz

returns **/ foo ./~bar baz** to ensure that later commands that use the third component do not attempt to perform tilde substitution.

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; see 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 directory separator. If *name* contains no separators then 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 volume

Returns the absolute paths to the volumes mounted on the system, as a proper Tcl list. On the Macintosh, this will be a list of the mounted drives, both local and network. N.B. if two drives have the same name, they will both appear on the volume list, but there is currently no way, from Tcl, to access any but the first of these drives. On UNIX, the command will always return "/", since all filesystems are locally mounted. On Windows, it will return a list of the available local drives (e.g. {a:/ c:/}).

file writable *name*

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

PORTABILITY ISSUES

Unix

These commands always operate using the real user and group identifiers, not the effective ones.

SEE ALSO

filename

KEYWORDS

attributes, copy files, delete files, directory, file, move files, name, rename files, stat

NAME

fileevent − Execute a script when a channel becomes readable or writable

SYNOPSIS

fileevent *channelId* **readable** ?*script*?

fileevent *channelId* **writable** ?*script*?

DESCRIPTION

This command is used to create *file event handlers*. A file event handler is a binding between a channel and a script, such that the script is evaluated whenever the channel becomes readable or writable. File event handlers are most commonly used to allow data to be received from another process on an event-driven basis, so that the receiver can continue to interact with the user while waiting for the data to arrive. If an application invokes **gets** or **read** on a blocking channel when there is no input data available, the process will block; until the input data arrives, it will not be able to service other events, so it will appear to the user to ''freeze up''. With **fileevent**, the process can tell when data is present and only invoke **gets** or **read** when they won't block.

The *channelId* argument to **fileevent** refers to an open channel, such as the return value from a previous **open** or **socket** command. If the *script* argument is specified, then **fileevent** creates a new event handler: *script* will be evaluated whenever the channel becomes readable or writable (depending on the second argument to **fileevent**). In this case **fileevent** returns an empty string. The **readable** and **writable** event handlers for a file are independent, and may be created and deleted separately. However, there may be at most one **readable** and one **writable** handler for a file at a given time in a given interpreter. If **fileevent** is called when the specified handler already exists in the invoking interpreter, the new script replaces the old one.

If the *script* argument is not specified, **fileevent** returns the current script for *channelId*, or an empty string if there is none. If the *script* argument is specified as an empty string then the event handler is deleted, so that no script will be invoked. A file event handler is also deleted automatically whenever its channel is closed or its interpreter is deleted.

A channel is considered to be readable if there is unread data available on the underlying device. A channel is also considered to be readable if there is unread data in an input buffer, except in the special case where the most recent attempt to read from the channel was a **gets** call that could not find a complete line in the input buffer. This feature allows a file to be read a line at a time in nonblocking mode using events. A channel is also considered to be readable if an end of file or error condition is present on the underlying file or device. It is important for *script* to check for these conditions and handle them appropriately; for example, if there is no special check for end of file, an infinite loop may occur where *script* reads no data, returns, and is immediately invoked again.

A channel is considered to be writable if at least one byte of data can be written to the underlying file or device without blocking, or if an error condition is present on the underlying file or device.

Event-driven I/O works best for channels that have been placed into nonblocking mode with the **fconfigure** command. In blocking mode, a **puts** command may block if you give it more data than the underlying file or device can accept, and a **gets** or **read** command will block if you attempt to read more data than is ready; no events will be processed while the commands block. In nonblocking mode **puts**, **read**, and **gets** never block. See the documentation for the individual commands for information on how they handle blocking and nonblocking channels.

The script for a file event is executed at global level (outside the context of any Tcl procedure) in the interpreter in which the **fileevent** command was invoked. If an error occurs while executing the script then the **bgerror** mechanism is used to report the error. In addition, the file event handler is deleted if it ever returns an error; this is done in order to prevent infinite loops due to buggy handlers.

CREDITS

fileevent is based on the **addinput** command created by Mark Diekhans.

SEE ALSO

bgerror, fconfigure, gets, puts, read

KEYWORDS

asynchronous I/O, blocking, channel, event handler, nonblocking, readable, script, writable.

NAME

filename − File name conventions supported by Tcl commands

INTRODUCTION

All Tcl commands and C procedures that take file names as arguments expect the file names to be in one of three forms, depending on the current platform. On each platform, Tcl supports file names in the standard forms(s) for that platform. In addition, on all platforms, Tcl supports a Unix-like syntax intended to provide a convenient way of constructing simple file names. However, scripts that are intended to be portable should not assume a particular form for file names. Instead, portable scripts must use the **file split** and **file join** commands to manipulate file names (see the **file** manual entry for more details).

PATH TYPES

File names are grouped into three general types based on the starting point for the path used to specify the file: absolute, relative, and volume-relative. Absolute names are completely qualified, giving a path to the file relative to a particular volume and the root directory on that volume. Relative names are unqualified, giving a path to the file relative to the current working directory. Volume-relative names are partially qualified, either giving the path relative to the root directory on the current volume, or relative to the current directory of the specified volume. The **file pathtype** command can be used to determine the type of a given path.

PATH SYNTAX

The rules for native names depend on the value reported in the Tcl array element **tcl_platform(platform)**:

mac On Apple Macintosh systems, Tcl supports two forms of path names. The normal Mac style names use colons as path separators. Paths may be relative or absolute, and file names may contain any character other than colon. A leading colon causes the rest of the path to be interpreted relative to the current directory. If a path contains a colon that is not at the beginning, then the path is interpreted as an absolute path. Sequences of two or more colons anywhere in the path are used to construct relative paths where **::** refers to the parent of the current directory, **:::** refers to the parent of the parent, and so forth.

In addition to Macintosh style names, Tcl also supports a subset of Unix-like names. If a path contains no colons, then it is interpreted like a Unix path. Slash is used as the path separator. The file name **.** refers to the current directory, and **..** refers to the parent of the current directory. However, some names like **/** or **/..** have no mapping, and are interpreted as Macintosh names. In general, commands that generate file names will return Macintosh style names, but commands that accept file names will take both Macintosh and Unix-style names.

The following examples illustrate various forms of path names:

:	Relative path to the current folder.
MyFile	Relative path to a file named **MyFile** in the current folder.
MyDisk:MyFile	Absolute path to a file named **MyFile** on the device named **MyDisk**.
:MyDir:MyFile	Relative path to a file name **MyFile** in a folder named **MyDir** in the current folder.
::MyFile	Relative path to a file named **MyFile** in the folder above the current folder.
:::MyFile	Relative path to a file named **MyFile** in the folder two levels above the current folder.
/MyDisk/MyFile	Absolute path to a file named **MyFile** on the device named **MyDisk**.
../MyFile	Relative path to a file named **MyFile** in the folder above the current folder.

unix On Unix platforms, Tcl uses path names where the components are separated by slashes. Path names may be relative or absolute, and file names may contain any character other than slash. The file names **.** and **..** are special and refer to the current directory and the parent of the current directory respectively. Multiple adjacent slash characters are interpreted as a single separator. The following examples illustrate various forms of path names:

/	Absolute path to the root directory.
/etc/passwd	Absolute path to the file named **passwd** in the directory **etc** in the root directory.
.	Relative path to the current directory.
foo	Relative path to the file **foo** in the current directory.
foo/bar	Relative path to the file **bar** in the directory **foo** in the current directory.
../foo	Relative path to the file **foo** in the directory above the current directory.

windows On Microsoft Windows platforms, Tcl supports both drive-relative and UNC style names. Both / and \ may be used as directory separators in either type of name. Drive-relative names consist of an optional drive specifier followed by an absolute or relative path. UNC paths follow the general form **\\servername\sharename\path\file**. In both forms, the file names **.** and **..** are special and refer to the current directory and the parent of the current directory respectively. The following examples illustrate various forms of path names:

\\Host\share/file	Absolute UNC path to a file called **file** in the root directory of the export point **share** on the host **Host**.
c:foo	Volume-relative path to a file **foo** in the current directory on drive **c**.
c:/foo	Absolute path to a file **foo** in the root directory of drive **c**.
foo\bar	Relative path to a file **bar** in the **foo** directory in the current directory on the current volume.
\foo	Volume-relative path to a file **foo** in the root directory of the current volume.

TILDE SUBSTITUTION

In addition to the file name rules described above, Tcl also supports *csh*-style tilde substitution. If a file name starts with a tilde, then the file name will be interpreted as if the first element is replaced with the location of the home directory for the given user. If the tilde is followed immediately by a separator, then the **$HOME** environment variable is substituted. Otherwise the characters between the tilde and the next separator are taken as a user name, which is used to retrieve the user's home directory for substitution.

The Macintosh and Windows platforms do not support tilde substitution when a user name follows the tilde. On these platforms, attempts to use a tilde followed by a user name will generate an error. File names that have a tilde without a user name will be substituted using the **$HOME** environment variable, just like for Unix.

PORTABILITY ISSUES

Not all file systems are case sensitive, so scripts should avoid code that depends on the case of characters in a file name. In addition, the character sets allowed on different devices may differ, so scripts should choose file names that do not contain special characters like: **<>:"/\|**. The safest approach is to use names consisting of alphanumeric characters only. Also Windows 3.1 only supports file names with a root of no more than 8 characters and an extension of no more than 3 characters.

KEYWORDS

current directory, absolute file name, relative file name, volume-relative file name, portability

NAME

flush − Flush buffered output for a channel

SYNOPSIS

flush *channelId*

DESCRIPTION

Flushes any output that has been buffered for *channelId*. *ChannelId* must be a channel identifier such as returned by a previous **open** or **socket** command, and it must have been opened for writing. If the channel is in blocking mode the command does not return until all the buffered output has been flushed to the channel. If the channel is in nonblocking mode, the command may return before all buffered output has been flushed; the remainder will be flushed in the background as fast as the underlying file or device is able to absorb it.

SEE ALSO

open(n), socket(n)

KEYWORDS

blocking, buffer, channel, flush, nonblocking, output

NAME

for − ‘‘For’’ loop

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.

Note: *test* should almost always be enclosed in braces. If not, variable substitutions will be made before the **for** command starts executing, which means that variable changes made by the loop body will not be considered in the expression. This is likely to result in an infinite loop. If *test* is enclosed in braces, variable substitutions are delayed until the expression is evaluated (before each loop iteration), so changes in the variables will be visible. For an example, try the following script with and without the braces around **$x<10**:

```
for {set x 0} {$x<10} {incr x} {
    puts "x is $x"
}
```

KEYWORDS

for, iteration, looping

NAME

foreach − Iterate over all elements in one or more lists

SYNOPSIS

foreach *varname list body*
foreach *varlist1 list1* ?*varlist2 list2 ...*? *body*

DESCRIPTION

The **foreach** command implements a loop where the loop variable(s) take on values from one or more lists. In the simplest case there is one loop variable, *varname*, and one list, *list*, that is a list of values to assign to *varname*. The *body* argument is a Tcl script. For each element of *list* (in order from first to last), **foreach** assigns the contents of the element to *varname* as if the **lindex** command had been used to extract the element, then calls the Tcl interpreter to execute *body*.

In the general case there can be more than one value list (e.g., *list1* and *list2*), and each value list can be associated with a list of loop variables (e.g., *varlist1* and *varlist2*). During each iteration of the loop the variables of each *varlist* are assigned consecutive values from the corresponding *list*. Values in each *list* are used in order from first to last, and each value is used exactly once. The total number of loop iterations is large enough to use up all the values from all the value lists. If a value list does not contain enough elements for each of its loop variables in each iteration, empty values are used for the missing elements.

The **break** and **continue** statements may be invoked inside *body*, with the same effect as in the **for** command. **Foreach** returns an empty string.

EXAMPLES

The following loop uses i and j as loop variables to iterate over pairs of elements of a single list.

```
set x {}
foreach {i j} {a b c d e f} {
   lappend x $j $i
}
# The value of x is "b a d c f e"
# There are 3 iterations of the loop.
```

The next loop uses i and j to iterate over two lists in parallel.

```
set x {}
foreach i {a b c} j {d e f g} {
   lappend x $i $j
}
# The value of x is "a d b e c f {} g"
# There are 4 iterations of the loop.
```

The two forms are combined in the following example.

```
set x {}
foreach i {a b c} {j k} {d e f g} {
   lappend x $i $j $k
}
# The value of x is "a d e b f g c {} {}"
# There are 3 iterations of the loop.
```

KEYWORDS

foreach, iteration, list, looping

NAME

format − Format a string in the style of sprintf

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 specifiers as in **sprintf**, and the additional arguments, if any, provide values to be substituted into 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 specifier. The conversion specifier 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 specifier. If there are multiple conversion specifiers in the format string, then each one controls the conversion of one additional *arg*. The **format** command must be given enough *arg*s to meet the needs of all of the conversion specifiers in *formatString*.

Each conversion specifier may contain up to six different parts: an XPG3 position specifier, 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 specifier requires multiple arguments because of * characters in the specifier then successive arguments are used, starting with the argument given by the number. This follows the XPG3 conventions for positional specifiers. If there are any positional specifiers in *formatString* then all of the specifiers must be positional.

The second portion of a conversion specifier 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 specifier 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 specifier 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 omitted unless 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 the trailing 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 fifth part of a conversion specifier 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 specifier 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.yyy***e**±*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 specifier.

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** specifiers 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.

KEYWORDS

conversion specifier, format, sprintf, string, substitution

NAME

gets − Read a line from a channel

SYNOPSIS

gets *channelId* ?*varName*?

DESCRIPTION

This command reads the next line from *channelId*, returns everything in the line up to (but not including) the end-of-line character(s), and discards the end-of-line character(s). If *varName* is omitted the line is returned as the result of the command. 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 returned.

If end of file occurs while scanning for an end of line, the command returns whatever input is available up to the end of file. If *channelId* is in nonblocking mode and there is not a full line of input available, the command returns an empty string and does not consume any input. If *varName* is specified and an empty string is returned in *varName* because of end-of-file or because of insufficient data in nonblocking mode, then the return count is -1. Note that if *varName* is not specified then the end-of-file and no-full-line-available cases can produce the same results as if there were an input line consisting only of the end-of-line character(s). The **eof** and **fblocked** commands can be used to distinguish these three cases.

SEE ALSO

eof(n), fblocked(n)

KEYWORDS

blocking, channel, end of file, end of line, line, nonblocking, read

NAME

glob − Return names of files that match patterns

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:

−nocomplain 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 you want 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.

PORTABILITY ISSUES

Unlike other Tcl commands that will accept both network and native style names (see the **filename** manual entry for details on how native and network names are specified), the **glob** command only accepts native names. Also, for Windows UNC names, the servername and sharename components of the path may not contain ?, ∗, or [] constructs.

KEYWORDS

exist, file, glob, pattern

NAME

global − Access global variables

SYNOPSIS

global *varname* ?*varname* ...?

DESCRIPTION

This command is ignored unless a Tcl procedure is being interpreted. If so then it declares the given *varname*'s to be global variables rather than local ones. Global variables are variables in the global namespace. For the duration of the current procedure (and only while executing in the current procedure), any reference to any of the *varname*s will refer to the global variable by the same name.

SEE ALSO

namespace(n), variable(n)

KEYWORDS

global, namespace, procedure, variable

NAME

history − Manipulate the history list

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). Event **0** refers to the current event.

[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 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 clear
Erase the history list. The current keep limit is retained. The history event numbers are reset.

history event ?*event*?
Returns the value of the event given by *event*. *Event* defaults to **−1**.

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. If *count* is not specified, the current keep limit is returned.

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 REVISION

Pre-8.0 Tcl had a complex history revision mechanism. The current mechanism is more limited, and the old history operations **substitute** and **words** have been removed. (As a consolation, the **clear** operation was added.)

The history option **redo** results in much simpler ‘‘history revision’’. When this option is invoked then the most recent event is modified to eliminate the history command and replace it with the result of the history command. If you want to redo an event without modifying history, then use the **event** operation to retrieve some event, and the **add** operation to add it to history and execute it.

KEYWORDS

event, history, record

NAME

Http − Client-side implementation of the HTTP/1.0 protocol.

SYNOPSIS

package require http ?2.0?

::http::config *?options?*

::http::geturl *url ?options?*

::http::formatQuery *list*

::http::reset *token*

::http::wait *token*

::http::status *token*

::http::size *token*

::http::code *token*

::http::data *token*

DESCRIPTION

The **http** package provides the client side of the HTTP/1.0 protocol. The package implements the GET, POST, and HEAD operations of HTTP/1.0. It allows configuration of a proxy host to get through firewalls. The package is compatible with the **Safesock** security policy, so it can be used by untrusted applets to do URL fetching from a restricted set of hosts.

The **::http::geturl** procedure does a HTTP transaction. Its *options* determine whether a GET, POST, or HEAD transaction is performed. The return value of **::http::geturl** is a token for the transaction. The value is also the name of an array in the ::http namespace
that contains state information about the transaction. The elements of this array are described in the STATE ARRAY section.

If the **-command** option is specified, then the HTTP operation is done in the background. **::http::geturl** returns immediately after generating the HTTP request and the callback is invoked when the transaction completes. For this to work, the Tcl event loop must be active. In Tk applications this is always true. For pure-Tcl applications, the caller can use **::http::wait** after calling **::http::geturl** to start the event loop.

COMMANDS

::http::config ?*options*?

The **::http::config** command is used to set and query the name of the proxy server and port, and the User-Agent name used in the HTTP requests. If no options are specified, then the current configuration is returned. If a single argument is specified, then it should be one of the flags described below. In this case the current value of that setting is returned. Otherwise, the options should be a set of flags and values that define the configuration:

−accept *mimetypes*

The Accept header of the request. The default is ∗/∗, which means that all types of documents are accepted. Otherwise you can supply a comma separated list of mime type patterns that you are willing to receive. For example, "image/gif, image/jpeg, text/∗".

−proxyhost *hostname*
The name of the proxy host, if any. If this value is the empty string, the URL host is contacted directly.

−proxyport *number*
The proxy port number.

−proxyfilter *command*
The command is a callback that is made during **::http::geturl** to determine if a proxy is required for a given host. One argument, a host name, is added to *command* when it is invoked. If a proxy is required, the callback should return a two element list containing the proxy server and proxy port. Otherwise the filter should return an empty list. The default filter returns the values of the **−proxyhost** and **−proxyport** settings if they are non-empty.

−useragent *string*
The value of the User-Agent header in the HTTP request. The default is **"Tcl http client package 2.0."**

::http::geturl *url* ?*options*?
The **::http::geturl** command is the main procedure in the package. The **−query** option causes a POST operation and the **−validate** option causes a HEAD operation; otherwise, a GET operation is performed. The **::http::geturl** command returns a *token* value that can be used to get information about the transaction. See the STATE ARRAY section for details. The **::http::geturl** command blocks until the operation completes, unless the **−command** option specifies a callback that is invoked when the HTTP transaction completes. **::http::geturl** takes several options:

−blocksize *size*
The blocksize used when reading the URL. At most *size* bytes are read at once. After each block, a call to the **−progress** callback is made.

−channel *name*
Copy the URL contents to channel *name* instead of saving it in **state(body)**.

−command *callback*
Invoke *callback* after the HTTP transaction completes. This option causes **::http::geturl** to return immediately. The *callback* gets an additional argument that is the *token* returned from **::http::geturl**. This token is the name of an array that is described in the STATE ARRAY section. Here is a template for the callback:

```
proc httpCallback {token} {
    upvar #0 $token state
    # Access state as a Tcl array
}
```

−handler *callback*
Invoke *callback* whenever HTTP data is available; if present, nothing else will be done with the HTTP data. This procedure gets two additional arguments: the socket for the HTTP data and the *token* returned from **::http::geturl**. The token is the name of a global array that is described in the STATE ARRAY section. The procedure is expected to return the number of bytes read from the socket. Here is a template for the callback:

```
proc httpHandlerCallback {socket token} {
    upvar #0 $token state
    # Access socket, and state as a Tcl array
    ...
    (example: set data [read $socket 1000];set nbytes [string length $data])
    ...
```

```
        return nbytes
    }
```

−headers *keyvaluelist*
: This option is used to add extra headers to the HTTP request. The *keyvaluelist* argument must be a list with an even number of elements that alternate between keys and values. The keys become header field names. Newlines are stripped from the values so the header cannot be corrupted. For example, if *keyvaluelist* is **Pragma no-cache** then the following header is included in the HTTP request:
Pragma: no-cache

−progress *callback*
: The *callback* is made after each transfer of data from the URL. The callback gets three additional arguments: the *token* from **::http::geturl**, the expected total size of the contents from the **Content-Length** meta-data, and the current number of bytes transferred so far. The expected total size may be unknown, in which case zero is passed to the callback. Here is a template for the progress callback:

```
proc httpProgress {token total current} {
    upvar #0 $token state
}
```

−query *query*
: This flag causes **::http::geturl** to do a POST request that passes the *query* to the server. The *query* must be a x-url-encoding formatted query. The **::http::formatQuery** procedure can be used to do the formatting.

−timeout *milliseconds*
: If *milliseconds* is non-zero, then **::http::geturl** sets up a timeout to occur after the specified number of milliseconds. A timeout results in a call to **::http::reset** and to the **-command** callback, if specified. The return value of **::http::status** is **timeout** after a timeout has occurred.

−validate *boolean*
: If *boolean* is non-zero, then **::http::geturl** does an HTTP HEAD request. This request returns meta information about the URL, but the contents are not returned. The meta information is available in the **state(meta)** variable after the transaction. See the STATE ARRAY section for details.

::http::formatQuery *key value* ?*key value* ...?
: This procedure does x-url-encoding of query data. It takes an even number of arguments that are the keys and values of the query. It encodes the keys and values, and generates one string that has the proper & and = separators. The result is suitable for the **−query** value passed to **::http::geturl**.

::http::reset *token* ?*why*?
: This command resets the HTTP transaction identified by *token*, if any. This sets the **state(status)** value to *why*, which defaults to **reset**, and then calls the registered **−command** callback.

::http::wait *token*
: This is a convenience procedure that blocks and waits for the transaction to complete. This only works in trusted code because it uses **vwait**.

::http::data *token*
: This is a convenience procedure that returns the **body** element (i.e., the URL data) of the state array.

::http::status *token*
: This is a convenience procedure that returns the **status** element of the state array.

::http::code *token*
This is a convenience procedure that returns the **http** element of the state array.

::http::size *token*
This is a convenience procedure that returns the **currentsize** element of the state array.

STATE ARRAY

The **::http::geturl** procedure returns a *token* that can be used to get to the state of the HTTP transaction in the form of a Tcl array. Use this construct to create an easy-to-use array variable:

```
upvar #0 $token state
```

The following elements of the array are supported:

body The contents of the URL. This will be empty if the **−channel** option has been specified. This value is returned by the **::http::data** command.

currentsize
The current number of bytes fetched from the URL. This value is returned by the **::http::size** command.

error If defined, this is the error string seen when the HTTP transaction was aborted.

http The HTTP status reply from the server. This value is returned by the **::http::code** command. The format of this value is:

code string

The *code* is a three-digit number defined in the HTTP standard. A code of 200 is OK. Codes beginning with 4 or 5 indicate errors. Codes beginning with 3 are redirection errors. In this case the **Location** meta-data specifies a new URL that contains the requested information.

meta The HTTP protocol returns meta-data that describes the URL contents. The **meta** element of the state array is a list of the keys and values of the meta-data. This is in a format useful for initializing an array that just contains the meta-data:

```
array set meta $state(meta)
```

Some of the meta-data keys are listed below, but the HTTP standard defines more, and servers are free to add their own.

Content-Type
The type of the URL contents. Examples include **text/html**, **image/gif, application/postscript** and **application/x-tcl**.

Content-Length
The advertised size of the contents. The actual size obtained by **::http::geturl** is available as **state(size)**.

Location
An alternate URL that contains the requested data.

status Either **ok**, for successful completion, **reset** for user-reset, or **error** for an error condition. During the transaction this value is the empty string.

totalsize
A copy of the **Content-Length** meta-data value.

type A copy of the **Content-Type** meta-data value.

url The requested URL.

EXAMPLE

```
# Copy a URL to a file and print meta-data
proc ::http::copy { url file {chunk 4096} } {
  set out [open $file w]
```

```
    set token [geturl $url -channel $out -progress ::http::Progress \
      -blocksize $chunk]
    close $out
    # This ends the line started by http::Progress
    puts stderr ""
    upvar #0 $token state
    set max 0
    foreach {name value} $state(meta) {
      if {[string length $name] > $max} {
        set max [string length $name]
      }
      if {[regexp -nocase ^location$ $name]} {
        # Handle URL redirects
        puts stderr "Location:$value"
        return [copy [string trim $value] $file $chunk]
      }
    }
    incr max
    foreach {name value} $state(meta) {
      puts [format "%-*s %s" $max $name: $value]
    }

    return $token
}
proc ::http::Progress {args} {
    puts -nonewline stderr . ; flush stderr
}
```

SEE ALSO

safe(n), socket(n), safesock(n)

KEYWORDS

security policy, socket

NAME

if − Execute scripts conditionally

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*.

KEYWORDS

boolean, conditional, else, false, if, true

NAME

incr − Increment the value of a variable

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 also returned as result.

KEYWORDS

add, increment, variable, value

NAME

info − Return information about the state of the Tcl interpreter

SYNOPSIS

info *option* ?*arg arg* ...?

DESCRIPTION

This command provides information about various internals of the Tcl interpreter. The legal *option*'s (which may be abbreviated) are:

info args *procname*

Returns a list containing the names of the arguments to procedure *procname*, in order. *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 *pattern* isn't specified, returns a list of names of all the Tcl commands in the current namespace, including both the built-in commands written in C and the command procedures defined using the **proc** command. If *pattern* is specified, only those names matching *pattern* are returned. Matching is determined using the same rules as for **string match**. *pattern* can be a qualified name like **Foo::print***. That is, it may specify a particular namespace using a sequence of namespace names separated by **::**s, and may have pattern matching special characters at the end to specify a set of commands in that namespace. If *pattern* is a qualified name, the resulting list of command names has each one qualified with the name of the specified namespace.

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 doesn't 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* into 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 *pattern* isn't specified, returns a list of all the names of currently-defined global variables. Global variables are variables in the global namespace. If *pattern* is specified, only those names matching *pattern* are returned. Matching is determined using the same rules as for **string match**.

info hostname

Returns the name of the computer on which this invocation is being executed.

info level ?*number*?

If *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 *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 *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. This is actually the value of the **tcl_library** variable and may be changed by setting **tcl_library**. See the **tclvars** manual entry for more information.

info loaded ?*interp*?
Returns a list describing all of the packages that have been loaded into *interp* with the **load** command. Each list element is a sub-list with two elements consisting of the name of the file from which the package was loaded and the name of the package. For statically-loaded packages the file name will be an empty string. If *interp* is omitted then information is returned for all packages loaded in any interpreter in the process. To get a list of just the packages in the current interpreter, specify an empty string for the *interp* argument.

info locals ?*pattern*?
If *pattern* isn't specified, 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 *pattern* is specified, only those names matching *pattern* are returned. Matching is determined using the same rules as for **string match**.

info nameofexecutable
Returns the full path name of the binary file from which the application was invoked. If Tcl was unable to identify the file, then an empty string is returned.

info patchlevel
Returns the value of the global variable **tcl_patchLevel**; see the **tclvars** manual entry for more information.

info procs ?*pattern*?
If *pattern* isn't specified, returns a list of all the names of Tcl command procedures in the current namespace. If *pattern* is specified, only those procedure names in the current namespace matching *pattern* are returned. Matching is determined using the same rules as for **string match**.

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 sharedlibextension
Returns the extension used on this platform for the names of files containing shared libraries (for example, **.so** under Solaris). If shared libraries aren't supported on this platform then an empty string is returned.

info tclversion
Returns the value of the global variable **tcl_version**; see the **tclvars** manual entry for more information.

info vars ?*pattern*?
If *pattern* isn't specified, returns a list of all the names of currently-visible variables. This includes locals and currently-visible globals. If *pattern* is specified, only those names matching *pattern* are returned. Matching is determined using the same rules as for **string match**. *pattern* can be a qualified name like **Foo::option***. That is, it may specify a particular namespace using a sequence

of namespace names separated by **::**s, and may have pattern matching special characters at the end to specify a set of variables in that namespace. If *pattern* is a qualified name, the resulting list of variable names has each matching namespace variable qualified with the name of its namespace.

KEYWORDS

command, information, interpreter, level, namespace, procedure, variable

NAME

interp − Create and manipulate Tcl interpreters

SYNOPSIS

interp *option* ?*arg arg ...*?

DESCRIPTION

This command makes it possible to create one or more new Tcl interpreters that co-exist with the creating interpreter in the same application. The creating interpreter is called the *master* and the new interpreter is called a *slave*. A master can create any number of slaves, and each slave can itself create additional slaves for which it is master, resulting in a hierarchy of interpreters.

Each interpreter is independent from the others: it has its own name space for commands, procedures, and global variables. A master interpreter may create connections between its slaves and itself using a mechanism called an *alias*. An *alias* is a command in a slave interpreter which, when invoked, causes a command to be invoked in its master interpreter or in another slave interpreter. The only other connections between interpreters are through environment variables (the **env** variable), which are normally shared among all interpreters in the application. Note that the name space for files (such as the names returned by the **open** command) is no longer shared between interpreters. Explicit commands are provided to share files and to transfer references to open files from one interpreter to another.

The **interp** command also provides support for *safe* interpreters. A safe interpreter is a slave whose functions have been greatly restricted, so that it is safe to execute untrusted scripts without fear of them damaging other interpreters or the application's environment. For example, all IO channel creation commands and subprocess creation commands are made inaccessible to safe interpreters. See SAFE INTERPRETERS below for more information on what features are present in a safe interpreter. The dangerous functionality is not removed from the safe interpreter; instead, it is *hidden*, so that only trusted interpreters can obtain access to it. For a detailed explanation of hidden commands, see HIDDEN COMMANDS, below. The alias mechanism can be used for protected communication (analogous to a kernel call) between a slave interpreter and its master. See ALIAS INVOCATION, below, for more details on how the alias mechanism works.

A qualified interpreter name is a proper Tcl lists containing a subset of its ancestors in the interpreter hierarchy, terminated by the string naming the interpreter in its immediate master. Interpreter names are relative to the interpreter in which they are used. For example, if **a** is a slave of the current interpreter and it has a slave **a1**, which in turn has a slave **a11**, the qualified name of **a11** in **a** is the list **a1 a11**.

The **interp** command, described below, accepts qualified interpreter names as arguments; the interpreter in which the command is being evaluated can always be referred to as **{}** (the empty list or string). Note that it is impossible to refer to a master (ancestor) interpreter by name in a slave interpreter except through aliases. Also, there is no global name by which one can refer to the first interpreter created in an application. Both restrictions are motivated by safety concerns.

THE INTERP COMMAND

The **interp** command is used to create, delete, and manipulate slave interpreters, and to share or transfer channels between interpreters. It can have any of several forms, depending on the *option* argument:

interp alias *srcPath srcCmd*

Returns a Tcl list whose elements are the *targetCmd* and *arg*s associated with the alias named *srcCmd* (all of these are the values specified when the alias was created; it is possible that the actual source command in the slave is different from *srcCmd* if it was renamed).

interp alias *srcPath srcCmd* **{}**

Deletes the alias for *srcCmd* in the slave interpreter identified by *srcPath*. *srcCmd* refers to the

name under which the alias was created; if the source command has been renamed, the renamed command will be deleted.

interp alias *srcPath srcCmd targetPath targetCmd* ?*arg arg ...*?

This command creates an alias between one slave and another (see the **alias** slave command below for creating aliases between a slave and its master). In this command, either of the slave interpreters may be anywhere in the hierarchy of interpreters under the interpreter invoking the command. *SrcPath* and *srcCmd* identify the source of the alias. *SrcPath* is a Tcl list whose elements select a particular interpreter. For example, ''**a b**'' identifies an interpreter **b**, which is a slave of interpreter **a**, which is a slave of the invoking interpreter. An empty list specifies the interpreter invoking the command. *srcCmd* gives the name of a new command, which will be created in the source interpreter. *TargetPath* and *targetCmd* specify a target interpreter and command, and the *arg* arguments, if any, specify additional arguments to *targetCmd* which are prepended to any arguments specified in the invocation of *srcCmd*. *TargetCmd* may be undefined at the time of this call, or it may already exist; it is not created by this command. The alias arranges for the given target command to be invoked in the target interpreter whenever the given source command is invoked in the source interpreter. See ALIAS INVOCATION below for more details.

interp aliases ?*path*?

This command returns a Tcl list of the names of all the source commands for aliases defined in the interpreter identified by *path*.

interp create ?**−safe**? ?**− −**? ?*path*?

Creates a slave interpreter identified by *path* and a new command, called a *slave command*. The name of the slave command is the last component of *path*. The new slave interpreter and the slave command are created in the interpreter identified by the path obtained by removing the last component from *path*. For example, if *path is* **a b c** then a new slave interpreter and slave command named **c** are created in the interpreter identified by the path **a b**. The slave command may be used to manipulate the new interpreter as described below. If *path* is omitted, Tcl creates a unique name of the form **interp***x*, where *x* is an integer, and uses it for the interpreter and the slave command. If the **−safe** switch is specified (or if the master interpreter is a safe interpreter), the new slave interpreter will be created as a safe interpreter with limited functionality; otherwise the slave will include the full set of Tcl built-in commands and variables. The **− −** switch can be used to mark the end of switches; it may be needed if *path* is an unusual value such as **−safe**. The result of the command is the name of the new interpreter. The name of a slave interpreter must be unique among all the slaves for its master; an error occurs if a slave interpreter by the given name already exists in this master.

interp delete ?*path ...?*

Deletes zero or more interpreters given by the optional *path* arguments, and for each interpreter, it also deletes its slaves. The command also deletes the slave command for each interpreter deleted. For each *path* argument, if no interpreter by that name exists, the command raises an error.

interp eval *path arg* ?*arg ...*?

This command concatenates all of the *arg* arguments in the same fashion as the **concat** command, then evaluates the resulting string as a Tcl script in the slave interpreter identified by *path*. The result of this evaluation (including error information such as the **errorInfo** and **errorCode** variables, if an error occurs) is returned to the invoking interpreter.

interp exists *path*

Returns **1** if a slave interpreter by the specified *path* exists in this master, **0** otherwise. If *path* is omitted, the invoking interpreter is used.

interp expose *path hiddenName* ?*exposedCmdName*?

Makes the hidden command *hiddenName* exposed, eventually bringing it back under a new *exposedCmdName* name (this name is currently accepted only if it is a valid global name space

name without any ::), in the interpreter denoted by *path*. If an exposed command with the targetted name already exists, this command fails. Hidden commands are explained in more detail in HIDDEN COMMANDS, below.

interp hide *path exposedCmdName* ?*hiddenCmdName*?
Makes the exposed command *exposedCmdName* hidden, renaming it to the hidden command *hiddenCmdName*, or keeping the same name if *hiddenCmdName* is not given, in the interpreter denoted by *path*. If a hidden command with the targetted name already exists, this command fails. Currently both *exposedCmdName* and *hiddenCmdName* can not contain namespace qualifiers, or an error is raised. Commands to be hidden by **interp hide** are looked up in the global namespace even if the current namespace is not the global one. This prevents slaves from fooling a master interpreter into hiding the wrong command, by making the current namespace be different from the global one. Hidden commands are explained in more detail in HIDDEN COMMANDS, below.

interp hidden *path*
Returns a list of the names of all hidden commands in the interpreter identified by *path*.

interp invokehidden *path* ?**-global**? *hiddenCmdName* ?*arg ...*?
Invokes the hidden command *hiddenCmdName* with the arguments supplied in the interpreter denoted by *path*. No substitutions or evaluation are applied to the arguments. If the **-global** flag is present, the hidden command is invoked at the global level in the target interpreter; otherwise it is invoked at the current call frame and can access local variables in that and outer call frames. Hidden commands are explained in more detail in HIDDEN COMMANDS, below.

interp issafe ?*path*?
Returns **1** if the interpreter identified by the specified *path* is safe, **0** otherwise.

interp marktrusted *path*
Marks the interpreter identified by *path* as trusted. Does not expose the hidden commands. This command can only be invoked from a trusted interpreter. The command has no effect if the interpreter identified by *path* is already trusted.

interp share *srcPath channelId destPath*
Causes the IO channel identified by *channelId* to become shared between the interpreter identified by *srcPath* and the interpreter identified by *destPath*. Both interpreters have the same permissions on the IO channel. Both interpreters must close it to close the underlying IO channel; IO channels accessible in an interpreter are automatically closed when an interpreter is destroyed.

interp slaves ?*path*?
Returns a Tcl list of the names of all the slave interpreters associated with the interpreter identified by *path*. If *path* is omitted, the invoking interpreter is used.

interp target *path alias*
Returns a Tcl list describing the target interpreter for an alias. The alias is specified with an interpreter path and source command name, just as in **interp alias** above. The name of the target interpreter is returned as an interpreter path, relative to the invoking interpreter. If the target interpreter for the alias is the invoking interpreter then an empty list is returned. If the target interpreter for the alias is not the invoking interpreter or one of its descendants then an error is generated. The target command does not have to be defined at the time of this invocation.

interp transfer *srcPath channelId destPath*
Causes the IO channel identified by *channelId* to become available in the interpreter identified by *destPath* and unavailable in the interpreter identified by *srcPath*.

SLAVE COMMAND

For each slave interpreter created with the **interp** command, a new Tcl command is created in the master interpreter with the same name as the new interpreter. This command may be used to invoke various operations on the interpreter. It has the following general form:

slave command ?*arg arg* ...?

Slave is the name of the interpreter, and *command* and the *arg*s determine the exact behavior of the command. The valid forms of this command are:

slave **aliases**

Returns a Tcl list whose elements are the names of all the aliases in *slave*. The names returned are the *srcCmd* values used when the aliases were created (which may not be the same as the current names of the commands, if they have been renamed).

slave **alias** *srcCmd*

Returns a Tcl list whose elements are the *targetCmd* and *arg*s associated with the alias named *srcCmd* (all of these are the values specified when the alias was created; it is possible that the actual source command in the slave is different from *srcCmd* if it was renamed).

slave **alias** *srcCmd* **{}**

Deletes the alias for *srcCmd* in the slave interpreter. *srcCmd* refers to the name under which the alias was created; if the source command has been renamed, the renamed command will be deleted.

slave **alias** *srcCmd targetCmd* ?*arg* ..?

Creates an alias such that whenever *srcCmd* is invoked in *slave*, *targetCmd* is invoked in the master. The *arg* arguments will be passed to *targetCmd* as additional arguments, prepended before any arguments passed in the invocation of *srcCmd*. See ALIAS INVOCATION below for details.

slave **eval** *arg* ?*arg* ..?

This command concatenates all of the *arg* arguments in the same fashion as the **concat** command, then evaluates the resulting string as a Tcl script in *slave*. The result of this evaluation (including error information such as the **errorInfo** and **errorCode** variables, if an error occurs) is returned to the invoking interpreter.

slave **expose** *hiddenName* ?*exposedCmdName*?

This command exposes the hidden command *hiddenName*, eventually bringing it back under a new *exposedCmdName* name (this name is currently accepted only if it is a valid global name space name without any ::), in *slave*. If an exposed command with the targetted name already exists, this command fails. For more details on hidden commands, see HIDDEN COMMANDS, below.

slave **hide** *exposedCmdName* ?*hiddenCmdName*?

This command hides the exposed command *exposedCmdName*, renaming it to the hidden command *hiddenCmdName*, or keeping the same name if the the argument is not given, in the *slave* interpreter. If a hidden command with the targetted name already exists, this command fails. Currently both *exposedCmdName* and *hiddenCmdName* can not contain namespace qualifiers, or an error is raised. Commands to be hidden are looked up in the global namespace even if the current namespace is not the global one. This prevents slaves from fooling a master interpreter into hiding the wrong command, by making the current namespace be different from the global one. For more details on hidden commands, see HIDDEN COMMANDS, below.

slave **hidden**

Returns a list of the names of all hidden commands in *slave*.

slave **invokehidden** ?**-global** *hiddenName* ?*arg* ..?

This command invokes the hidden command *hiddenName* with the supplied arguments, in *slave*. No substitutions or evaluations are applied to the arguments. If the **-global** flag is given, the

command is invoked at the global level in the slave; otherwise it is invoked at the current call frame and can access local variables in that or outer call frames. For more details on hidden commands, see HIDDEN COMMANDS, below.

slave **issafe**
Returns **1** if the slave interpreter is safe, **0** otherwise.

slave **marktrusted**
Marks the slave interpreter as trusted. Can only be invoked by a trusted interpreter. This command does not expose any hidden commands in the slave interpreter. The command has no effect if the slave is already trusted.

SAFE INTERPRETERS

A safe interpreter is one with restricted functionality, so that is safe to execute an arbitrary script from your worst enemy without fear of that script damaging the enclosing application or the rest of your computing environment. In order to make an interpreter safe, certain commands and variables are removed from the interpreter. For example, commands to create files on disk are removed, and the **exec** command is removed, since it could be used to cause damage through subprocesses. Limited access to these facilities can be provided, by creating aliases to the master interpreter which check their arguments carefully and provide restricted access to a safe subset of facilities. For example, file creation might be allowed in a particular subdirectory and subprocess invocation might be allowed for a carefully selected and fixed set of programs.

A safe interpreter is created by specifying the **−safe** switch to the **interp create** command. Furthermore, any slave created by a safe interpreter will also be safe.

A safe interpreter is created with exactly the following set of built-in commands:

after	**append**	**array**	**break**
case	**catch**	**clock**	**close**
concat	**continue**	**eof**	**error**
eval	**expr**	**fblocked**	**fileevent**
flush	**for**	**foreach**	**format**
gets	**global**	**history**	**if**
incr	**info**	**interp**	**join**
lappend	**lindex**	**linsert**	**list**
llength	**lower**	**lrange**	**lreplace**
lsearch	**lsort**	**package**	**pid**
proc	**puts**	**read**	**rename**
return	**scan**	**seek**	**set**
split	**string**	**subst**	**switch**
tell	**trace**	**unset**	**update**
uplevel	**upvar**	**vwait**	**while**

The following commands are hidden by **interp create** when it creates a safe interpreter:

cd	**exec**	**exit**	**fconfigure**
file	**glob**	**load**	**open**
pwd	**socket**	**source**	**vwait**

These commands can be recreated later as Tcl procedures or aliases, or re-exposed by **interp expose**.

In addition, the **env** variable is not present in a safe interpreter, so it cannot share environment variables with other interpreters. The **env** variable poses a security risk, because users can store sensitive information in an environment variable. For example, the PGP manual recommends storing the PGP private key protection password in the environment variable *PGPPASS*. Making this variable available to untrusted code executing in a safe interpreter would incur a security risk.

If extensions are loaded into a safe interpreter, they may also restrict their own functionality to eliminate unsafe commands. For a discussion of management of extensions for safety see the manual entries for **Safe−Tcl** and the **load** Tcl command.

ALIAS INVOCATION

The alias mechanism has been carefully designed so that it can be used safely when an untrusted script is executing in a safe slave and the target of the alias is a trusted master. The most important thing in guaranteeing safety is to ensure that information passed from the slave to the master is never evaluated or substituted in the master; if this were to occur, it would enable an evil script in the slave to invoke arbitrary functions in the master, which would compromise security.

When the source for an alias is invoked in the slave interpreter, the usual Tcl substitutions are performed when parsing that command. These substitutions are carried out in the source interpreter just as they would be for any other command invoked in that interpreter. The command procedure for the source command takes its arguments and merges them with the *targetCmd* and *arg*s for the alias to create a new array of arguments. If the words of *srcCmd* were ''*srcCmd arg1 arg2 ... argN*'', the new set of words will be ''*targetCmd arg arg ... arg arg1 arg2 ... argN*'', where *targetCmd* and *arg*s are the values supplied when the alias was created. *TargetCmd* is then used to locate a command procedure in the target interpreter, and that command procedure is invoked with the new set of arguments. An error occurs if there is no command named *targetCmd* in the target interpreter. No additional substitutions are performed on the words: the target command procedure is invoked directly, without going through the normal Tcl evaluation mechanism. Substitutions are thus performed on each word exactly once: *targetCmd* and *args* were substituted when parsing the command that created the alias, and *arg1* - *argN* are substituted when the alias's source command is parsed in the source interpreter.

When writing the *targetCmd*s for aliases in safe interpreters, it is very important that the arguments to that command never be evaluated or substituted, since this would provide an escape mechanism whereby the slave interpreter could execute arbitrary code in the master. This in turn would compromise the security of the system.

HIDDEN COMMANDS

Safe interpreters greatly restrict the functionality available to Tcl programs executing within them. Allowing the untrusted Tcl program to have direct access to this functionality is unsafe, because it can be used for a variety of attacks on the environment. However, there are times when there is a legitimate need to use the dangerous functionality in the context of the safe interpreter. For example, sometimes a program must be **source**d into the interpreter. Another example is Tk, where windows are bound to the hierarchy of windows for a specific interpreter; some potentially dangerous functions, e.g. window management, must be performed on these windows within the interpreter context.

The **interp** command provides a solution to this problem in the form of *hidden commands*. Instead of removing the dangerous commands entirely from a safe interpreter, these commands are hidden so they become unavailable to Tcl scripts executing in the interpreter. However, such hidden commands can be invoked by any trusted ancestor of the safe interpreter, in the context of the safe interpreter, using **interp invoke**. Hidden commands and exposed commands reside in separate name spaces. It is possible to define a hidden command and an exposed command by the same name within one interpreter.

Hidden commands in a slave interpreter can be invoked in the body of procedures called in the master during alias invocation. For example, an alias for **source** could be created in a slave interpreter. When it is invoked in the slave interpreter, a procedure is called in the master interpreter to check that the operation is allowable (e.g. it asks to source a file that the slave interpreter is allowed to access). The procedure then it invokes the hidden **source** command in the slave interpreter to actually source in the contents of the file. Note that two commands named **source** exist in the slave interpreter: the alias, and the hidden command.

Because a master interpreter may invoke a hidden command as part of handling an alias invocation, great care must be taken to avoid evaluating any arguments passed in through the alias invocation. Otherwise, malicious slave interpreters could cause a trusted master interpreter to execute dangerous commands on their behalf. See the section on ALIAS INVOCATION for a more complete discussion of this topic. To help avoid this problem, no substitutions or evaluations are applied to arguments of **interp invokehidden**.

Safe interpreters are not allowed to invoke hidden commands in themselves or in their descendants. This prevents safe slaves from gaining access to hidden functionality in themselves or their descendants.

The set of hidden commands in an interpreter can be manipulated by a trusted interpreter using **interp expose** and **interp hide**. The **interp expose** command moves a hidden command to the set of exposed commands in the interpreter identified by *path*, potentially renaming the command in the process. If an exposed command by the targetted name already exists, the operation fails. Similarly, **interp hide** moves an exposed command to the set of hidden commands in that interpreter. Safe interpreters are not allowed to move commands between the set of hidden and exposed commands, in either themselves or their descendants.

Currently, the names of hidden commands cannot contain namespace qualifiers, and you must first rename a command in a namespace to the global namespace before you can hide it. Commands to be hidden by **interp hide** are looked up in the global namespace even if the current namespace is not the global one. This prevents slaves from fooling a master interpreter into hiding the wrong command, by making the current namespace be different from the global one.

CREDITS

This mechanism is based on the Safe-Tcl prototype implemented by Nathaniel Borenstein and Marshall Rose.

SEE ALSO

load(n), safe(n), Tcl_CreateSlave(3)

KEYWORDS

alias, master interpreter, safe interpreter, slave interpreter

NAME

join − Create a string by joining together list elements

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.

KEYWORDS

element, join, list, separator

NAME

lappend − Append list elements onto a variable

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 *value*s 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.

KEYWORDS

append, element, list, variable

NAME

library − standard library of Tcl procedures

SYNOPSIS

auto_execok *cmd*
auto_load *cmd*
auto_mkindex *dir pattern pattern ...*
auto_mkindex_old *dir pattern pattern ...*
auto_reset
tcl_findLibrary *basename version patch initScript enVarName varName*
parray *arrayName*
tcl_endOfWord *str start*
tcl_startOfNextWord *str start*
tcl_startOfPreviousWord *str start*
tcl_wordBreakAfter *str start*
tcl_wordBreakBefore *str start*

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 [file join [info library] init.tcl]

If the library procedure **Tcl_Init** is invoked from an application's **Tcl_AppInit** procedure, this happens automatically. The code in **init.tcl** 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 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*. If no pattern is given a pattern of ∗**.tcl** will be assumed. 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 by sourcing them into a slave interpreter and monitoring the proc and namespace commands that are executed. Extensions can use the (undocumented) auto_mkindex_parser package to register other commands that can contribute to the auto_load index. You will have to read through init.tcl to see how this works.

Auto_mkindex_old 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're used.

tcl_findLibrary *basename version patch initScript enVarName varName*

This is a standard search procedure for use by extensions during their initialization. They call this procedure to look for their script library in several standard directories. The last component of the name of the library directory is normally *basenameversion* (e.g., tk8.0), but it might be "library" when in the build hierarchies. The *initScript* file will be sourced into the interpreter once it is found. The directory in which this file is found is stored into the global variable *varName*. If this variable is already defined (e.g., by C code during application initialization) then no searching is done. Otherwise the search looks in these directories: the directory named by the environment variable *enVarName*; relative to the Tcl library directory; relative to the executable file in the standard installation bin or bin/*arch* directory; relative to the executable file in the current build tree; relative to the executable file in a parallel build tree.

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.

tcl_endOfWord *str start*

Returns the index of the first end-of-word location that occurs after a starting index *start* in the string *str*. An end-of-word location is defined to be the first non-word character following the first word character after the starting point. Returns -1 if there are no more end-of-word locations after the starting point. See the description of **tcl_wordchars** and **tcl_nonwordchars** below for more details on how Tcl determines which characters are word characters.

tcl_startOfNextWord *str start*

Returns the index of the first start-of-word location that occurs after a starting index *start* in the string *str*. A start-of-word location is defined to be the first word character following a non-word character. Returns −1 if there are no more start-of-word locations after the starting point.

tcl_startOfPreviousWord *str start*
Returns the index of the first start-of-word location that occurs before a starting index *start* in the string *str*. Returns −1 if there are no more start-of-word locations before the starting point.

tcl_wordBreakAfter *str start*
Returns the index of the first word boundary after the starting index *start* in the string *str*. Returns −1 if there are no more boundaries after the starting point in the given string. The index returned refers to the second character of the pair that comprises a boundary.

tcl_wordBreakBefore *str start*
Returns the index of the first word boundary before the starting index *start* in the string *str*. Returns −1 if there are no more boundaries before the starting point in the given string. The index returned refers to the second character of the pair that comprises a boundary.

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, then **unknown** will not attempt to auto-exec any commands.

auto_noload
If set to any value, then **unknown** will not attempt to auto-load any commands.

auto_path
If set, then it must contain a valid Tcl list giving directories to search during auto-load operations.

env(TCL_LIBRARY)
If set, then 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 then a default value is used.

env(TCLLIBPATH)
If set, then 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.

tcl_nonwordchars
This variable contains a regular expression that is used by routines like **tcl_endOfWord** to identify whether a character is part of a word or not. If the pattern matches a character, the character is considered to be a non-word character. On Windows platforms, spaces, tabs, and newlines are considered non-word characters. Under Unix, everything but numbers, letters and underscores are considered non-word characters.

tcl_wordchars
This variable contains a regular expression that is used by routines like **tcl_endOfWord** to identify whether a character is part of a word or not. If the pattern matches a character, the character is considered to be a word character. On Windows platforms, words are comprised of any character that is not a space, tab, or newline. Under Unix, words are comprised of numbers, letters or

underscores.

unknown_active
This variable is set by **unknown** to indicate that it is active. It is used to detect errors where **unknown** recurses on itself infinitely. The variable is unset before **unknown** returns.

KEYWORDS
auto-exec, auto-load, library, unknown, word, whitespace

NAME

lindex − Retrieve an element from a list

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 *index* is negative or greater than or equal to the number of elements in *value*, then an empty string is returned. If *index* has the value **end**, it refers to the last element in the list.

KEYWORDS

element, index, list

NAME

linsert − Insert elements into a list

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 *index*th 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* has the value **end**, or if it is greater than or equal to the number of elements in the list, then the new elements are appended to the list.

KEYWORDS

element, insert, list

NAME

list − Create a list

SYNOPSIS

list ?*arg arg* ...?

DESCRIPTION

This command returns a list comprised of all the *arg*s, or an empty string if no *arg*s are specified. Braces and backslashes get 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 *arg*s 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}

KEYWORDS

element, list

NAME

llength − Count the number of elements in a list

SYNOPSIS

llength *list*

DESCRIPTION

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

KEYWORDS

element, list, length

NAME

load − Load machine code and initialize new commands.

SYNOPSIS

load *fileName*
load *fileName packageName*
load *fileName packageName interp*

DESCRIPTION

This command loads binary code from a file into the application's address space and calls an initialization procedure in the package to incorporate it into an interpreter. *fileName* is the name of the file containing the code; its exact form varies from system to system but on most systems it is a shared library, such as a **.so** file under Solaris or a DLL under Windows. *packageName* is the name of the package, and is used to compute the name of an initialization procedure. *interp* is the path name of the interpreter into which to load the package (see the **interp** manual entry for details); if *interp* is omitted, it defaults to the interpreter in which the **load** command was invoked.

Once the file has been loaded into the application's address space, one of two initialization procedures will be invoked in the new code. Typically the initialization procedure will add new commands to a Tcl interpreter. The name of the initialization procedure is determined by *packageName* and whether or not the target interpreter is a safe one. For normal interpreters the name of the initialization procedure will have the form *pkg*_**Init**, where *pkg* is the same as *packageName* except that the first letter is converted to upper case and all other letters are converted to lower case. For example, if *packageName* is **foo** or **FOo**, the initialization procedure's name will be **Foo_Init**.

If the target interpreter is a safe interpreter, then the name of the initialization procedure will be *pkg*_**SafeInit** instead of *pkg*_**Init**. The *pkg*_**SafeInit** function should be written carefully, so that it initializes the safe interpreter only with partial functionality provided by the package that is safe for use by untrusted code. For more information on Safe−Tcl, see the **safe** manual entry.

The initialization procedure must match the following prototype:

typedef int Tcl_PackageInitProc(Tcl_Interp ∗*interp*);

The *interp* argument identifies the interpreter in which the package is to be loaded. The initialization procedure must return **TCL_OK** or **TCL_ERROR** to indicate whether or not it completed successfully; in the event of an error it should set *interp*->*result* to point to an error message. The result of the **load** command will be the result returned by the initialization procedure.

The actual loading of a file will only be done once for each *fileName* in an application. If a given *fileName* is loaded into multiple interpreters, then the first **load** will load the code and call the initialization procedure; subsequent **load**s will call the initialization procedure without loading the code again. It is not possible to unload or reload a package.

The **load** command also supports packages that are statically linked with the application, if those packages have been registered by calling the **Tcl_StaticPackage** procedure. If *fileName* is an empty string, then *packageName* must be specified.

If *packageName* is omitted or specified as an empty string, Tcl tries to guess the name of the package. This may be done differently on different platforms. The default guess, which is used on most UNIX platforms, is to take the last element of *fileName*, strip off the first three characters if they are **lib**, and use any following alphabetic and underline characters as the module name. For example, the command **load libxyz4.2.so** uses the module name **xyz** and the command **load bin/last.so {}** uses the module name **last**.

If *fileName* is an empty string, then *packageName* must be specified. The **load** command first searches for a statically loaded package (one that has been registered by calling the **Tcl_StaticPackage** procedure) by that name; if one is found, it is used. Otherwise, the **load** command searches for a dynamically loaded

package by that name, and uses it if it is found. If several different files have been **load**ed with different versions of the package, Tcl picks the file that was loaded first.

BUGS

If the same file is **load**ed by different *fileName*s, it will be loaded into the process's address space multiple times. The behavior of this varies from system to system (some systems may detect the redundant loads, others may not).

SEE ALSO

info sharedlibextension, Tcl_StaticPackage, safe(n)

KEYWORDS

binary code, loading, safe interpreter, shared library

NAME

lrange − Return one or more adjacent elements from a list

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. *First* or *last* may be **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***]**"

KEYWORDS

element, list, range, sublist

NAME

lreplace − Replace elements in a list with new elements

SYNOPSIS

lreplace *list first last* ?*element 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* of the first element to be replaced (0 refers to the first element). 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. If *last* is less than *first* then no elements are deleted; the new elements are simply inserted before *first*. *First* or *last* may be **end** (or any abbreviation of it) to refer to the last element of the list. 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.

KEYWORDS

element, list, replace

NAME

lsearch − See if a list contains a particular element

SYNOPSIS

lsearch ?*mode*? *list pattern*

DESCRIPTION

This command searches the elements of *list* to see if one of them matches *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 *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**.

KEYWORDS

list, match, pattern, regular expression, search, string

NAME

lsort − Sort the elements of a list

SYNOPSIS

lsort ?*options*? *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 options 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.

−dictionary Use dictionary-style comparison. This is the same as **−ascii** except (a) case is ignored except as a tie-breaker and (b) if two strings contain embedded numbers, the numbers compare as integers, not characters. For example, in **−dictionary** mode, **bigBoy** sorts between **bigbang** and **bigboy**, and **x10y** sorts between **x9y** and **x11y**.

−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).

−index *index* If this option is specified, each of the elements of *list* must itself be a proper Tcl sublist. Instead of sorting based on whole sublists, **lsort** will extract the *index*’th element from each sublist and sort based on the given element. The keyword **end** is allowed for the *index* to sort on the last sublist element. For example,

```
lsort -integer -index 1 {{First 24} {Second 18} {Third 30}}
```

returns **{Second 18} {First 24} {Third 30}**. This option is much more efficient than using **−command** to achieve the same effect.

KEYWORDS

element, list, order, sort

NAME

namespace − create and manipulate contexts for commands and variables

SYNOPSIS

namespace ?*option*? ?*arg ...*?

DESCRIPTION

The **namespace** command lets you create, access, and destroy separate contexts for commands and variables. See the section **WHAT IS A NAMESPACE?** below for a brief overview of namespaces. The legal *option*'s are listed below. Note that you can abbreviate the *option*'s.

namespace children ?*namespace*? ?*pattern*?

Returns a list of all child namespaces that belong to the namespace *namespace*. If *namespace* is not specified, then the children are returned for the current namespace. This command returns fully-qualified names, which start with **::**. If the optional *pattern* is given, then this command returns only the names that match the glob-style pattern. The actual pattern used is determined as follows: a pattern that starts with **::** is used directly, otherwise the namespace *namespace* (or the fully-qualified name of the current namespace) is prepended onto the the pattern.

namespace code *script*

Captures the current namespace context for later execution of the script *script*. It returns a new script in which *script* has been wrapped in a **namespace code** command. The new script has two important properties. First, it can be evaluated in any namespace and will cause *script* to be evaluated in the current namespace (the one where the **namespace code** command was invoked). Second, additional arguments can be appended to the resulting script and they will be passed to *script* as additional arguments. For example, suppose the command **set script [namespace code {foo bar}]** is invoked in namespace **::a::b**. Then **eval "$script x y"** can be executed in any namespace (assuming the value of **script** has been passed in properly) and will have the same effect as the command **namespace eval ::a::b {foo bar x y}**. This command is needed because extensions like Tk normally execute callback scripts in the global namespace. A scoped command captures a command together with its namespace context in a way that allows it to be executed properly later. See the section **SCOPED VALUES** for some examples of how this is used to create callback scripts.

namespace current

Returns the fully-qualified name for the current namespace. The actual name of the global namespace is "" (i.e., an empty string), but this command returns **::** for the global namespace as a convenience to programmers.

namespace delete ?*namespace namespace ...*?

Each namespace *namespace* is deleted and all variables, procedures, and child namespaces contained in the namespace are deleted. If a procedure is currently executing inside the namespace, the namespace will be kept alive until the procedure returns; however, the namespace is marked to prevent other code from looking it up by name. If a namespace doesn't exist, this command returns an error. If no namespace names are given, this command does nothing.

namespace eval *namespace arg* ?*arg ...*?

Activates a namespace called *namespace* and evaluates some code in that context. If the namespace does not already exist, it is created. If more than one *arg* argument is specified, the arguments are concatenated together with a space between each one in the same fashion as the **eval** command, and the result is evaluated.

If *namespace* has leading namespace qualifiers and any leading namespaces do not exist, they are automatically created.

namespace export ?−**clear**? ?*pattern pattern ...*?
Specifies which commands are exported from a namespace. The exported commands are those that can be later imported into another namespace using a **namespace import** command. Both commands defined in a namespace and commands the namespace has previously imported can be exported by a namespace. The commands do not have to be defined at the time the **namespace export** command is executed. Each *pattern* may contain glob-style special characters, but it may not include any namespace qualifiers. That is, the pattern can only specify commands in the current (exporting) namespace. Each *pattern* is appended onto the namespace's list of export patterns. If the −**clear** flag is given, the namespace's export pattern list is reset to empty before any *pattern* arguments are appended. If no *pattern*s are given and the −**clear** flag isn't given, this command returns the namespace's current export list.

namespace forget ?*pattern pattern ...*?
Removes previously imported commands from a namespace. Each *pattern* is a qualified name such as **foo::x** or **a::b::p***. Qualified names contain **::**s and qualify a name with the name of one or more namespaces. Each *pattern* is qualified with the name of an exporting namespace and may have glob-style special characters in the command name at the end of the qualified name. Glob characters may not appear in a namespace name. This command first finds the matching exported commands. It then checks whether any of those those commands were previously imported by the current namespace. If so, this command deletes the corresponding imported commands. In effect, this un-does the action of a **namespace import** command.

namespace import ?−**force**? ?*pattern pattern ...*?
Imports commands into a namespace. Each *pattern* is a qualified name like **foo::x** or **a::p***. That is, it includes the name of an exporting namespace and may have glob-style special characters in the command name at the end of the qualified name. Glob characters may not appear in a namespace name. All the commands that match a *pattern* string and which are currently exported from their namespace are added to the current namespace. This is done by creating a new command in the current namespace that points to the exported command in its original namespace; when the new imported command is called, it invokes the exported command. This command normally returns an error if an imported command conflicts with an existing command. However, if the −**force** option is given, imported commands will silently replace existing commands. The **namespace import** command has snapshot semantics: that is, only requested commands that are currently defined in the exporting namespace are imported. In other words, you can import only the commands that are in a namespace at the time when the **namespace import** command is executed. If another command is defined and exported in this namespace later on, it will not be imported.

namespace inscope *namespace arg* ?*arg ...*?
Executes a script in the context of a particular namespace. This command is not expected to be used directly by programmers; calls to it are generated implicitly when applications use **namespace code** commands to create callback scripts that the applications then register with, e.g., Tk widgets. The **namespace inscope** command is much like the **namespace eval** command except that it has **lappend** semantics and the namespace must already exist. It treats the first argument as a list, and appends any arguments after the first onto the end as proper list elements. **namespace inscope ::foo a x y z** is equivalent to **namespace eval ::foo [concat a [list x y z]]** This **lappend** semantics is important because many callback scripts are actually prefixes.

namespace origin *command*
Returns the fully-qualified name of the original command to which the imported command *command* refers. When a command is imported into a namespace, a new command is created in that namespace that points to the actual command in the exporting namespace. If a command is imported into a sequence of namespaces *a, b,...,n* where each successive namespace just imports the command from the previous namespace, this command returns the fully-qualified name of the original command in the first namespace, *a*. If *command* does not refer to an imported command,

the command's own fully-qualified name is returned.

namespace parent ?*namespace*?
Returns the fully-qualified name of the parent namespace for namespace *namespace*. If *namespace* is not specified, the fully-qualified name of the current namespace's parent is returned.

namespace qualifiers *string*
Returns any leading namespace qualifiers for *string*. Qualifiers are namespace names separated by **::**s. For the *string* **::foo::bar::x**, this command returns **::foo::bar**, and for **::** it returns **“”** (an empty string). This command is the complement of the **namespace tail** command. Note that it does not check whether the namespace names are, in fact, the names of currently defined namespaces.

namespace tail *string*
Returns the simple name at the end of a qualified string. Qualifiers are namespace names separated by **::**s. For the *string* **::foo::bar::x**, this command returns **x**, and for **::** it returns **“”** (an empty string). This command is the complement of the **namespace qualifiers** command. It does not check whether the namespace names are, in fact, the names of currently defined namespaces.

namespace which ?−**command**? ?−**variable**? *name*
Looks up *name* as either a command or variable and returns its fully-qualified name. For example, if *name* does not exist in the current namespace but does exist in the global namespace, this command returns a fully-qualified name in the global namespace. If the command or variable does not exist, this command returns an empty string. If no flag is given, *name* is treated as a command name. See the section **NAME RESOLUTION** below for an explanation of the rules regarding name resolution.

WHAT IS A NAMESPACE?

A namespace is a collection of commands and variables. It encapsulates the commands and variables to ensure that they won't interfere with the commands and variables of other namespaces. Tcl has always had one such collection, which we refer to as the *global namespace*. The global namespace holds all global variables and commands. The **namespace eval** command lets you create new namespaces. For example,

```
namespace eval Counter {
  namespace export Bump
  variable num 0

  proc Bump {} {
    variable num
    incr num
  }
}
```

creates a new namespace containing the variable **num** and the procedure **Bump**. The commands and variables in this namespace are separate from other commands and variables in the same program. If there is a command named **Bump** in the global namespace, for example, it will be different from the command **Bump** in the **Counter** namespace.

Namespace variables resemble global variables in Tcl. They exist outside of the procedures in a namespace but can be accessed in a procedure via the **variable** command, as shown in the example above.

Namespaces are dynamic. You can add and delete commands and variables at any time, so you can build up the contents of a namespace over time using a series of **namespace eval** commands. For example, the following series of commands has the same effect as the namespace definition shown above:

```
namespace eval Counter {
  variable num 0
  proc Bump {} {
```

```
        variable num
        return [incr num]
    }
}
namespace eval Counter {
    proc test {args} {
        return $args
    }
}
namespace eval Counter {
    rename test ""
}
```

Note that the **test** procedure is added to the **Counter** namespace, and later removed via the **rename** command.

Namespaces can have other namespaces within them, so they nest hierarchically. A nested namespace is encapsulated inside its parent namespace and can not interfere with other namespaces.

QUALIFIED NAMES

Each namespace has a textual name such as **history** or **::safe::interp**. Since namespaces may nest, qualified names are used to refer to commands, variables, and child namespaces contained inside namespaces. Qualified names are similar to the hierarchical path names for Unix files or Tk widgets, except that **::** is used as the separator instead of / or **.**. The topmost or global namespace has the name "" (i.e., an empty string), although **::** is a synonym. As an example, the name **::safe::interp::create** refers to the command **create** in the namespace **interp** that is a child of of namespace **::safe**, which in turn is a child of the global namespace **::**.

If you want to access commands and variables from another namespace, you must use some extra syntax. Names must be qualified by the namespace that contains them. From the global namespace, we might access the **Counter** procedures like this:

```
Counter::Bump 5
Counter::Reset
```

We could access the current count like this:

```
puts "count = $Counter::num"
```

When one namespace contains another, you may need more than one qualifier to reach its elements. If we had a namespace **Foo** that contained the namespace **Counter**, you could invoke its **Bump** procedure from the global namespace like this:

```
Foo::Counter::Bump 3
```

You can also use qualified names when you create and rename commands. For example, you could add a procedure to the **Foo** namespace like this:

```
proc Foo::Test {args} {return $args}
```

And you could move the same procedure to another namespace like this:

```
rename Foo::Test Bar::Test
```

There are a few remaining points about qualified names that we should cover. Namespaces have nonempty names except for the global namespace. **::** is disallowed in simple command, variable, and namespace names except as a namespace separator. Extra **:**s in a qualified name are ignored; that is, two or more **:**s are treated as a namespace separator. A trailing **::** in a qualified variable or command name refers to the variable or command named {}. However, a trailing **::** in a qualified namespace name is ignored.

NAME RESOLUTION

In general, all Tcl commands that take variable and command names support qualified names. This means you can give qualified names to such commands as **set**, **proc**, **rename**, and **interp alias**. If you provide a fully-qualified name that starts with a **::**, there is no question about what command, variable, or namespace you mean. However, if the name does not start with a **::** (i.e., is *relative*), Tcl follows a fixed rule for looking it up: Command and variable names are always resolved by looking first in the current namespace, and then in the global namespace. Namespace names, on the other hand, are always resolved by looking in only the current namespace.

In the following example,

```
set traceLevel 0
namespace eval Debug {
  printTrace $traceLevel
}
```

Tcl looks for **traceLevel** in the namespace **Debug** and then in the global namespace. It looks up the command **printTrace** in the same way. If a variable or command name is not found in either context, the name is undefined. To make this point absolutely clear, consider the following example:

```
set traceLevel 0
namespace eval Foo {
  variable traceLevel 3

  namespace eval Debug {
    printTrace $traceLevel
  }
}
```

Here Tcl looks for **traceLevel** first in the namespace **Foo::Debug**. Since it is not found there, Tcl then looks for it in the global namespace. The variable **Foo::traceLevel** is completely ignored during the name resolution process.

You can use the **namespace which** command to clear up any question about name resolution. For example, the command:

```
namespace eval Foo::Debug {namespace which −variable traceLevel}
```

returns **::traceLevel**. On the other hand, the command,

```
namespace eval Foo {namespace which −variable traceLevel}
```

returns **::Foo::traceLevel**.

As mentioned above, namespace names are looked up differently than the names of variables and commands. Namespace names are always resolved in the current namespace. This means, for example, that a **namespace eval** command that creates a new namespace always creates a child of the current namespace unless the new namespace name begins with a **::**.

Tcl has no access control to limit what variables, commands, or namespaces you can reference. If you provide a qualified name that resolves to an element by the name resolution rule above, you can access the element.

You can access a namespace variable from a procedure in the same namespace by using the **variable** command. Much like the **global** command, this creates a local link to the namespace variable. If necessary, it also creates the variable in the current namespace and initializes it. Note that the **global** command only creates links to variables in the global namespace. It is not necessary to use a **variable** command if you always refer to the namespace variable using an appropriate qualified name.

IMPORTING COMMANDS

Namespaces are often used to represent libraries. Some library commands are used so frequently that it is a nuisance to type their qualified names. For example, suppose that all of the commands in a package like BLT are contained in a namespace called **Blt**. Then you might access these commands like this:

Blt::graph .g −background red
Blt::table . .g 0,0

If you use the **graph** and **table** commands frequently, you may want to access them without the **Blt::** prefix. You can do this by importing the commands into the current namespace, like this:

namespace import Blt::*

This adds all exported commands from the **Blt** namespace into the current namespace context, so you can write code like this:

graph .g −background red
table . .g 0,0

The **namespace import** command only imports commands from a namespace that that namespace exported with a **namespace export** command.

Importing *every* command from a namespace is generally a bad idea since you don't know what you will get. It is better to import just the specific commands you need. For example, the command

namespace import Blt::graph Blt::table

imports only the **graph** and **table** commands into the current context.

If you try to import a command that already exists, you will get an error. This prevents you from importing the same command from two different packages. But from time to time (perhaps when debugging), you may want to get around this restriction. You may want to reissue the **namespace import** command to pick up new commands that have appeared in a namespace. In that case, you can use the **−force** option, and existing commands will be silently overwritten:

namespace import −force Blt::graph Blt::table

If for some reason, you want to stop using the imported commands, you can remove them with an **namespace forget** command, like this:

namespace forget Blt::*

This searches the current namespace for any commands imported from **Blt**. If it finds any, it removes them. Otherwise, it does nothing. After this, the **Blt** commands must be accessed with the **Blt::** prefix.

When you delete a command from the exporting namespace like this:

rename Blt::graph ""

the command is automatically removed from all namespaces that import it.

EXPORTING COMMANDS

You can export commands from a namespace like this:

```
namespace eval Counter {
  namespace export Bump Reset
  variable num 0
  variable max 100

  proc Bump {{by 1}} {
    variable num
    incr num $by
    check
    return $num
  }
  proc Reset {} {
    variable num
    set num 0
  }
```

```
    proc check {} {
        variable num
        variable max
        if {$num > $max} {
            error "too high!"
        }
    }
}
```

The procedures **Bump** and **Reset** are exported, so they are included when you import from the **Counter** namespace, like this:

namespace import Counter::*

However, the **check** procedure is not exported, so it is ignored by the import operation.

The **namespace import** command only imports commands that were declared as exported by their namespace. The **namespace export** command specifies what commands may be imported by other namespaces. If a **namespace import** command specifies a command that is not exported, the command is not imported.

SEE ALSO

variable(n)

KEYWORDS

exported, internal, variable

NAME

open − Open a file-based or command pipeline channel

SYNOPSIS

open *fileName*
open *fileName access*
open *fileName access permissions*

DESCRIPTION

This command opens a file, serial port, or command pipeline and returns a channel identifier that may be used in future invocations of commands like **read**, **puts**, and **close**. If the first character of *fileName* is not **|** then the command opens a file: *fileName* gives the name of the file to open, and it must conform to the conventions described in the **filename** manual entry.

The *access* argument, if present, indicates the way in which the file (or command pipeline) is to be accessed. In the first form *access* may have any of the following values:

r Open the file for reading only; the file must already exist. This is the default value if *access* is not specified.

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 exist (without this flag it is an error for the file not to exist).

EXCL If **CREAT** is also specified, 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, and possibly in subsequent I/O operations. The exact behavior of this flag is system- and device-dependent; its use is discouraged (it is better to use the **fconfigure** command to put a file in nonblocking mode). For details refer to your 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.

COMMAND PIPELINES

If the first character of *fileName* is ''|'' then the remaining characters of *fileName* are treated as a list of arguments that describe a command pipeline to invoke, in the same style as the arguments for **exec**. In this case, the channel identifier returned by **open** may be used to write to the command's input pipe or read from its output pipe, depending on the value of *access*. If write-only access is used (e.g. *access* is **w**), then standard output for the pipeline is directed to the current standard output unless overridden by the command. If read-only access is used (e.g. *access* is **r**), standard input for the pipeline is taken from the current standard input unless overridden by the command.

SERIAL COMMUNICATIONS

If *fileName* refers to a serial port, then the specified serial port is opened and initialized in a platform-dependent manner. Acceptable values for the *fileName* to use to open a serial port are described in the PORTABILITY ISSUES section.

CONFIGURATION OPTIONS

The **fconfigure** command can be used to query and set the following configuration option for open serial ports:

−mode *baud,parity,data,stop*

This option is a set of 4 comma-separated values: the baud rate, parity, number of data bits, and number of stop bits for this serial port. The *baud* rate is a simple integer that specifies the connection speed. *Parity* is one of the following letters: **n**, **o**, **e**, **m**, **s**; respectively signifying the parity options of ''none'', ''odd'', ''even'', ''mark'', or ''space''. *Data* is the number of data bits and should be an integer from 5 to 8, while *stop* is the number of stop bits and should be the integer 1 or 2.

PORTABILITY ISSUES

Windows (all versions)

Valid values for *fileName* to open a serial port are of the form **com***X***:**, where *X* is a number, generally from 1 to 4. An attempt to open a serial port that does not exist will fail.

Windows NT

When running Tcl interactively, there may be some strange interactions between the real console, if one is present, and a command pipeline that uses standard input or output. If a command pipeline is opened for reading, some of the lines entered at the console will be sent to the command pipeline and some will be sent to the Tcl evaluator. If a command pipeline is opened for writing, keystrokes entered into the console are not visible until the the pipe is closed. This behavior occurs whether the command pipeline is executing 16-bit or 32-bit applications. These problems only occur because both Tcl and the child application are competing for the console at the same time. If the command pipeline is started from a script, so that Tcl is not accessing the console, or if the command pipeline does not use standard input or output, but is redirected from or to a file, then the above problems do not occur.

Windows 95

A command pipeline that executes a 16-bit DOS application cannot be opened for both reading and writing, since 16-bit DOS applications that receive standard input from a pipe and send standard output to a pipe run synchronously. Command pipelines that do not execute 16-bit DOS applications run asynchronously and can be opened for both reading and writing.

When running Tcl interactively, there may be some strange interactions between the real console, if one is present, and a command pipeline that uses standard input or output. If a command

pipeline is opened for reading from a 32-bit application, some of the keystrokes entered at the console will be sent to the command pipeline and some will be sent to the Tcl evaluator. If a command pipeline is opened for writing to a 32-bit application, no output is visible on the console until the the pipe is closed. These problems only occur because both Tcl and the child application are competing for the console at the same time. If the command pipeline is started from a script, so that Tcl is not accessing the console, or if the command pipeline does not use standard input or output, but is redirected from or to a file, then the above problems do not occur.

Whether or not Tcl is running interactively, if a command pipeline is opened for reading from a 16-bit DOS application, the call to **open** will not return until end-of-file has been received from the command pipeline's standard output. If a command pipeline is opened for writing to a 16-bit DOS application, no data will be sent to the command pipeline's standard output until the pipe is actually closed. This problem occurs because 16-bit DOS applications are run synchronously, as described above.

Windows 3.X

A command pipeline can execute 16-bit or 32-bit DOS or Windows applications, but the call to **open** will not return until the last program in the pipeline has finished executing; command pipelines run synchronously. If the pipeline is opened with write access (either just writing or both reading and writing) the first application in the pipeline will instead see an immediate end-of-file; any data the caller writes to the open pipe will instead be discarded.

Since Tcl cannot be run with a real console under Windows 3.X, there are no interactions between command pipelines and the console.

Macintosh

Opening a serial port is not currently implemented under Macintosh.

Opening a command pipeline is not supported under Macintosh, since applications do not support the concept of standard input or output.

Unix

Valid values for *fileName* to open a serial port are generally of the form **/dev/tty***X*, where *X* is **a** or **b**, but the name of any pseudo-file that maps to a serial port may be used.

When running Tcl interactively, there may be some strange interactions between the console, if one is present, and a command pipeline that uses standard input. If a command pipeline is opened for reading, some of the lines entered at the console will be sent to the command pipeline and some will be sent to the Tcl evaluator. This problem only occurs because both Tcl and the child application are competing for the console at the same time. If the command pipeline is started from a script, so that Tcl is not accessing the console, or if the command pipeline does not use standard input, but is redirected from a file, then the above problem does not occur.

See the PORTABILITY ISSUES section of the **exec** command for additional information not specific to command pipelines about executing applications on the various platforms

SEE ALSO

close(n), filename(n), gets(n), read(n), puts(n), exec(n)

KEYWORDS

access mode, append, create, file, non-blocking, open, permissions, pipeline, process, serial

NAME

package − Facilities for package loading and version control

SYNOPSIS

package forget *package*
package ifneeded *package version* ?*script*?
package names
package provide *package* ?*version*?
package require ?−**exact**? *package* ?*version*?
package unknown ?*command*?
package vcompare *version1 version2*
package versions *package*
package vsatisfies *version1 version2*

DESCRIPTION

This command keeps a simple database of the packages available for use by the current interpreter and how to load them into the interpreter. It supports multiple versions of each package and arranges for the correct version of a package to be loaded based on what is needed by the application. This command also detects and reports version clashes. Typically, only the **package require** and **package provide** commands are invoked in normal Tcl scripts; the other commands are used primarily by system scripts that maintain the package database.

The behavior of the **package** command is determined by its first argument. The following forms are permitted:

package forget *package*

Removes all information about *package* from this interpreter, including information provided by both **package ifneeded** and **package provide**.

package ifneeded *package version* ?*script*?

This command typically appears only in system configuration scripts to set up the package database. It indicates that a particular version of a particular package is available if needed, and that the package can be added to the interpreter by executing *script*. The script is saved in a database for use by subsequent **package require** commands; typically, *script* sets up auto-loading for the commands in the package (or calls **load** and/or **source** directly), then invokes **package provide** to indicate that the package is present. There may be information in the database for several different versions of a single package. If the database already contains information for *package* and *version*, the new *script* replaces the existing one. If the *script* argument is omitted, the current script for version *version* of package *package* is returned, or an empty string if no **package ifneeded** command has been invoked for this *package* and *version*.

package names

Returns a list of the names of all packages in the interpreter for which a version has been provided (via **package provide**) or for which a **package ifneeded** script is available. The order of elements in the list is arbitrary.

package provide *package* ?*version*?

This command is invoked to indicate that version *version* of package *package* is now present in the interpreter. It is typically invoked once as part of an **ifneeded** script, and again by the package itself when it is finally loaded. An error occurs if a different version of *package* has been provided by a previous **package provide** command. If the *version* argument is omitted, then the command returns the version number that is currently provided, or an empty string if no **package provide** command has been invoked for *package* in this interpreter.

package require ?**−exact**? *package* ?*version*?
This command is typically invoked by Tcl code that wishes to use a particular version of a particular package. The arguments indicate which package is wanted, and the command ensures that a suitable version of the package is loaded into the interpreter. If the command succeeds, it returns the version number that is loaded; otherwise it generates an error. If both the **−exact** switch and the *version* argument are specified then only the given version is acceptable. If **−exact** is omitted but *version* is specified, then versions later than *version* are also acceptable as long as they have the same major version number as *version*. If both **−exact** and *version* are omitted then any version whatsoever is acceptable. If a version of *package* has already been provided (by invoking the **package provide** command), then its version number must satisfy the criteria given by **−exact** and *version* and the command returns immediately. Otherwise, the command searches the database of information provided by previous **package ifneeded** commands to see if an acceptable version of the package is available. If so, the script for the highest acceptable version number is invoked; it must do whatever is necessary to load the package, including calling **package provide** for the package. If the **package ifneeded** database does not contain an acceptable version of the package and a **package unknown** command has been specified for the interpreter then that command is invoked; when it completes, Tcl checks again to see if the package is now provided or if there is a **package ifneeded** script for it. If all of these steps fail to provide an acceptable version of the package, then the command returns an error.

package unknown ?*command*?
This command supplies a ‘‘last resort’’ command to invoke during **package require** if no suitable version of a package can be found in the **package ifneeded** database. If the *command* argument is supplied, it contains the first part of a command; when the command is invoked during a **package require** command, Tcl appends two additional arguments giving the desired package name and version. For example, if *command* is **foo bar** and later the command **package require test 2.4** is invoked, then Tcl will execute the command **foo bar test 2.4** to load the package. If no version number is supplied to the **package require** command, then the version argument for the invoked command will be an empty string. If the **package unknown** command is invoked without a *command* argument, then the current **package unknown** script is returned, or an empty string if there is none. If *command* is specified as an empty string, then the current **package unknown** script is removed, if there is one.

package vcompare *version1 version2*
Compares the two version numbers given by *version1* and *version2*. Returns -1 if *version1* is an earlier version than *version2*, 0 if they are equal, and 1 if *version1* is later than **version2**.

package versions *package*
Returns a list of all the version numbers of *package* for which information has been provided by **package ifneeded** commands.

package vsatisfies *version1 version2*
Returns 1 if scripts written for *version2* will work unchanged with *version1* (i.e. *version1* is equal to or greater than *version2* and they both have the same major version number), 0 otherwise.

VERSION NUMBERS
Version numbers consist of one or more decimal numbers separated by dots, such as 2 or 1.162 or 3.1.13.1. The first number is called the major version number. Larger numbers correspond to later versions of a package, with leftmost numbers having greater significance. For example, version 2.1 is later than 1.3 and version 3.4.6 is later than 3.3.5. Missing fields are equivalent to zeroes: version 1.3 is the same as version 1.3.0 and 1.3.0.0, so it is earlier than 1.3.1 or 1.3.0.2. A later version number is assumed to be upwards compatible with an earlier version number as long as both versions have the same major version number. For example, Tcl scripts written for version 2.3 of a package should work unchanged under versions 2.3.2, 2.4, and 2.5.1. Changes in the major version number signify incompatible changes: if code is written to use

version 2.1 of a package, it is not guaranteed to work unmodified with either version 1.7.3 or version 3.1.

PACKAGE INDICES

The recommended way to use packages in Tcl is to invoke **package require** and **package provide** commands in scripts, and use the procedure **pkg_mkIndex** to create package index files. Once you've done this, packages will be loaded automatically in response to **package require** commands. See the documentation for **pkg_mkIndex** for details.

KEYWORDS

package, version

NAME

pid − Retrieve process id(s)

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.

KEYWORDS

file, pipeline, process identifier

NAME

pkg_mkIndex − Build an index for automatic loading of packages

SYNOPSIS

pkg_mkIndex *dir pattern* ?*pattern pattern ...*?

DESCRIPTION

Pkg_mkIndex is a utility procedure that is part of the standard Tcl library. It is used to create index files that allow packages to be loaded automatically when **package require** commands are executed. To use **pkg_mkIndex**, follow these steps:

[1] Create the package(s). Each package may consist of one or more Tcl script files or binary files. Binary files must be suitable for loading with the **load** command with a single argument; for example, if the file is **test.so** it must be possible to load this file with the command **load test.so**. Each script file must contain a **package provide** command to declare the package and version number, and each binary file must contain a call to **Tcl_PkgProvide**.

[2] Create the index by invoking **pkg_mkIndex**. The *dir* argument gives the name of a directory and each *pattern* argument is a **glob**-style pattern that selects script or binary files in *dir*. **Pkg_mkIndex** will create a file **pkgIndex.tcl** in *dir* with package information about all the files given by the *pattern* arguments. It does this by loading each file and seeing what packages and new commands appear (this is why it is essential to have **package provide** commands or **Tcl_PkgProvide** calls in the files, as described above).

[3] Install the package as a subdirectory of one of the directories given by the **tcl_pkgPath** variable. If **$tcl_pkgPath** contains more than one directory, machine-dependent packages (e.g., those that contain binary shared libraries) should normally be installed under the first directory and machine-independent packages (e.g., those that contain only Tcl scripts) should be installed under the second directory. The subdirectory should include the package's script and/or binary files as well as the **pkgIndex.tcl** file. As long as the package is installed as a subdirectory of a directory in **$tcl_pkgPath** it will automatically be found during **package require** commands.

If you install the package anywhere else, then you must ensure that the directory contaiingn the package is in the **auto_path** global variable or an immediate subdirectory of one of the directories in **auto_path**. **Auto_path** contains a list of directories that are searched by both the auto-loader and the package loader; by default it includes **$tcl_pkgPath**. The package loader also checks all of the subdirectories of the directories in **auto_path**. You can add a directory to **auto_path** explicitly in your application, or you can add the directory to your **TCLLIBPATH** environment variable: if this environment variable is present, Tcl initializes **auto_path** from it during application startup.

[4] Once the above steps have been taken, all you need to do to use a package is to invoke **package require**. For example, if versions 2.1, 2.3, and 3.1 of package **Test** have been indexed by **pkg_mkIndex**, the command **package require Test** will make version 3.1 available and the command **package require −exact Test 2.1** will make version 2.1 available. There may be many versions of a package in the various index files in **auto_path**, but only one will actually be loaded in a given interpreter, based on the first call to **package require**. Different versions of a package may be loaded in different interpreters.

PACKAGES AND THE AUTO-LOADER

The package management facilities overlap somewhat with the auto-loader, in that both arrange for files to be loaded on-demand. However, package management is a higher-level mechanism that uses the auto-loader for the last step in the loading process. It is generally better to index a package with **pkg_mkIndex**

rather than **auto_mkindex** because the package mechanism provides version control: several versions of a package can be made available in the index files, with different applications using different versions based on **package require** commands. In contrast, **auto_mkindex** does not understand versions so it can only handle a single version of each package. It is probably not a good idea to index a given package with both **pkg_mkIndex** and **auto_mkindex**. If you use **pkg_mkIndex** to index a package, its commands cannot be invoked until **package require** has been used to select a version; in contrast, packages indexed with **auto_mkindex** can be used immediately since there is no version control.

HOW IT WORKS

Pkg_mkIndex depends on the **package unknown** command, the **package ifneeded** command, and the auto-loader. The first time a **package require** command is invoked, the **package unknown** script is invoked. This is set by Tcl initialization to a script that evaluates all of the **pkgIndex.tcl** files in the **auto_path**. The **pkgIndex.tcl** files contain **package ifneeded** commands for each version of each available package; these commands invoke **package provide** commands to announce the availability of the package, and they setup auto-loader information to load the files of the package. A given file of a given version of a given package isn't actually loaded until the first time one of its commands is invoked. Thus, after invoking **package require** you won't see the package's commands in the interpreter, but you will be able to invoke the commands and they will be auto-loaded.

KEYWORDS

auto-load, index, package, version

NAME

proc − Create a Tcl procedure

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. Normally, *name* is unqualified (does not include the names of any containing namespaces), and the new procedure is created in the current namespace. If *name* includes any namespace qualifiers, the procedure is created in the specified namespace. *Args* specifies the formal arguments to the procedure. It consists of a list, possibly empty, each of whose elements specifies one argument. Each argument specifier is also a list with either one or two fields. If there is only a single field in the specifier 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. Namespace variables can only be accessed by invoking the **variable** 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.

KEYWORDS

argument, procedure

NAME

puts − Write to a channel

SYNOPSIS

puts ?−**nonewline**? ?*channelId*? *string*

DESCRIPTION

Writes the characters given by *string* to the channel given by *channelId*. *ChannelId* must be a channel identifier such as returned from a previous invocation of **open** or **socket**. It must have been opened for output. If no *channelId* 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.

Newline characters in the output are translated by **puts** to platform-specific end-of-line sequences according to the current value of the −**translation** option for the channel (for example, on PCs newlines are normally replaced with carriage-return-linefeed sequences; on Macintoshes newlines are normally replaced with carriage-returns). See the **fconfigure** manual entry for a discussion of end-of-line translations.

Tcl buffers output internally, so characters written with **puts** may not appear immediately on the output file or device; Tcl will normally delay output until the buffer is full or the channel is closed. You can force output to appear immediately with the **flush** command.

When the output buffer fills up, the **puts** command will normally block until all the buffered data has been accepted for output by the operating system. If *channelId* is in nonblocking mode then the **puts** command will not block even if the operating system cannot accept the data. Instead, Tcl continues to buffer the data and writes it in the background as fast as the underlying file or device can accept it. The application must use the Tcl event loop for nonblocking output to work; otherwise Tcl never finds out that the file or device is ready for more output data. It is possible for an arbitrarily large amount of data to be buffered for a channel in nonblocking mode, which could consume a large amount of memory. To avoid wasting memory, nonblocking I/O should normally be used in an event-driven fashion with the **fileevent** command (don't invoke **puts** unless you have recently been notified via a file event that the channel is ready for more output data).

SEE ALSO

fileevent(n)

KEYWORDS

channel, newline, output, write

NAME

pwd − Return the current working directory

SYNOPSIS

pwd

DESCRIPTION

Returns the path name of the current working directory.

KEYWORDS

working directory

NAME

read − Read from a channel

SYNOPSIS

read ?**−nonewline**? *channelId*

read *channelId numBytes*

DESCRIPTION

In the first form, the **read** command reads all of the data from *channelId* up to the end of the file. 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* left in the file; in this case all the remaining bytes are returned.

If *channelId* is in nonblocking mode, the command may not read as many bytes as requested: once all available input has been read, the command will return the data that is available rather than blocking for more input. The **−nonewline** switch is ignored if the command returns before reaching the end of the file.

Read translates end-of-line sequences in the input into newline characters according to the **−translation** option for the channel. See the manual entry for **fconfigure** for details on the **−translation** option.

SEE ALSO

eof(n), fblocked(n), fconfigure(n)

KEYWORDS

blocking, channel, end of line, end of file, nonblocking, read, translation

NAME

regexp − Match a regular expression against a string

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 left-most 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 *subMatchVar*s. 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** has been specified or to an empty string otherwise.

REGULAR EXPRESSIONS

Regular expressions are implemented using Henry Spencer's package (thanks, Henry!), 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** chooses 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 leftmost matching sub-expression is chosen.

[3] In *, **+**, and **?** constructs, longer matches are chosen 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.

KEYWORDS

match, regular expression, string

NAME

registry − Manipulate the Windows registry

SYNOPSIS

package require registry 1.0

registry *option keyName* ?*arg arg* ...?

DESCRIPTION

The **registry** package provides a general set of operations for manipulating the Windows registry. The package implements the **registry** Tcl command. This command is only supported on the Windows platform. Warning: this command should be used with caution as a corrupted registry can leave your system in an unusable state.

KeyName is the name of a registry key. Registry keys must be one of the following forms:

\\hostname\rootname\keypath

rootname\keypath

rootname

Hostname specifies the name of any valid Windows host that exports its registry. The *rootname* component must be one of **HKEY_LOCAL_MACHINE**, **HKEY_USERS**, **HKEY_CLASSES_ROOT**, **HKEY_CURRENT_USER**, or **HKEY_CURRENT_CONFIG**. The *keypath* can be one or more registry key names separated by backslash (\) characters.

Option indicates what to do with the registry key name. Any unique abbreviation for *option* is acceptable. The valid options are:

registry delete *keyName* ?*valueName*?

If the optional *valueName* argument is present, the specified value under *keyName* will be deleted from the registry. If the optional *valueName* is omitted, the specified key and any subkeys or values beneath it in the registry heirarchy will be deleted. If the key could not be deleted then an error is generated. If the key did not exist, the command has no effect.

registry get *keyName valueName*

Returns the data associated with the value *valueName* under the key *keyName*. If either the key or the value does not exist, then an error is generated. For more details on the format of the returned data, see SUPPORTED TYPES, below.

registry keys *keyName* ?*pattern*?

If *pattern* isn't specified, returns a list of names of all the subkeys of *keyName*. If *pattern* is specified, only those names matching *pattern* are returned. Matching is determined using the same rules as for **string match**. If the specified *keyName* does not exist, then an error is generated.

registry set *keyName* ?*valueName data* ?*type*??

If *valueName* isn't specified, creates the key *keyName* if it doesn't already exist. If *valueName* is specified, creates the key *keyName* and value *valueName* if necessary. The contents of *valueName* are set to *data* with the type indicated by *type*. If *type* isn't specified, the type **sz** is assumed. For more details on the data and type arguments, see SUPPORTED TYPES below.

registry type *keyName valueName*

Returns the type of the value *valueName* in the key *keyName*. For more information on the possible types, see SUPPORTED TYPES, below.

registry values *keyName* ?*pattern*?

If *pattern* isn't specified, returns a list of names of all the values of *keyName*. If *pattern* is

specified, only those names matching *pattern* are returned. Matching is determined using the same rules as for **string match**.

SUPPORTED TYPES

Each value under a key in the registry contains some data of a particular type in a type-specific representation. The **registry** command converts between this internal representation and one that can be manipulated by Tcl scripts. In most cases, the data is simply returned as a Tcl string. The type indicates the intended use for the data, but does not actually change the representation. For some types, the **registry** command returns the data in a different form to make it easier to manipulate. The following types are recognized by the registry command:

binary The registry value contains arbitrary binary data. The data is represented exactly in Tcl, including any embedded nulls. Tcl

none The registry value contains arbitrary binary data with no defined type. The data is represented exactly in Tcl, including any embedded nulls.

sz The registry value contains a null-terminated string. The data is represented in Tcl as a string.

expand_sz The registry value contains a null-terminated string that contains unexpanded references to environment variables in the normal Windows style (for example, "%PATH%"). The data is represented in Tcl as a string.

dword The registry value contains a little-endian 32-bit number. The data is represented in Tcl as a decimal string.

dword_big_endian The registry value contains a big-endian 32-bit number. The data is represented in Tcl as a decimal string.

link The registry value contains a symbolic link. The data is represented exactly in Tcl, including any embedded nulls.

multi_sz The registry value contains an array of null-terminated strings. The data is represented in Tcl as a list of strings.

resource_list The registry value contains a device-driver resource list. The data is represented exactly in Tcl, including any embedded nulls.

In addition to the symbolically named types listed above, unknown types are identified using a 32-bit integer that corresponds to the type code returned by the system interfaces. In this case, the data is represented exactly in Tcl, including any embedded nulls.

PORTABILITY ISSUES

The registry command is only available on Windows.

KEYWORDS

registry

NAME

regsub − Perform substitutions based on regular expression pattern matching

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*. 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's 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 −.

The command returns a count of the number of matching ranges that were found and replaced. See the manual entry for **regexp** for details on the interpretation of regular expressions.

KEYWORDS

match, pattern, regular expression, substitute

NAME

rename − Rename or delete a command

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. *oldName* and *newName* may include namespace qualifiers (names of containing namespaces). If a command is renamed into a different namespace, future invocations of it will execute in the new namespace. The **rename** command returns an empty string as result.

KEYWORDS

command, delete, namespace, rename

NAME

resource − Manipulate Macintosh resources

SYNOPSIS

resource *option* ?*arg arg ...*?

DESCRIPTION

The **resource** command provides some generic operations for dealing with Macintosh resources. This command is only supported on the Macintosh platform. Each Macintosh file consists of two *forks*: a *data* fork and a *resource* fork. You use the normal open, puts, close, etc. commands to manipulate the data fork. You must use this command, however, to interact with the resource fork. *Option* indicates what resource command to perform. Any unique abbreviation for *option* is acceptable. The valid options are:

resource close *rsrcRef*

Closes the given resource reference (obtained from **resource open**). Resources from that resource file will no longer be available.

resource delete ?*options*? *resourceType*

This command will delete the resource specified by *options* and type *resourceType* (see RESOURCE TYPES below). The options give you several ways to specify the resource to be deleted.

−**id** *resourceId*

If the **-id** option is given the id *resourceId* (see RESOURCE IDS below) is used to specify the resource to be deleted. The id must be a number - to specify a name use the −**name** option.

−**name** *resourceName*

If **-name** is specified, the resource named *resourceName* will be deleted. If the **-id** is also provided, then there must be a resource with BOTH this name and this id. If no name is provided, then the id will be used regardless of the name of the actual resource.

−**file** *resourceRef*

If the **-file** option is specified then the resource will be deleted from the file pointed to by *resourceRef*. Otherwise the first resource with the given *resourceName* and or *resourceId* which is found on the resource file path will be deleted. To inspect the file path, use the *resource files* **command.**

resource files ?*resourceRef*?

If *resourceRef*is not provided, this command returns a Tcl list of the resource references for all the currently open resource files. The list is in the normal Macintosh search order for resources. If *resourceRef* is specified, the command will return the path to the file whose resource fork is represented by that token.

resource list *resourceType* ?*resourceRef*?

List all of the resources ids of type *resourceType* (see RESOURCE TYPES below). If *resourceRef* is specified then the command will limit the search to that particular resource file. Otherwise, all resource files currently opened by the application will be searched. A Tcl list of either the resource name's or resource id's of the found resources will be returned. See the RESOURCE IDS section below for more details about what a resource id is.

resource open *fileName* ?*permissions*?

Open the resource for the file *fileName*. Standard file permissions may also be specified (see the manual entry for **open** for details). A resource reference (*resourceRef*) is returned that can be used by the other resource commands. An error can occur if the file doesn't exist or the file does not have a resource fork. However, if you open the file with write permissions the file and/or resource

fork will be created instead of generating an error.

resource read *resourceType resourceId* ?*resourceRef*?
Read the entire resource of type *resourceType* (see RESOURCE TYPES below) and the name or id of *resourceId* (see RESOURCE IDS below) into memory and return the result. If *resourceRef* is specified we limit our search to that resource file, otherwise we search all open resource forks in the application. It is important to note that most Macintosh resource use a binary format and the data returned from this command may have embedded NULLs or other non-ASCII data.

resource types ?*resourceRef*?
This command returns a Tcl list of all resource types (see RESOURCE TYPES below) found in the resource file pointed to by *resourceRef*. If *resourceRef* is not specified it will return all the resource types found in every resource file currently opened by the application.

resource write ?*options*? *resourceType data*
This command will write the passed in *data* as a new resource of type *resourceType* (see RESOURCE TYPES below). Several options are available that describe where and how the resource is stored.

−**id** *resourceId*
If the **-id** option is given the id *resourceId* (see RESOURCE IDS below) is used for the new resource, otherwise a unique id will be generated that will not conflict with any existing resource. However, the id must be a number - to specify a name use the −**name** option.

−**name** *resourceName*
If **-name** is specified the resource will be named *resourceName*, otherwise it will have the empty string as the name.

−**file** *resourceRef*
If the **-file** option is specified then the resource will be written in the file pointed to by *resourceRef*, otherwise the most resently open resource will be used.

−**force** If the target resource already exists, then by default Tcl will not overwrite it, but raise an error instead. Use the -force flag to force overwriting the extant resource.

RESOURCE TYPES

Resource types are defined as a four character string that is then mapped to an underlying id. For example, **TEXT** refers to the Macintosh resource type for text. The type **STR#** is a list of counted strings. All Macintosh resources must be of some type. See Macintosh documentation for a more complete list of resource types that are commonly used.

RESOURCE IDS

For this command the notion of a resource id actually refers to two ideas in Macintosh resources. Every place you can use a resource Id you can use either the resource name or a resource number. Names are always searched or returned in preference to numbers. For example, the **resource list** command will return names if they exist or numbers if the name is NULL.

SEE ALSO

open

PORTABILITY ISSUES

The resource command is only available on Macintosh.

KEYWORDS

open, resource

NAME

return − Return from a procedure

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**.

KEYWORDS

break, continue, error, procedure, return

NAME

Safe Base − A mechanism for creating and manipulating safe interpreters.

SYNOPSIS

::safe::interpCreate ?*slave*? ?*options...*?

::safe::interpInit *slave* ?*options...*?

::safe::interpConfigure *slave* ?*options...*?

::safe::interpDelete *slave*

::safe::interpAddToAccessPath *slave directory*

::safe::interpFindInAccessPath *slave directory*

::safe::setLogCmd ?*cmd arg...*?

OPTIONS

?**−accessPath** *pathList*? ?**−statics** *boolean*? ?**−noStatics**? ?**−nested** *boolean*? ?**−nestedLoadOk**? ?**−deleteHook** *script*?

DESCRIPTION

Safe Tcl is a mechanism for executing untrusted Tcl scripts safely and for providing mediated access by such scripts to potentially dangerous functionality.

The Safe Base ensures that untrusted Tcl scripts cannot harm the hosting application. The Safe Base prevents integrity and privacy attacks. Untrusted Tcl scripts are prevented from corrupting the state of the hosting application or computer. Untrusted scripts are also prevented from disclosing information stored on the hosting computer or in the hosting application to any party.

The Safe Base allows a master interpreter to create safe, restricted interpreters that contain a set of predefined aliases for the **source**, **load**, **file** and **exit** commands and are able to use the auto-loading and package mechanisms.

No knowledge of the file system structure is leaked to the safe interpreter, because it has access only to a virtualized path containing tokens. When the safe interpreter requests to source a file, it uses the token in the virtual path as part of the file name to source; the master interpreter transparently translates the token into a real directory name and executes the requested operation (see the section **SECURITY** below for details). Different levels of security can be selected by using the optional flags of the commands described below.

All commands provided in the master interpreter by the Safe Base reside in the **safe** namespace:

COMMANDS

The following commands are provided in the master interpreter:

::safe::interpCreate ?*slave*? ?*options...*?

Creates a safe interpreter, installs the aliases described in the section **ALIASES** and initializes the auto-loading and package mechanism as specified by the supplied **options**. See the **OPTIONS** section below for a description of the optional arguments. If the *slave* argument is omitted, a name will be generated. **::safe::interpCreate** always returns the interpreter name.

::safe::interpInit *slave* ?*options...*?

This command is similar to **interpCreate** except it that does not create the safe interpreter. *slave*

must have been created by some other means, like **interp create −safe**.

::safe::interpConfigure *slave* ?*options...*?

If no *options* are given, returns the settings for all options for the named safe interpreter as a list of options and their current values for that *slave*. If a single additional argument is provided, it will return a list of 2 elements *name* and *value* where *name* is the full name of that option and *value* the current value for that option and the *slave*. If more than two additional arguments are provided, it will reconfigure the safe interpreter and change each and only the provided options. See the section on **OPTIONS** below for options description. Example of use:

```
# Create a new interp with the same configuration as "$i0" :
set i1 [eval safe::interpCreate [safe::interpConfigure $i0]]
# Get the current deleteHook
set dh [safe::interpConfigure $i0 −del]
# Change (only) the statics loading ok attribute of an interp
# and its deleteHook (leaving the rest unchanged) :
safe::interpConfigure $i0 −delete {foo bar} −statics 0 ;
```

::safe::interpDelete *slave*

Deletes the safe interpreter and cleans up the corresponding master interpreter data structures. If a *deleteHook* script was specified for this interpreter it is evaluated before the interpreter is deleted, with the name of the interpreter as an additional argument.

::safe::interpFindInAccessPath *slave directory*

This command finds and returns the token for the real directory *directory* in the safe interpreter's current virtual access path. It generates an error if the directory is not found. Example of use:

```
$slave eval [list set tk_library [::safe::interpFindInAccessPath $name $tk_library]]
```

::safe::interpAddToAccessPath *slave directory*

This command adds *directory* to the virtual path maintained for the safe interpreter in the master, and returns the token that can be used in the safe interpreter to obtain access to files in that directory. If the directory is already in the virtual path, it only returns the token without adding the directory to the virtual path again. Example of use:

```
$slave eval [list set tk_library [::safe::interpAddToAccessPath $name $tk_library]]
```

::safe::setLogCmd ?*cmd arg...*?

This command installs a script that will be called when interesting life cycle events occur for a safe interpreter. When called with no arguments, it returns the currently installed script. When called with one argument, an empty string, the currently installed script is removed and logging is turned off. The script will be invoked with one additional argument, a string describing the event of interest. The main purpose is to help in debugging safe interpreters. Using this facility you can get complete error messages while the safe interpreter gets only generic error messages. This prevents a safe interpreter from seeing messages about failures and other events that might contain sensitive information such as real directory names.

Example of use:

```
::safe::setLogCmd puts stderr
```

Below is the output of a sample session in which a safe interpreter attempted to source a file not found in its virtual access path. Note that the safe interpreter only received an error message saying that the file was not found:

```
NOTICE for slave interp10 : Created
NOTICE for slave interp10 : Setting accessPath=(/foo/bar) staticsok=1 nestedok=0 deletehook=()
NOTICE for slave interp10 : auto_path in interp10 has been set to {$p(:0:)}
ERROR for slave interp10 : /foo/bar/init.tcl: no such file or directory
```

OPTIONS

The following options are common to **::safe::interpCreate**, **::safe::interpInit**, and **::safe::interpConfigure**. Any option name can be abbreviated to its minimal non-ambiguous name. Option names are not case sensitive.

−accessPath *directoryList*
: This option sets the list of directories from which the safe interpreter can **source** and **load** files. If this option is not specified, or if it is given as the empty list, the safe interpreter will use the same directories as its master for auto-loading. See the section **SECURITY** below for more detail about virtual paths, tokens and access control.

−statics *boolean*
: This option specifies if the safe interpreter will be allowed to load statically linked packages (like **load {} Tk**). The default value is **true** : safe interpreters are allowed to load statically linked packages.

−noStatics
: This option is a convenience shortcut for **-statics false** and thus specifies that the safe interpreter will not be allowed to load statically linked packages.

−nested *boolean*
: This option specifies if the safe interpreter will be allowed to load packages into its own sub-interpreters. The default value is **false** : safe interpreters are not allowed to load packages into their own sub-interpreters.

−nestedLoadOk
: This option is a convenience shortcut for **-nested true** and thus specifies the safe interpreter will be allowed to load packages into its own sub-interpreters.

−deleteHook *script*
: When this option is given an non empty *script*, it will be evaluated in the master with the name of the safe interpreter as an additional argument just before actually deleting the safe interpreter. Giving an empty value removes any currently installed deletion hook script for that safe interpreter. The default value (**{}**) is not to have any deletion call back.

ALIASES

The following aliases are provided in a safe interpreter:

source *fileName*
: The requested file, a Tcl source file, is sourced into the safe interpreter if it is found. The **source** alias can only source files from directories in the virtual path for the safe interpreter. The **source** alias requires the safe interpreter to use one of the token names in its virtual path to denote the directory in which the file to be sourced can be found. See the section on **SECURITY** for more discussion of restrictions on valid filenames.

load *fileName*
: The requested file, a shared object file, is dynamically loaded into the safe interpreter if it is found. The filename must contain a token name mentioned in the virtual path for the safe interpreter for it to be found successfully. Additionally, the shared object file must contain a safe entry point; see the manual page for the **load** command for more details.

file ?*subCmd args...*?
: The **file** alias provides access to a safe subset of the subcommands of the **file** command; it allows only **dirname**, **join**, **extension**, **root**, **tail**, **pathname** and **split** subcommands. For more details on what these subcommands do see the manual page for the **file** command.

exit The calling interpreter is deleted and its computation is stopped, but the Tcl process in which this interpreter exists is not terminated.

SECURITY

The Safe Base does not attempt to completely prevent annoyance and denial of service attacks. These forms of attack prevent the application or user from temporarily using the computer to perform useful work, for example by consuming all available CPU time or all available screen real estate. These attacks, while aggravating, are deemed to be of lesser importance in general than integrity and privacy attacks that the Safe Base is to prevent.

The commands available in a safe interpreter, in addition to the safe set as defined in **interp** manual page, are mediated aliases for **source**, **load**, **exit**, and a safe subset of **file**. The safe interpreter can also auto-load code and it can request that packages be loaded.

Because some of these commands access the local file system, there is a potential for information leakage about its directory structure. To prevent this, commands that take file names as arguments in a safe interpreter use tokens instead of the real directory names. These tokens are translated to the real directory name while a request to, e.g., source a file is mediated by the master interpreter. This virtual path system is maintained in the master interpreter for each safe interpreter created by **::safe::interpCreate** or initialized by **::safe::interpInit** and the path maps tokens accessible in the safe interpreter into real path names on the local file system thus preventing safe interpreters from gaining knowledge about the structure of the file system of the host on which the interpreter is executing. The only valid file names arguments for the **source** and **load** aliases provided to the slave are path in the form of **[file join** *token filename***]** (ie, when using the native file path formats: *token*/*filename* on Unix, *token**filename* on Windows, and *token*:*filename* on the Mac), where *token* is representing one of the directories of the *accessPath* list and *filename* is one file in that directory (no sub directories access are allowed).

When a token is used in a safe interpreter in a request to source or load a file, the token is checked and translated to a real path name and the file to be sourced or loaded is located on the file system. The safe interpreter never gains knowledge of the actual path name under which the file is stored on the file system.

To further prevent potential information leakage from sensitive files that are accidentally included in the set of files that can be sourced by a safe interpreter, the **source** alias restricts access to files meeting the following constraints: the file name must fourteen characters or shorter, must not contain more than one dot ("**.**"), must end up with the extension **.tcl** or be called **tclIndex**.

Each element of the initial access path list will be assigned a token that will be set in the slave **auto_path** and the first element of that list will be set as the **tcl_library** for that slave.

If the access path argument is not given or is the empty list, the default behavior is to let the slave access the same packages as the master has access to (Or to be more precise: only packages written in Tcl (which by definition can't be dangerous as they run in the slave interpreter) and C extensions that provides a Safe_Init entry point). For that purpose, the master's **auto_path** will be used to construct the slave access path. In order that the slave successfully loads the Tcl library files (which includes the auto-loading mechanism itself) the **tcl_library** will be added or moved to the first position if necessary, in the slave access path, so the slave **tcl_library** will be the same as the master's (its real path will still be invisible to the slave though). In order that auto-loading works the same for the slave and the master in this by default case, the first-level sub directories of each directory in the master **auto_path** will also be added (if not already included) to the slave access path. You can always specify a more restrictive path for which sub directories will never be searched by explicitly specifying your directory list with the **−accessPath** flag instead of relying on this default mechanism.

When the *accessPath* is changed after the first creation or initialization (ie through **interpConfigure -accessPath** *list*), an **auto_reset** is automatically evaluated in the safe interpreter to synchronize its **auto_index** with the new token list.

SEE ALSO

interp(n), library(n), load(n), package(n), source(n), unknown(n)

KEYWORDS

alias, auto−loading, auto_mkindex, load, master interpreter, safe interpreter, slave interpreter, source

NAME

scan − Parse string using conversion specifiers in the style of sscanf

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 conversions performed, or -1 if the end of the input string is reached before any conversions have been performed. *String* gives the input to be parsed and *format* indicates how to parse it, using **%** conversion specifiers 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 matches any number of white space characters in *string* (including zero). Otherwise, if it isn't a **%** character then it must match the next character of *string*. When a **%** is encountered in *formatString*, it indicates the start of a conversion specifier. A conversion specifier 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 specifier in *formatString*, it first skips any white-space characters in *string*. Then it converts the next input characters according to the conversion specifier 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 (e.g. 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 specifier 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 specifiers 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).

KEYWORDS

conversion specifier, parse, scan

NAME

seek − Change the access position for an open channel

SYNOPSIS

seek *channelId offset* ?*origin*?

DESCRIPTION

Changes the current access position for *channelId*. *ChannelId* must be a channel identifier such as returned from a previous invocation of **open** or **socket**. The *offset* and *origin* arguments specify the position at which the next read or write will occur for *channelId*. *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* bytes from the start of the underlying file or device.

current The new access position will be *offset* bytes from the current access position; a negative *offset* moves the access position backwards in the underlying file or device.

end The new access position will be *offset* bytes from the end of the file or device. 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**.

The command flushes all buffered output for the channel before the command returns, even if the channel is in nonblocking mode. It also discards any buffered and unread input. This command returns an empty string. An error occurs if this command is applied to channels whose underlying file or device does not support seeking.

KEYWORDS

access position, file, seek

NAME

set − Read and write variables

SYNOPSIS

set *varName* ?*value*?

DESCRIPTION

Returns the value of variable *varName*. If *value* is specified, then set the value of *varName* to *value*, creating a new variable if one doesn't already exist, and return 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. Normally, *varName* is unqualified (does not include the names of any containing namespaces), and the variable of that name in the current namespace is read or written. If *varName* includes namespace qualifiers (in the array name if it refers to an array element), the variable in the specified namespace is read or written.

If no procedure is active, then *varName* refers to a namespace variable (global variable if the current namespace is the global namespace). If a procedure is active, then *varName* refers to a parameter or local variable of the procedure unless the **global** command was invoked to declare *varName* to be global, or unless a **variable** command was invoked to declare *varName* to be a namespace variable.

KEYWORDS

read, write, variable

NAME

socket − Open a TCP network connection

SYNOPSIS

socket ?*options*? *host port*

socket −server *command* ?*options*? *port*

DESCRIPTION

This command opens a network socket and returns a channel identifier that may be used in future invocations of commands like **read**, **puts** and **flush**. At present only the TCP network protocol is supported; future releases may include support for additional protocols. The **socket** command may be used to open either the client or server side of a connection, depending on whether the **−server** switch is specified.

CLIENT SOCKETS

If the **−server** option is not specified, then the client side of a connection is opened and the command returns a channel identifier that can be used for both reading and writing. *Port* and *host* specify a port to connect to; there must be a server accepting connections on this port. *Port* is an integer port number and *host* is either a domain-style name such as **www.sunlabs.com** or a numerical IP address such as **127.0.0.1**. Use *localhost* to refer to the host on which the command is invoked.

The following options may also be present before *host* to specify additional information about the connection:

−myaddr *addr*

Addr gives the domain-style name or numerical IP address of the client-side network interface to use for the connection. This option may be useful if the client machine has multiple network interfaces. If the option is omitted then the client-side interface will be chosen by the system software.

−myport *port*

Port specifies an integer port number to use for the client's side of the connection. If this option is omitted, the client's port number will be chosen at random by the system software.

−async The **−async** option will cause the client socket to be connected asynchronously. This means that the socket will be created immediately but may not yet be connected to the server, when the call to **socket** returns. When a **gets** or **flush** is done on the socket before the connection attempt succeeds or fails, if the socket is in blocking mode, the operation will wait until the connection is completed or fails. If the socket is in nonblocking mode and a **gets** or **flush** is done on the socket before the connection attempt succeeds or fails, the operation returns immediately and **fblocked** on the socket returns 1.

SERVER SOCKETS

If the **−server** option is specified then the new socket will be a server for the port given by *port*. Tcl will automatically accept connections to the given port. For each connection Tcl will create a new channel that may be used to communicate with the client. Tcl then invokes *command* with three additional arguments: the name of the new channel, the address, in network address notation, of the client's host, and the client's port number.

The following additional option may also be specified before *host*:

−myaddr *addr*

Addr gives the domain-style name or numerical IP address of the server-side network interface to use for the connection. This option may be useful if the server machine has multiple network interfaces. If the option is omitted then the server socket is bound to the special address

INADDR_ANY so that it can accept connections from any interface.

Server channels cannot be used for input or output; their sole use is to accept new client connections. The channels created for each incoming client connection are opened for input and output. Closing the server channel shuts down the server so that no new connections will be accepted; however, existing connections will be unaffected.

Server sockets depend on the Tcl event mechanism to find out when new connections are opened. If the application doesn't enter the event loop, for example by invoking the **vwait** command or calling the C procedure **Tcl_DoOneEvent**, then no connections will be accepted.

CONFIGURATION OPTIONS

The **fconfigure** command can be used to query several readonly configuration options for socket channels:

−sockname

This option returns a list of three elements, the address, the host name and the port number for the socket. If the host name cannot be computed, the second element is identical to the address, the first element of the list.

−peername

This option is not supported by server sockets. For client and accepted sockets, this option returns a list of three elements; these are the address, the host name and the port to which the peer socket is connected or bound. If the host name cannot be computed, the second element of the list is identical to the address, its first element.

SEE ALSO

flush(n), open(n), read(n)

KEYWORDS

bind, channel, connection, domain name, host, network address, socket, tcp

NAME

source − Evaluate a file or resource as a Tcl script

SYNOPSIS

source *fileName*

source −rsrc *resourceName* ?*fileName*?

source −rsrcid *resourceId* ?*fileName*?

DESCRIPTION

This command takes the contents of the specified file or resource and passes it to the Tcl interpreter as a text script. The return value from **source** is the return value of the last command executed in the script. If an error occurs in evaluating the contents of the script then the **source** command will return that error. If a **return** command is invoked from within the script then the remainder of the file will be skipped and the **source** command will return normally with the result from the **return** command.

The *−rsrc* and *−rsrcid* forms of this command are only available on Macintosh computers. These versions of the command allow you to source a script from a **TEXT** resource. You may specify what **TEXT** resource to source by either name or id. By default Tcl searches all open resource files, which include the current application and any loaded C extensions. Alternatively, you may specify the *fileName* where the **TEXT** resource can be found.

KEYWORDS

file, script

NAME

split − Split a string into a proper Tcl list

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"**.

KEYWORDS

list, split, string

NAME

string − Manipulate strings

SYNOPSIS

string *option arg* ?*arg ...?*

DESCRIPTION

Performs one of several string operations, depending on *option*. The legal *option*s (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 *charIndex*'th 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*

See if *pattern* matches *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 *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 *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. An index of **end** (or any abbreviation of it) refers 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).

string wordend *string index*
Returns the index of the character just after the last one in the word containing character *index* of *string*. A word is considered to be any contiguous range of alphanumeric or underscore characters, or any single character other than these.

string wordstart *string index*
Returns the index of the first character in the word containing character *index* of *string*. A word is considered to be any contiguous range of alphanumeric or underscore characters, or any single character other than these.

KEYWORDS
case conversion, compare, index, match, pattern, string, word

NAME

subst − Perform backslash, command, and variable substitutions

SYNOPSIS

subst ?**−nobackslashes**? ?**−nocommands**? ?**−novariables**? *string*

DESCRIPTION

This command performs variable substitutions, command substitutions, and backslash substitutions on its *string* argument and returns the fully-substituted result. The substitutions are performed in exactly the same way as for Tcl commands. As a result, the *string* argument is actually substituted twice, once by the Tcl parser in the usual fashion for Tcl commands, and again by the *subst* command.

If any of the **−nobackslashes**, **−nocommands**, or **−novariables** are specified, then the corresponding substitutions are not performed. For example, if **−nocommands** is specified, no command substitution is performed: open and close brackets are treated as ordinary characters with no special interpretation.

Note: when it performs its substitutions, *subst* does not give any special treatment to double quotes or curly braces. For example, the script

```
set a 44
subst {xyz {$a}}
```

returns ‘‘**xyz {44}**’’, not ‘‘**xyz {$a}**’’.

KEYWORDS

backslash substitution, command substitution, variable substitution

NAME

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

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* {format 2}
    default {format 3}
}
```

will return **1**, and

```
switch xyz {
    a
        −
    b
```

```
            {format 1}
        a*
            {format 2}
        default
            {format 3}
    }
```

will return **3**.

KEYWORDS

switch, match, regular expression

NAME

tclvars − Variables used by Tcl

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 This variable 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 doesn't 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.
Under Windows, the environment variables PATH and COMSPEC in any capitalization are converted automatically to upper case. For instance, the PATH variable could be exported by the operating system as ''path'', ''Path'', ''PaTh'', etc., causing otherwise simple Tcl code to have to support many special cases. All other environment variables inherited by Tcl are left unmodified.
On the Macintosh, the environment variable is constructed by Tcl as no global environment variable exists. The environment variables that are created for Tcl include:

LOGIN
This holds the Chooser name of the Macintosh.

USER This also holds the Chooser name of the Macintosh.

SYS_FOLDER
The path to the system directory.

APPLE_M_FOLDER
The path to the Apple Menu directory.

CP_FOLDER
The path to the control panels directory.

DESK_FOLDER
The path to the desk top directory.

EXT_FOLDER
The path to the system extensions directory.

PREF_FOLDER
The path to the preferences directory.

PRINT_MON_FOLDER
The path to the print monitor directory.

SHARED_TRASH_FOLDER
The path to the network trash directory.

TRASH_FOLDER
The path to the trash directory.

START_UP_FOLDER
The path to the start up directory.

PWD The path to the application's default directory.

You can also create your own environment variables for the Macintosh. A file named *Tcl Environment Variables* may be placed in the preferences folder in the Mac system folder. Each line of this file should be of the form *VAR_NAME=var_data*.

The last alternative is to place environment variables in a 'STR#' resource named *Tcl Environment Variables* of the application. This is considered a little more ''Mac like'' than a Unix style Environment Variable file. Each entry in the 'STR#' resource has the same format as above. The source code file *tclMacEnv.c* contains the implementation of the env mechanisms. This file contains many #define's that allow customization of the env mechanisms to fit your applications needs.

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 errors, 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*

This 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), OVERFLOW (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_library

This variable holds the name of a directory containing the system library of Tcl scripts, such as those used for auto-loading. The value of this variable is returned by the **info library** command. See the **library** manual entry for details of the facilities provided by the Tcl script library. Normally each application or package will have its own application-specific script library in addition to the Tcl script library; each application should set a global variable with a name like **$***app***_library** (where *app* is the application's name) to hold the network file name for that application's library directory. The initial value of **tcl_library** is set when an interpreter is created by searching several different directories until one is found that contains an appropriate Tcl startup script. If the **TCL_LIBRARY** environment variable exists, then the directory it names is checked first. If **TCL_LIBRARY** isn't set or doesn't refer to an appropriate directory, then Tcl checks several other directories based on a compiled-in default location, the location of the binary containing the application, and the current working directory.

tcl_patchLevel

When an interpreter is created Tcl initializes this variable to hold a string giving the current patch level for Tcl, such as **7.3p2** for Tcl 7.3 with the first two official patches, or **7.4b4** for the fourth beta release of Tcl 7.4. The value of this variable is returned by the **info patchlevel** command.

tcl_pkgPath

This variable holds a list of directories indicating where packages are normally installed. It typically contains either one or two entries; if it contains two entries, the first is normally a directory for platform-dependent packages (e.g., shared library binaries) and the second is normally a directory for platform-independent packages (e.g., script files). Typically a package is installed as a subdirectory of one of the entries in **$tcl_pkgPath**. The directories in **$tcl_pkgPath** are included by default in the **auto_path** variable, so they and their immediate subdirectories are automatically searched for packages during **package require** commands. Note: **tcl_pkgPath** it not intended to be modified by the application. Its value is added to **auto_path** at startup; changes to **tcl_pkgPath** are not reflected in **auto_path**. If you want Tcl to search additional directories for packages you should add the names of those directories to **auto_path**, not **tcl_pkgPath**.

tcl_platform

This is an associative array whose elements contain information about the platform on which the application is running, such as the name of the operating system, its current release number, and the machine's instruction set. The elements listed below will always be defined, but they may have empty strings as values if Tcl couldn't retrieve any relevant information. In addition, extensions and applications may add additional values to the array. The predefined elements are:

byteOrder

The native byte order of this machine: either **littleEndian** or **bigEndian**.

machine

The instruction set executed by this machine, such as **intel**, **PPC**, **68k**, or **sun4m**. On UNIX machines, this is the value returned by **uname -m**.

os The name of the operating system running on this machine, such as **Win32s**, **Windows NT**, **MacOS**, or **SunOS**. On UNIX machines, this is the value returned by **uname -s**.

osVersion
The version number for the operating system running on this machine. On UNIX machines, this is the value returned by **uname -r**.

platform
Either **windows**, **macintosh**, or **unix**. This identifies the general operating environment of the machine.

tcl_precision
This variable controls the number of digits to generate when converting floating-point values to strings. It defaults to 12. 17 digits is ''perfect'' for IEEE floating-point in that it allows double-precision values to be converted to strings and back to binary with no loss of information. However, using 17 digits prevents any rounding, which produces longer, less intuitive results. For example, **expr 1.4** returns 1.3999999999999999 with **tcl_precision** set to 17, vs. 1.4 if **tcl_precision** is 12.
All interpreters in a process share a single **tcl_precision** value: changing it in one interpreter will affect all other interpreters as well. However, safe interpreters are not allowed to modify the variable.

tcl_rcFileName
This variable is used during initialization to indicate the name of a user-specific startup file. If it is set by application-specific initialization, then the Tcl startup code will check for the existence of this file and **source** it if it exists. For example, for **wish** the variable is set to **~/.wishrc** for Unix and **~/wishrc.tcl** for Windows.

tcl_rcRsrcName
This variable is only used on Macintosh systems. The variable is used during initialization to indicate the name of a user-specific **TEXT** resource located in the application or extension resource forks. If it is set by application-specific initialization, then the Tcl startup code will check for the existence of this resource and **source** it if it exists. For example, the Macintosh **wish** application has the variable is set to **tclshrc**.

tcl_traceCompile
The value of this variable can be set to control how much tracing information is displayed during bytecode compilation. By default, tcl_traceCompile is zero and no information is displayed. Setting tcl_traceCompile to 1 generates a one line summary in stdout whenever a procedure or top level command is compiled. Setting it to 2 generates a detailed listing in stdout of the bytecode instructions emitted during every compilation. This variable is useful in tracking down suspected problems with the Tcl compiler. It is also occasionally useful when converting existing code to use Tcl8.0.

tcl_traceExec
The value of this variable can be set to control how much tracing information is displayed during bytecode execution. By default, tcl_traceExec is zero and no information is displayed. Setting tcl_traceExec to 1 generates a one line trace in stdout on each call to a Tcl procedure. Setting it to 2 generates a line of output whenever any Tcl command is invoked that contains the name of the command and its arguments. Setting it to 3 produces a detailed trace showing the result of executing each bytecode instruction. Note that when tcl_traceExec is 2 or 3, commands such as set and incr that have been entirely replaced by a sequence of bytecode instructions are not shown. Setting this variable is useful in tracking down suspected problems with the bytecode compiler and interpreter. It is also occasionally useful when converting code to use Tcl8.0.

tcl_version

When an interpreter is created Tcl initializes this variable to hold the version number for this version of Tcl in the form *x.y*. Changes to *x* represent major changes with probable incompatibilities and changes to *y* represent small enhancements and bug fixes that retain backward compatibility. The value of this variable is returned by the **info tclversion** command.

KEYWORDS

arithmetic, bytecode, compiler, error, environment, POSIX, precision, subprocess, variables

NAME

tell − Return current access position for an open channel

SYNOPSIS

tell *channelId*

DESCRIPTION

Returns a decimal string giving the current access position in *channelId*. The value returned is -1 for channels that do not support seeking.

KEYWORDS

access position, channel, seeking

NAME

time − Time the execution of a script

SYNOPSIS

time *script* ?*count*?

DESCRIPTION

This command will call the Tcl interpreter *count* times to evaluate *script* (or once if *count* isn't 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.

KEYWORDS

script, time

NAME

trace − Monitor variable accesses

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 *option*'s (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 the 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. *Name1* and *name2* are not necessarily the same as the name used in the **trace variable** command: the **upvar** command allows a procedure to reference a variable under a different name. *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.

KEYWORDS

read, variable, write, trace, unset

NAME

unknown − Handle attempts to use non-existent commands

SYNOPSIS

unknown *cmdName* ?*arg arg* ...?

DESCRIPTION

This command is invoked by the Tcl interpreter whenever a script tries to invoke a command that doesn't exist. The implementation of **unknown** isn't part of the Tcl core; instead, it is a library procedure defined by default when Tcl starts up. You can override the default **unknown** to change its functionality.

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.

The default implementation of **unknown** behaves as follows. It first calls the **auto_load** library procedure 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* can't 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, then **unknown** carries out history substitution in the same way that **csh** would for these constructs. Finally, **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 then 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.

KEYWORDS

error, non-existent command

NAME

unset − Delete variables

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.

KEYWORDS

remove, variable

NAME

update − Process pending events and idle callbacks

SYNOPSIS

update ?**idletasks**?

DESCRIPTION

This command is used to bring the application ‘‘up to date’’ by entering the event loop repeated until all pending events (including idle callbacks) have been processed.

If the **idletasks** keyword is specified as an argument to the command, then no new events or errors are processed; only idle callbacks are invoked. This causes operations that are normally deferred, such as display updates and window layout calculations, to be performed immediately.

The **update idletasks** command is useful in scripts where changes have been made to the application's state and you want those changes to appear on the display immediately, rather than waiting for the script to complete. Most display updates are performed as idle callbacks, so **update idletasks** will cause them to run. However, there are some kinds of updates that only happen in response to events, such as those triggered by window size changes; these updates will not occur in **update idletasks**.

The **update** command with no options is useful in scripts where you are performing a long-running computation but you still want the application to respond to events such as user interactions; if you occasionally call **update** then user input will be processed during the next call to **update**.

KEYWORDS

event, flush, handler, idle, update

NAME

uplevel − Execute a script in a different stack frame

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).

namespace eval is another way (besides procedure calls) that the Tcl naming context can change. It adds a call frame to the stack to represent the namespace context. This means each **namespace eval** command counts as another call level for **uplevel** and **upvar** commands. For example, **info level 1** will return a list describing a command that is either the outermost procedure call or the outermost **namespace eval** command. Also, **uplevel #0** evaluates a script at top-level in the outermost namespace (the global namespace).

SEE ALSO

namespace(n)

KEYWORDS

context, level, namespace, stack frame, variables

NAME

upvar − Create link to variable in a different stack frame

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. There must not exist a variable by the name *myVar* at the time **upvar** is invoked. *MyVar* is always treated as the name of a variable, not an array element. Even if the name looks like an array element, such as **a(b)**, a regular variable is created. *OtherVar* may refer to a scalar variable, an array, or 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.

namespace eval is another way (besides procedure calls) that the Tcl naming context can change. It adds a call frame to the stack to represent the namespace context. This means each **namespace eval** command counts as another call level for **uplevel** and **upvar** commands. For example, **info level 1** will return a list describing a command that is either the outermost procedure call or the outermost **namespace eval** command. Also, **uplevel #0** evaluates a script at top-level in the outermost namespace (the global namespace).

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.

BUGS

If *otherVar* refers to an element of an array, then variable traces set for the entire array will not be invoked when *myVar* is accessed (but traces on the particular element will still be invoked). In particular, if the array is **env**, then changes made to *myVar* will not be passed to subprocesses correctly.

SEE ALSO

namespace(n)

KEYWORDS

context, frame, global, level, namespace, procedure, variable

NAME

variable − create and initialize a namespace variable

SYNOPSIS

variable ?*name value...*? *name* ?*value*?

DESCRIPTION

This command is normally used within a **namespace eval** command to create one or more variables within a namespace. Each variable *name* is initialized with *value*. The *value* for the last variable is optional.

If a variable *name* does not exist, it is created. In this case, if *value* is specified, it is assigned to the newly created variable. If no *value* is specified, the new variable is left undefined. If the variable already exists, it is set to *value* if *value* is specified or left unchanged if no *value* is given. Normally, *name* is unqualified (does not include the names of any containing namespaces), and the variable is created in the current namespace. If *name* includes any namespace qualifiers, the variable is created in the specified namespace.

If the **variable** command is executed inside a Tcl procedure, it creates local variables linked to the corresponding namespace variables. In this way the **variable** command resembles the **global** command, although the **global** command only links to variables in the global namespace. If any *value*s are given, they are used to modify the values of the associated namespace variables. If a namespace variable does not exist, it is created and optionally initialized.

A *name* argument cannot reference an element within an array. Instead, *name* should reference the entire array, and the initialization *value* should be left off. After the variable has been declared, elements within the array can be set using ordinary **set** or **array** commands.

SEE ALSO

global(n), namespace(n)

KEYWORDS

global, namespace, procedure, variable

NAME

vwait − Process events until a variable is written

SYNOPSIS

vwait *varName*

DESCRIPTION

This command enters the Tcl event loop to process events, blocking the application if no events are ready. It continues processing events until some event handler sets the value of variable *varName*. Once *varName* has been set, the **vwait** command will return as soon as the event handler that modified *varName* completes.

In some cases the **vwait** command may not return immediately after *varName* is set. This can happen if the event handler that sets *varName* does not complete immediately. For example, if an event handler sets *varName* and then itself calls **vwait** to wait for a different variable, then it may not return for a long time. During this time the top-level **vwait** is blocked waiting for the event handler to complete, so it cannot return either.

KEYWORDS

event, variable, wait

NAME

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

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.

Note: *test* should almost always be enclosed in braces. If not, variable substitutions will be made before the **while** command starts executing, which means that variable changes made by the loop body will not be considered in the expression. This is likely to result in an infinite loop. If *test* is enclosed in braces, variable substitutions are delayed until the expression is evaluated (before each loop iteration), so changes in the variables will be visible. For an example, try the following script with and without the braces around **$x<10**:

```
set x 0
while {$x<10} {
    puts "x is $x"
    incr x
}
```

KEYWORDS

boolean value, loop, test, while

NAME

bell − Ring a display's bell

SYNOPSIS

bell ?**−displayof** *window*?

DESCRIPTION

This command rings the bell on the display for *window* and returns an empty string. If the **−displayof** option is omitted, the display of the application's main window is used by default. The command uses the current bell-related settings for the display, which may be modified with programs such as **xset**.

This command also resets the screen saver for the screen. Some screen savers will ignore this, but others will reset so that the screen becomes visible again.

KEYWORDS

beep, bell, ring

NAME

bind − Arrange for X events to invoke Tcl scripts

SYNOPSIS

bind *tag*

bind *tag sequence*

bind *tag sequence script*

bind *tag sequence +script*

INTRODUCTION

The **bind** command associates Tcl scripts with X events. If all three arguments are specified, **bind** will arrange for *script* (a Tcl script) to be evaluated whenever the event(s) given by *sequence* occur in the window(s) identified by *tag*. If *script* is prefixed with a ‘‘+’’, then it is appended to any existing binding for *sequence*; otherwise *script* replaces any existing binding. If *script* is an empty string then the current binding for *sequence* is destroyed, leaving *sequence* unbound. In all of the cases where a *script* argument is provided, **bind** returns an empty string.

If *sequence* is specified without a *script*, then the script currently bound to *sequence* is returned, or an empty string is returned if there is no binding for *sequence*. If neither *sequence* nor *script* is specified, then the return value is a list whose elements are all the sequences for which there exist bindings for *tag*.

The *tag* argument determines which window(s) the binding applies to. If *tag* begins with a dot, as in **.a.b.c**, then it must be the path name for a window; otherwise it may be an arbitrary string. Each window has an associated list of tags, and a binding applies to a particular window if its tag is among those specified for the window. Although the **bindtags** command may be used to assign an arbitrary set of binding tags to a window, the default binding tags provide the following behavior:

If a tag is the name of an internal window the binding applies to that window.

If the tag is the name of a toplevel window the binding applies to the toplevel window and all its internal windows.

If the tag is the name of a class of widgets, such as **Button**, the binding applies to all widgets in that class;

If *tag* has the value **all**, the binding applies to all windows in the application.

EVENT PATTERNS

The *sequence* argument specifies a sequence of one or more event patterns, with optional white space between the patterns. Each event pattern may take one of three forms. In the simplest case it is a single printing ASCII character, such as **a** or **[**. The character may not be a space character or the character <. This form of pattern matches a **KeyPress** event for the particular character. The second form of pattern is longer but more general. It has the following syntax:

<*modifier-modifier-type-detail*>

The entire event pattern is surrounded by angle brackets. Inside the angle brackets are zero or more modifiers, an event type, and an extra piece of information (*detail*) identifying a particular button or keysym. Any of the fields may be omitted, as long as at least one of *type* and *detail* is present. The fields must be separated by white space or dashes.

The third form of pattern is used to specify a user-defined, named virtual event. It has the following syntax:

<<*name*>>

The entire virtual event pattern is surrounded by double angle brackets. Inside the angle brackets is the

user-defined name of the virtual event. Modifiers, such as **Shift** or **Control**, may not be combined with a virtual event to modify it. Bindings on a virtual event may be created before the virtual event is defined, and if the definition of a virtual event changes dynamically, all windows bound to that virtual event will respond immediately to the new definition.

MODIFIERS

Modifiers consist of any of the following values:

Control	**Mod2, M2**
Shift	**Mod3, M3**
Lock	**Mod4, M4**
Button1, B1	**Mod5, M5**
Button2, B2	**Meta, M**
Button3, B3	**Alt**
Button4, B4	**Double**
Button5, B5	**Triple**
Mod1, M1	

Where more than one value is listed, separated by commas, the values are equivalent. Most of the modifiers have the obvious X meanings. For example, **Button1** requires that button 1 be depressed when the event occurs. For a binding to match a given event, the modifiers in the event must include all of those specified in the event pattern. An event may also contain additional modifiers not specified in the binding. For example, if button 1 is pressed while the shift and control keys are down, the pattern **<Control-Button-1>** will match the event, but **<Mod1-Button-1>** will not. If no modifiers are specified, then any combination of modifiers may be present in the event.

Meta and **M** refer to whichever of the **M1** through **M5** modifiers is associated with the meta key(s) on the keyboard (keysyms **Meta_R** and **Meta_L**). If there are no meta keys, or if they are not associated with any modifiers, then **Meta** and **M** will not match any events. Similarly, the **Alt** modifier refers to whichever modifier is associated with the alt key(s) on the keyboard (keysyms **Alt_L** and **Alt_R**).

The **Double** and **Triple** modifiers are a convenience for specifying double mouse clicks and other repeated events. They cause a particular event pattern to be repeated 2 or 3 times, and also place a time and space requirement on the sequence: for a sequence of events to match a **Double** or **Triple** pattern, all of the events must occur close together in time and without substantial mouse motion in between. For example, **<Double-Button-1>** is equivalent to **<Button-1><Button-1>** with the extra time and space requirement.

EVENT TYPES

The *type* field may be any of the standard X event types, with a few extra abbreviations. Below is a list of all the valid types; where two names appear together, they are synonyms.

ButtonPress, Button	**Expose**	**Map**
ButtonRelease	**FocusIn**	**Motion**
Circulate	**FocusOut**	**Property**
Colormap	**Gravity**	**Reparent**
Configure	**KeyPress, Key**	**Unmap**
Destroy	**KeyRelease**	**Visibility**
Enter	**Leave**	**Activate**
Deactivate		

The last part of a long event specification is *detail*. In the case of a **ButtonPress** or **ButtonRelease** event, it is the number of a button (1-5). If a button number is given, then only an event on that particular button will match; if no button number is given, then an event on any button will match. Note: giving a specific

button number is different than specifying a button modifier; in the first case, it refers to a button being pressed or released, while in the second it refers to some other button that is already depressed when the matching event occurs. If a button number is given then *type* may be omitted: if will default to **ButtonPress**. For example, the specifier **<1>** is equivalent to **<ButtonPress-1>**.

If the event type is **KeyPress** or **KeyRelease**, then *detail* may be specified in the form of an X keysym. Keysyms are textual specifications for particular keys on the keyboard; they include all the alphanumeric ASCII characters (e.g. ''a'' is the keysym for the ASCII character ''a''), plus descriptions for non-alphanumeric characters (''comma'' is the keysym for the comma character), plus descriptions for all the non-ASCII keys on the keyboard (''Shift_L'' is the keysm for the left shift key, and ''F1'' is the keysym for the F1 function key, if it exists). The complete list of keysyms is not presented here; it is available in other X documentation and may vary from system to system. If necessary, you can use the **%K** notation described below to print out the keysym name for a particular key. If a keysym *detail* is given, then the *type* field may be omitted; it will default to **KeyPress**. For example, **<Control-comma>** is equivalent to **<Control-KeyPress-comma>**.

BINDING SCRIPTS AND SUBSTITUTIONS

The *script* argument to **bind** is a Tcl script, which will be executed whenever the given event sequence occurs. *Command* will be executed in the same interpreter that the **bind** command was executed in, and it will run at global level (only global variables will be accessible). If *script* contains any **%** characters, then the script will not be executed directly. Instead, a new script will be generated by replacing each **%**, and the character following it, with information from the current event. The replacement depends on the character following the **%**, as defined in the list below. Unless otherwise indicated, the replacement string is the decimal value of the given field from the current event. Some of the substitutions are only valid for certain types of events; if they are used for other types of events the value substituted is undefined.

%% Replaced with a single percent.

%# The number of the last client request processed by the server (the *serial* field from the event). Valid for all event types.

%a The *above* field from the event, formatted as a hexadecimal number. Valid only for **Configure** events.

%b The number of the button that was pressed or released. Valid only for **ButtonPress** and **ButtonRelease** events.

%c The *count* field from the event. Valid only for **Expose** events.

%d The *detail* field from the event. The **%d** is replaced by a string identifying the detail. For **Enter**, **Leave**, **FocusIn**, and **FocusOut** events, the string will be one of the following:

NotifyAncestor	**NotifyNonlinearVirtual**
NotifyDetailNone	**NotifyPointer**
NotifyInferior	**NotifyPointerRoot**
NotifyNonlinear	**NotifyVirtual**

For events other than these, the substituted string is undefined.

%f The *focus* field from the event (**0** or **1**). Valid only for **Enter** and **Leave** events.

%h The *height* field from the event. Valid only for **Configure** and **Expose** events.

%k The *keycode* field from the event. Valid only for **KeyPress** and **KeyRelease** events.

%m The *mode* field from the event. The substituted string is one of **NotifyNormal**, **NotifyGrab**, **NotifyUngrab**, or **NotifyWhileGrabbed**. Valid only for **Enter**, **FocusIn**, **FocusOut**, and **Leave** events.

%o The *override_redirect* field from the event. Valid only for **Map**, **Reparent**, and **Configure** events.

%p The *place* field from the event, substituted as one of the strings **PlaceOnTop** or **PlaceOnBottom**. Valid only for **Circulate** events.

%s The *state* field from the event. For **ButtonPress**, **ButtonRelease**, **Enter**, **KeyPress**, **KeyRelease**, **Leave**, and **Motion** events, a decimal string is substituted. For **Visibility**, one of the strings **VisibilityUnobscured**, **VisibilityPartiallyObscured**, and **VisibilityFullyObscured** is substituted.

%t The *time* field from the event. Valid only for events that contain a *time* field.

%w The *width* field from the event. Valid only for **Configure** and **Expose** events.

%x The *x* field from the event. Valid only for events containing an *x* field.

%y The *y* field from the event. Valid only for events containing a *y* field.

%A Substitutes the ASCII character corresponding to the event, or the empty string if the event doesn't correspond to an ASCII character (e.g. the shift key was pressed). **XLookupString** does all the work of translating from the event to an ASCII character. Valid only for **KeyPress** and **KeyRelease** events.

%B The *border_width* field from the event. Valid only for **Configure** events.

%E The *send_event* field from the event. Valid for all event types.

%K The keysym corresponding to the event, substituted as a textual string. Valid only for **KeyPress** and **KeyRelease** events.

%N The keysym corresponding to the event, substituted as a decimal number. Valid only for **KeyPress** and **KeyRelease** events.

%R The *root* window identifier from the event. Valid only for events containing a *root* field.

%S The *subwindow* window identifier from the event, formatted as a hexadecimal number. Valid only for events containing a *subwindow* field.

%T The *type* field from the event. Valid for all event types.

%W The path name of the window to which the event was reported (the *window* field from the event). Valid for all event types.

%X The *x_root* field from the event. If a virtual-root window manager is being used then the substituted value is the corresponding x-coordinate in the virtual root. Valid only for **ButtonPress**, **ButtonRelease**, **KeyPress**, **KeyRelease**, and **Motion** events.

%Y The *y_root* field from the event. If a virtual-root window manager is being used then the substituted value is the corresponding y-coordinate in the virtual root. Valid only for **ButtonPress**, **ButtonRelease**, **KeyPress**, **KeyRelease**, and **Motion** events.

The replacement string for a %-replacement is formatted as a proper Tcl list element. This means that it will be surrounded with braces if it contains spaces, or special characters such as **$** and **{** may be preceded by backslashes. This guarantees that the string will be passed through the Tcl parser when the binding script is evaluated. Most replacements are numbers or well-defined strings such as **Above**; for these replacements no special formatting is ever necessary. The most common case where reformatting occurs is for the **%A** substitution. For example, if *script* is

```
insert %A
```

and the character typed is an open square bracket, then the script actually executed will be

```
insert \[
```

This will cause the **insert** to receive the original replacement string (open square bracket) as its first argument. If the extra backslash hadn't been added, Tcl would not have been able to parse the script correctly.

MULTIPLE MATCHES

It is possible for several bindings to match a given X event. If the bindings are associated with different *tag*'s, then each of the bindings will be executed, in order. By default, a binding for the widget will be executed first, followed by a class binding, a binding for its toplevel, and an **all** binding. The **bindtags** command may be used to change this order for a particular window or to associate additional binding tags with the window.

The **continue** and **break** commands may be used inside a binding script to control the processing of matching scripts. If **continue** is invoked, then the current binding script is terminated but Tk will continue processing binding scripts associated with other *tag*'s. If the **break** command is invoked within a binding script, then that script terminates and no other scripts will be invoked for the event.

If more than one binding matches a particular event and they have the same *tag*, then the most specific binding is chosen and its script is evaluated. The following tests are applied, in order, to determine which of several matching sequences is more specific: (a) an event pattern that specifies a specific button or key is more specific than one that doesn't; (b) a longer sequence (in terms of number of events matched) is more specific than a shorter sequence; (c) if the modifiers specified in one pattern are a subset of the modifiers in another pattern, then the pattern with more modifiers is more specific. (d) a virtual event whose physical pattern matches the sequence is less specific than the same physical pattern that is not associated with a virtual event. (e) given a sequence that matches two or more virtual events, one of the virtual events will be chosen, but the order is undefined.

If the matching sequences contain more than one event, then tests (c)-(e) are applied in order from the most recent event to the least recent event in the sequences. If these tests fail to determine a winner, then the most recently registered sequence is the winner.

If there are two (or more) virtual events that are both triggered by the same sequence, and both of those virtual events are bound to the same window tag, then only one of the virtual events will be triggered, and it will be picked at random:

```
event add <<Paste>> <Control-y>
event add <<Paste>> <Button-2>
event add <<Scroll>> <Button-2>
bind Entry <<Paste>> {puts Paste}
bind Entry <<Scroll>> {puts Scroll}
```

If the user types Control-y, the **<<Paste>>** binding will be invoked, but if the user presses button 2 then one of either the **<<Paste>>** or the **<<Scroll>>** bindings will be invoked, but exactly which one gets invoked is undefined.

If an X event does not match any of the existing bindings, then the event is ignored. An unbound event is not considered to be an error.

MULTI-EVENT SEQUENCES AND IGNORED EVENTS

When a *sequence* specified in a **bind** command contains more than one event pattern, then its script is executed whenever the recent events (leading up to and including the current event) match the given sequence. This means, for example, that if button 1 is clicked repeatedly the sequence **<Double-ButtonPress-1>** will match each button press but the first. If extraneous events that would prevent a match occur in the middle of an event sequence then the extraneous events are ignored unless they are **KeyPress** or **ButtonPress** events. For example, **<Double-ButtonPress-1>** will match a sequence of presses of button 1, even though there will be **ButtonRelease** events (and possibly **Motion** events) between the **ButtonPress** events. Furthermore, a **KeyPress** event may be preceded by any number of other **KeyPress** events for modifier keys without the modifier keys preventing a match. For example, the event sequence **aB** will match a press of the **a** key, a release of the **a** key, a press of the **Shift** key, and a press of the **b** key: the press of **Shift** is ignored because it is a modifier key. Finally, if several **Motion** events occur in a row, only the last one is used for purposes of matching binding sequences.

ERRORS

If an error occurs in executing the script for a binding then the **bgerror** mechanism is used to report the error. The **bgerror** command will be executed at global level (outside the context of any Tcl procedure).

SEE ALSO

bgerror

KEYWORDS

form, manual

NAME

bindtags − Determine which bindings apply to a window, and order of evaluation

SYNOPSIS

bindtags *window* ?*tagList*?

DESCRIPTION

When a binding is created with the **bind** command, it is associated either with a particular window such as **.a.b.c**, a class name such as **Button**, the keyword **all**, or any other string. All of these forms are called *binding tags*. Each window contains a list of binding tags that determine how events are processed for the window. When an event occurs in a window, it is applied to each of the window's tags in order: for each tag, the most specific binding that matches the given tag and event is executed. See the **bind** command for more information on the matching process.

By default, each window has four binding tags consisting of the name of the window, the window's class name, the name of the window's nearest toplevel ancestor, and **all**, in that order. Toplevel windows have only three tags by default, since the toplevel name is the same as that of the window. The **bindtags** command allows the binding tags for a window to be read and modified.

If **bindtags** is invoked with only one argument, then the current set of binding tags for *window* is returned as a list. If the *tagList* argument is specified to **bindtags**, then it must be a proper list; the tags for *window* are changed to the elements of the list. The elements of *tagList* may be arbitrary strings; however, any tag starting with a dot is treated as the name of a window; if no window by that name exists at the time an event is processed, then the tag is ignored for that event. The order of the elements in *tagList* determines the order in which binding scripts are executed in response to events. For example, the command

bindtags .b {all . Button .b}

reverses the order in which binding scripts will be evaluated for a button named **.b** so that **all** bindings are invoked first, following by bindings for **.b**'s toplevel ("."), followed by class bindings, followed by bindings for **.b**. If *tagList* is an empty list then the binding tags for *window* are returned to the default state described above.

The **bindtags** command may be used to introduce arbitrary additional binding tags for a window, or to remove standard tags. For example, the command

bindtags .b {.b TrickyButton . all}

replaces the **Button** tag for **.b** with **TrickyButton**. This means that the default widget bindings for buttons, which are associated with the **Button** tag, will no longer apply to **.b**, but any bindings associated with **TrickyButton** (perhaps some new button behavior) will apply.

SEE ALSO

bind

KEYWORDS

binding, event, tag

NAME

bitmap − Images that display two colors

SYNOPSIS

image create bitmap ?*name*? ?*options*?

DESCRIPTION

A bitmap is an image whose pixels can display either of two colors or be transparent. A bitmap image is defined by four things: a background color, a foreground color, and two bitmaps, called the *source* and the *mask*. Each of the bitmaps specifies 0/1 values for a rectangular array of pixels, and the two bitmaps must have the same dimensions. For pixels where the mask is zero, the image displays nothing, producing a transparent effect. For other pixels, the image displays the foreground color if the source data is one and the background color if the source data is zero.

CREATING BITMAPS

Like all images, bitmaps are created using the **image create** command. Bitmaps support the following *options*:

−background *color*

Specifies a background color for the image in any of the standard ways accepted by Tk. If this option is set to an empty string then the background pixels will be transparent. This effect is achieved by using the source bitmap as the mask bitmap, ignoring any **−maskdata** or **−maskfile** options.

−data *string*

Specifies the contents of the source bitmap as a string. The string must adhere to X11 bitmap format (e.g., as generated by the **bitmap** program). If both the **−data** and **−file** options are specified, the **−data** option takes precedence.

−file *name*

name gives the name of a file whose contents define the source bitmap. The file must adhere to X11 bitmap format (e.g., as generated by the **bitmap** program).

−foreground *color*

Specifies a foreground color for the image in any of the standard ways accepted by Tk.

−maskdata *string*

Specifies the contents of the mask as a string. The string must adhere to X11 bitmap format (e.g., as generated by the **bitmap** program). If both the **−maskdata** and **−maskfile** options are specified, the **−maskdata** option takes precedence.

−maskfile *name*

name gives the name of a file whose contents define the mask. The file must adhere to X11 bitmap format (e.g., as generated by the **bitmap** program).

IMAGE COMMAND

When a bitmap image is created, Tk also creates a new command whose name is the same as the image. This command may be used to invoke various operations on the image. It has the following general form:

imageName option ?*arg arg ...*?

Option and the *arg*s determine the exact behavior of the command. The following commands are possible for bitmap images:

imageName **cget** *option*

Returns the current value of the configuration option given by *option*. *Option* may have any of the

values accepted by the **image create bitmap** command.

imageName **configure** ?*option*? ?*value option value ...*?
Query or modify the configuration options for the image. If no *option* is specified, returns a list describing all of the available options for *imageName* (see **Tk_ConfigureInfo** for information on the format of this list). If *option* is specified with no *value*, then the command returns a list describing the one named option (this list will be identical to the corresponding sublist of the value returned if no *option* is specified). If one or more *option−value* pairs are specified, then the command modifies the given option(s) to have the given value(s); in this case the command returns an empty string. *Option* may have any of the values accepted by the **image create bitmap** command.

KEYWORDS
bitmap, image

NAME

button − Create and manipulate button widgets

SYNOPSIS

button *pathName* ?*options*?

STANDARD OPTIONS

−activebackground	**−cursor**	**−highlightthickness**	**−takefocus**
−activeforeground	**−disabledforeground**	**−image**	**−text**
−anchor	**−font**	**−justify**	**−textvariable**
−background	**−foreground**	**−padx**	**−underline**
−bitmap	**−highlightbackground**	**−pady**	**−wraplength**
−borderwidth	**−highlightcolor**	**−relief**	

See the **options** manual entry for details on the standard options.

WIDGET-SPECIFIC OPTIONS

Command-Line Name: **−command**
Database Name: **command**
Database Class: **Command**

Specifies a Tcl command to associate with the button. This command is typically invoked when mouse button 1 is released over the button window.

Command-Line Name: **−default**
Database Name: **default**
Database Class: **Default**

Specifies one of three states for the default ring: **normal**, **active**, or **disabled**. In active state, the button is drawn with the platform specific appearance for a default button. In normal state, the button is drawn with the platform specific appearance for a non-default button, leaving enough space to draw the default button appearance. The normal and active states will result in buttons of the same size. In disabled state, the button is drawn with the non-default button appearance without leaving space for the default appearance. The disabled state may result in a smaller button than the active state. ring.

Command-Line Name: **−height**
Database Name: **height**
Database Class: **Height**

Specifies a desired height for the button. If an image or bitmap is being displayed in the button then the value is in screen units (i.e. any of the forms acceptable to **Tk_GetPixels**); for text it is in lines of text. If this option isn't specified, the button's desired height is computed from the size of the image or bitmap or text being displayed in it.

Command-Line Name: **−state**
Database Name: **state**
Database Class: **State**

Specifies one of three states for the button: **normal**, **active**, or **disabled**. In normal state the button is displayed using the **foreground** and **background** options. The active state is typically used when the pointer is over the button. In active state the button is displayed using the **activeForeground** and **activeBackground** options. Disabled state means that the button should be insensitive: the default bindings will refuse to activate the widget and will ignore mouse button presses. In this state the **disabledForeground** and **background** options determine how the button is displayed.

Command-Line Name: **−width**
Database Name: **width**
Database Class: **Width**

Specifies a desired width for the button. If an image or bitmap is being displayed in the button then the value is in screen units (i.e. any of the forms acceptable to **Tk_GetPixels**); for text it is in characters. If this option isn't specified, the button's desired width is computed from the size of the image or bitmap or text being displayed in it.

DESCRIPTION

The **button** command creates a new window (given by the *pathName* argument) and makes it into a button widget. Additional options, described above, may be specified on the command line or in the option database to configure aspects of the button such as its colors, font, text, and initial relief. The **button** command returns its *pathName* argument. At the time this command is invoked, there must not exist a window named *pathName*, but *pathName*'s parent must exist.

A button is a widget that displays a textual string, bitmap or image. If text is displayed, it must all be in a single font, but it can occupy multiple lines on the screen (if it contains newlines or if wrapping occurs because of the **wrapLength** option) and one of the characters may optionally be underlined using the **underline** option. It can display itself in either of three different ways, according to the **state** option; it can be made to appear raised, sunken, or flat; and it can be made to flash. When a user invokes the button (by pressing mouse button 1 with the cursor over the button), then the Tcl command specified in the **−command** option is invoked.

WIDGET COMMAND

The **button** command creates a new Tcl command whose name is *pathName*. This command may be used to invoke various operations on the widget. It has the following general form:

pathName option ?*arg arg ...*?

Option and the *arg*s determine the exact behavior of the command. The following commands are possible for button widgets:

pathName **cget** *option*

Returns the current value of the configuration option given by *option*. *Option* may have any of the values accepted by the **button** command.

pathName **configure** ?*option*? ?*value option value ...*?

Query or modify the configuration options of the widget. If no *option* is specified, returns a list describing all of the available options for *pathName* (see **Tk_ConfigureInfo** for information on the format of this list). If *option* is specified with no *value*, then the command returns a list describing the one named option (this list will be identical to the corresponding sublist of the value returned if no *option* is specified). If one or more *option−value* pairs are specified, then the command modifies the given widget option(s) to have the given value(s); in this case the command returns an empty string. *Option* may have any of the values accepted by the **button** command.

pathName **flash**

Flash the button. This is accomplished by redisplaying the button several times, alternating between active and normal colors. At the end of the flash the button is left in the same normal/active state as when the command was invoked. This command is ignored if the button's state is **disabled**.

pathName **invoke**

Invoke the Tcl command associated with the button, if there is one. The return value is the return value from the Tcl command, or an empty string if there is no command associated with the

button. This command is ignored if the button's state is **disabled**.

DEFAULT BINDINGS

Tk automatically creates class bindings for buttons that give them default behavior:

[1] A button activates whenever the mouse passes over it and deactivates whenever the mouse leaves the button. Under Windows, this binding is only active when mouse button 1 has been pressed over the button.

[2] A button's relief is changed to sunken whenever mouse button 1 is pressed over the button, and the relief is restored to its original value when button 1 is later released.

[3] If mouse button 1 is pressed over a button and later released over the button, the button is invoked. However, if the mouse is not over the button when button 1 is released, then no invocation occurs.

[4] When a button has the input focus, the space key causes the button to be invoked.

If the button's state is **disabled** then none of the above actions occur: the button is completely non-responsive.

The behavior of buttons can be changed by defining new bindings for individual widgets or by redefining the class bindings.

KEYWORDS

button, widget

NAME

canvas − Create and manipulate canvas widgets

SYNOPSIS

canvas *pathName* ?*options*?

STANDARD OPTIONS

−background	**−highlightthickness**	**−insertwidth**	**−takefocus**
−borderwidth	**−insertbackground**	**−relief**	**−xscrollcommand**
−cursor	**−insertborderwidth**	**−selectbackground**	**−yscrollcommand**
−highlightbackground	**−insertofftime**	**−selectborderwidth**	
−highlightcolor	**−insertontime**	**−selectforeground**	

See the **options** manual entry for details on the standard options.

WIDGET-SPECIFIC OPTIONS

Command-Line Name: **−closeenough**
Database Name: **closeEnough**
Database Class: **CloseEnough**

Specifies a floating-point value indicating how close the mouse cursor must be to an item before it is considered to be ‘‘inside’’ the item. Defaults to 1.0.

Command-Line Name: **−confine**
Database Name: **confine**
Database Class: **Confine**

Specifies a boolean value that indicates whether or not it should be allowable to set the canvas’s view outside the region defined by the **scrollRegion** argument. Defaults to true, which means that the view will be constrained within the scroll region.

Command-Line Name: **−height**
Database Name: **height**
Database Class: **Height**

Specifies a desired window height that the canvas widget should request from its geometry manager. The value may be specified in any of the forms described in the COORDINATES section below.

Command-Line Name: **−scrollregion**
Database Name: **scrollRegion**
Database Class: **ScrollRegion**

Specifies a list with four coordinates describing the left, top, right, and bottom coordinates of a rectangular region. This region is used for scrolling purposes and is considered to be the boundary of the information in the canvas. Each of the coordinates may be specified in any of the forms given in the COORDINATES section below.

Command-Line Name: **−width**
Database Name: **width**
Database Class: **width**

Specifies a desired window width that the canvas widget should request from its geometry manager. The value may be specified in any of the forms described in the COORDINATES section below.

Command-Line Name: **−xscrollincrement**
Database Name: **xScrollIncrement**
Database Class: **ScrollIncrement**

Specifies an increment for horizontal scrolling, in any of the usual forms permitted for screen

distances. If the value of this option is greater than zero, the horizontal view in the window will be constrained so that the canvas x coordinate at the left edge of the window is always an even multiple of **xScrollIncrement**; furthermore, the units for scrolling (e.g., the change in view when the left and right arrows of a scrollbar are selected) will also be **xScrollIncrement**. If the value of this option is less than or equal to zero, then horizontal scrolling is unconstrained.

Command-Line Name: **−yscrollincrement**
Database Name: **yScrollIncrement**
Database Class: **ScrollIncrement**

Specifies an increment for vertical scrolling, in any of the usual forms permitted for screen distances. If the value of this option is greater than zero, the vertical view in the window will be constrained so that the canvas y coordinate at the top edge of the window is always an even multiple of **yScrollIncrement**; furthermore, the units for scrolling (e.g., the change in view when the top and bottom arrows of a scrollbar are selected) will also be **yScrollIncrement**. If the value of this option is less than or equal to zero, then vertical scrolling is unconstrained.

INTRODUCTION

The **canvas** command creates a new window (given by the *pathName* argument) and makes it into a canvas widget. Additional options, described above, may be specified on the command line or in the option database to configure aspects of the canvas such as its colors and 3-D relief. The **canvas** command returns its *pathName* argument. At the time this command is invoked, there must not exist a window named *pathName*, but *pathName*'s parent must exist.

Canvas widgets implement structured graphics. A canvas displays any number of *items*, which may be things like rectangles, circles, lines, and text. Items may be manipulated (e.g. moved or re-colored) and commands may be associated with items in much the same way that the **bind** command allows commands to be bound to widgets. For example, a particular command may be associated with the <Button-1> event so that the command is invoked whenever button 1 is pressed with the mouse cursor over an item. This means that items in a canvas can have behaviors defined by the Tcl scripts bound to them.

DISPLAY LIST

The items in a canvas are ordered for purposes of display, with the first item in the display list being displayed first, followed by the next item in the list, and so on. Items later in the display list obscure those that are earlier in the display list and are sometimes referred to as being ''on top'' of earlier items. When a new item is created it is placed at the end of the display list, on top of everything else. Widget commands may be used to re-arrange the order of the display list.

Window items are an exception to the above rules. The underlying window systems require them always to be drawn on top of other items. In addition, the stacking order of window items is not affected by any of the canvas widget commands; you must use the **raise** and **lower** Tk commands instead.

ITEM IDS AND TAGS

Items in a canvas widget may be named in either of two ways: by id or by tag. Each item has a unique identifying number which is assigned to that item when it is created. The id of an item never changes and id numbers are never re-used within the lifetime of a canvas widget.

Each item may also have any number of *tags* associated with it. A tag is just a string of characters, and it may take any form except that of an integer. For example, ''x123'' is OK but ''123'' isn't. The same tag may be associated with many different items. This is commonly done to group items in various interesting ways; for example, all selected items might be given the tag ''selected''.

The tag **all** is implicitly associated with every item in the canvas; it may be used to invoke operations on all the items in the canvas.

The tag **current** is managed automatically by Tk; it applies to the *current item*, which is the topmost item whose drawn area covers the position of the mouse cursor. If the mouse is not in the canvas widget or is not over an item, then no item has the **current** tag.

When specifying items in canvas widget commands, if the specifier is an integer then it is assumed to refer to the single item with that id. If the specifier is not an integer, then it is assumed to refer to all of the items in the canvas that have a tag matching the specifier. The symbol *tagOrId* is used below to indicate that an argument specifies either an id that selects a single item or a tag that selects zero or more items. Some widget commands only operate on a single item at a time; if *tagOrId* is specified in a way that names multiple items, then the normal behavior is for the command to use the first (lowest) of these items in the display list that is suitable for the command. Exceptions are noted in the widget command descriptions below.

COORDINATES

All coordinates related to canvases are stored as floating-point numbers. Coordinates and distances are specified in screen units, which are floating-point numbers optionally followed by one of several letters. If no letter is supplied then the distance is in pixels. If the letter is **m** then the distance is in millimeters on the screen; if it is **c** then the distance is in centimeters; **i** means inches, and **p** means printers points (1/72 inch). Larger y-coordinates refer to points lower on the screen; larger x-coordinates refer to points farther to the right.

TRANSFORMATIONS

Normally the origin of the canvas coordinate system is at the upper-left corner of the window containing the canvas. It is possible to adjust the origin of the canvas coordinate system relative to the origin of the window using the **xview** and **yview** widget commands; this is typically used for scrolling. Canvases do not support scaling or rotation of the canvas coordinate system relative to the window coordinate system.

Individual items may be moved or scaled using widget commands described below, but they may not be rotated.

INDICES

Text items support the notion of an *index* for identifying particular positions within the item. Indices are used for commands such as inserting text, deleting a range of characters, and setting the insertion cursor position. An index may be specified in any of a number of ways, and different types of items may support different forms for specifying indices. Text items support the following forms for an index; if you define new types of text-like items, it would be advisable to support as many of these forms as practical. Note that it is possible to refer to the character just after the last one in the text item; this is necessary for such tasks as inserting new text at the end of the item.

number A decimal number giving the position of the desired character within the text item. 0 refers to the first character, 1 to the next character, and so on. A number less than 0 is treated as if it were zero, and a number greater than the length of the text item is treated as if it were equal to the length of the text item.

end Refers to the character just after the last one in the item (same as the number of characters in the item).

insert Refers to the character just before which the insertion cursor is drawn in this item.

sel.first Refers to the first selected character in the item. If the selection isn't in this item then this form is illegal.

sel.last Refers to the last selected character in the item. If the selection isn't in this item then this form

is illegal.

@x,y Refers to the character at the point given by *x* and *y*, where *x* and *y* are specified in the coordinate system of the canvas. If *x* and *y* lie outside the coordinates covered by the text item, then they refer to the first or last character in the line that is closest to the given point.

WIDGET COMMAND

The **canvas** command creates a new Tcl command whose name is *pathName*. This command may be used to invoke various operations on the widget. It has the following general form:

pathName option ?*arg arg ...*?

Option and the *arg*s determine the exact behavior of the command. The following widget commands are possible for canvas widgets:

pathName **addtag** *tag searchSpec* ?*arg arg ...*?

For each item that meets the constraints specified by *searchSpec* and the *arg*s, add *tag* to the list of tags associated with the item if it isn't already present on that list. It is possible that no items will satisfy the constraints given by *searchSpec* and *arg*s, in which case the command has no effect. This command returns an empty string as result. *SearchSpec* and *arg*'s may take any of the following forms:

above *tagOrId*

Selects the item just after (above) the one given by *tagOrId* in the display list. If *tagOrId* denotes more than one item, then the last (topmost) of these items in the display list is used.

all Selects all the items in the canvas.

below *tagOrId*

Selects the item just before (below) the one given by *tagOrId* in the display list. If *tagOrId* denotes more than one item, then the first (lowest) of these items in the display list is used.

closest *x y* ?*halo*? ?*start*?

Selects the item closest to the point given by *x* and *y*. If more than one item is at the same closest distance (e.g. two items overlap the point), then the top-most of these items (the last one in the display list) is used. If *halo* is specified, then it must be a non-negative value. Any item closer than *halo* to the point is considered to overlap it. The *start* argument may be used to step circularly through all the closest items. If *start* is specified, it names an item using a tag or id (if by tag, it selects the first item in the display list with the given tag). Instead of selecting the topmost closest item, this form will select the topmost closest item that is below *start* in the display list; if no such item exists, then the selection behaves as if the *start* argument had not been specified.

enclosed *x1 y1 x2 y2*

Selects all the items completely enclosed within the rectangular region given by *x1*, *y1*, *x2*, and *y2*. *X1* must be no greater then *x2* and *y1* must be no greater than *y2*.

overlapping *x1 y1 x2 y2*

Selects all the items that overlap or are enclosed within the rectangular region given by *x1*, *y1*, *x2*, and *y2*. *X1* must be no greater then *x2* and *y1* must be no greater than *y2*.

withtag *tagOrId*

Selects all the items given by *tagOrId*.

pathName **bbox** *tagOrId* ?*tagOrId tagOrId ...*?

Returns a list with four elements giving an approximate bounding box for all the items named by the *tagOrId* arguments. The list has the form "*x1 y1 x2 y2*" such that the drawn areas of all the

named elements are within the region bounded by *x1* on the left, *x2* on the right, *y1* on the top, and *y2* on the bottom. The return value may overestimate the actual bounding box by a few pixels. If no items match any of the *tagOrId* arguments or if the matching items have empty bounding boxes (i.e. they have nothing to display) then an empty string is returned.

pathName **bind** *tagOrId* ?*sequence*? ?*command*?

This command associates *command* with all the items given by *tagOrId* such that whenever the event sequence given by *sequence* occurs for one of the items the command will be invoked. This widget command is similar to the **bind** command except that it operates on items in a canvas rather than entire widgets. See the **bind** manual entry for complete details on the syntax of *sequence* and the substitutions performed on *command* before invoking it. If all arguments are specified then a new binding is created, replacing any existing binding for the same *sequence* and *tagOrId* (if the first character of *command* is ''+'' then *command* augments an existing binding rather than replacing it). In this case the return value is an empty string. If *command* is omitted then the command returns the *command* associated with *tagOrId* and *sequence* (an error occurs if there is no such binding). If both *command* and *sequence* are omitted then the command returns a list of all the sequences for which bindings have been defined for *tagOrId*.

The only events for which bindings may be specified are those related to the mouse and keyboard (such as **Enter**, **Leave**, **ButtonPress**, **Motion**, and **KeyPress**) or virtual events. The handling of events in canvases uses the current item defined in ITEM IDS AND TAGS above. **Enter** and **Leave** events trigger for an item when it becomes the current item or ceases to be the current item; note that these events are different than **Enter** and **Leave** events for windows. Mouse-related events are directed to the current item, if any. Keyboard-related events are directed to the focus item, if any (see the **focus** widget command below for more on this). If a virtual event is used in a binding, that binding can trigger only if the virtual event is defined by an underlying mouse-related or keyboard-related event.

It is possible for multiple bindings to match a particular event. This could occur, for example, if one binding is associated with the item's id and another is associated with one of the item's tags. When this occurs, all of the matching bindings are invoked. A binding associated with the **all** tag is invoked first, followed by one binding for each of the item's tags (in order), followed by a binding associated with the item's id. If there are multiple matching bindings for a single tag, then only the most specific binding is invoked. A **continue** command in a binding script terminates that script, and a **break** command terminates that script and skips any remaining scripts for the event, just as for the **bind** command.

If bindings have been created for a canvas window using the **bind** command, then they are invoked in addition to bindings created for the canvas's items using the **bind** widget command. The bindings for items will be invoked before any of the bindings for the window as a whole.

pathName **canvasx** *screenx* ?*gridspacing*?

Given a window x-coordinate in the canvas *screenx*, this command returns the canvas x-coordinate that is displayed at that location. If *gridspacing* is specified, then the canvas coordinate is rounded to the nearest multiple of *gridspacing* units.

pathName **canvasy** *screeny* ?*gridspacing*?

Given a window y-coordinate in the canvas *screeny* this command returns the canvas y-coordinate that is displayed at that location. If *gridspacing* is specified, then the canvas coordinate is rounded to the nearest multiple of *gridspacing* units.

pathName **cget** *option*

Returns the current value of the configuration option given by *option*. *Option* may have any of the values accepted by the **canvas** command.

pathName **configure ?***option*? ?*value*? ?*option value ...*?

Query or modify the configuration options of the widget. If no *option* is specified, returns a list describing all of the available options for *pathName* (see **Tk_ConfigureInfo** for information on the format of this list). If *option* is specified with no *value*, then the command returns a list describing the one named option (this list will be identical to the corresponding sublist of the value returned if no *option* is specified). If one or more *option−value* pairs are specified, then the command modifies the given widget option(s) to have the given value(s); in this case the command returns an empty string. *Option* may have any of the values accepted by the **canvas** command.

pathName **coords** *tagOrId* ?*x0 y0 ...*?

Query or modify the coordinates that define an item. If no coordinates are specified, this command returns a list whose elements are the coordinates of the item named by *tagOrId*. If coordinates are specified, then they replace the current coordinates for the named item. If *tagOrId* refers to multiple items, then the first one in the display list is used.

pathName **create** *type x y* ?*x y ...*? ?*option value ...*?

Create a new item in *pathName* of type *type*. The exact format of the arguments after **type** depends on **type**, but usually they consist of the coordinates for one or more points, followed by specifications for zero or more item options. See the subsections on individual item types below for more on the syntax of this command. This command returns the id for the new item.

pathName **dchars** *tagOrId first* ?*last*?

For each item given by *tagOrId*, delete the characters in the range given by *first* and *last*, inclusive. If some of the items given by *tagOrId* don't support text operations, then they are ignored. *First* and *last* are indices of characters within the item(s) as described in INDICES above. If *last* is omitted, it defaults to *first*. This command returns an empty string.

pathName **delete** ?*tagOrId tagOrId ...*?

Delete each of the items given by each *tagOrId*, and return an empty string.

pathName **dtag** *tagOrId* ?*tagToDelete*?

For each of the items given by *tagOrId*, delete the tag given by *tagToDelete* from the list of those associated with the item. If an item doesn't have the tag *tagToDelete* then the item is unaffected by the command. If *tagToDelete* is omitted then it defaults to *tagOrId*. This command returns an empty string.

pathName **find** *searchCommand* ?*arg arg ...*?

This command returns a list consisting of all the items that meet the constraints specified by *searchCommand* and *arg*'s. *SearchCommand* and *args* have any of the forms accepted by the **addtag** command. The items are returned in stacking order, with the lowest item first.

pathName **focus** ?*tagOrId*?

Set the keyboard focus for the canvas widget to the item given by *tagOrId*. If *tagOrId* refers to several items, then the focus is set to the first such item in the display list that supports the insertion cursor. If *tagOrId* doesn't refer to any items, or if none of them support the insertion cursor, then the focus isn't changed. If *tagOrId* is an empty string, then the focus item is reset so that no item has the focus. If *tagOrId* is not specified then the command returns the id for the item that currently has the focus, or an empty string if no item has the focus.

Once the focus has been set to an item, the item will display the insertion cursor and all keyboard events will be directed to that item. The focus item within a canvas and the focus window on the screen (set with the **focus** command) are totally independent: a given item doesn't actually have the input focus unless (a) its canvas is the focus window and (b) the item is the focus item within the canvas. In most cases it is advisable to follow the **focus** widget command with the **focus** command to set the focus window to the canvas (if it wasn't there already).

pathName **gettags** *tagOrId*

Return a list whose elements are the tags associated with the item given by *tagOrId*. If *tagOrId*

refers to more than one item, then the tags are returned from the first such item in the display list. If *tagOrId* doesn't refer to any items, or if the item contains no tags, then an empty string is returned.

pathName **icursor** *tagOrId index*
Set the position of the insertion cursor for the item(s) given by *tagOrId* to just before the character whose position is given by *index*. If some or all of the items given by *tagOrId* don't support an insertion cursor then this command has no effect on them. See INDICES above for a description of the legal forms for *index*. Note: the insertion cursor is only displayed in an item if that item currently has the keyboard focus (see the widget command **focus**, below), but the cursor position may be set even when the item doesn't have the focus. This command returns an empty string.

pathName **index** *tagOrId index*
This command returns a decimal string giving the numerical index within *tagOrId* corresponding to *index*. *Index* gives a textual description of the desired position as described in INDICES above. The return value is guaranteed to lie between 0 and the number of characters within the item, inclusive. If *tagOrId* refers to multiple items, then the index is processed in the first of these items that supports indexing operations (in display list order).

pathName **insert** *tagOrId beforeThis string*
For each of the items given by *tagOrId*, if the item supports text insertion then *string* is inserted into the item's text just before the character whose index is *beforeThis*. See INDICES above for information about the forms allowed for *beforeThis*. This command returns an empty string.

pathName **itemcget** *tagOrId option*
Returns the current value of the configuration option for the item given by *tagOrId* whose name is *option*. This command is similar to the **cget** widget command except that it applies to a particular item rather than the widget as a whole. *Option* may have any of the values accepted by the **create** widget command when the item was created. If *tagOrId* is a tag that refers to more than one item, the first (lowest) such item is used.

pathName **itemconfigure** *tagOrId* ?*option*? ?*value*? ?*option value ...*?
This command is similar to the **configure** widget command except that it modifies item-specific options for the items given by *tagOrId* instead of modifying options for the overall canvas widget. If no *option* is specified, returns a list describing all of the available options for the first item given by *tagOrId* (see **Tk_ConfigureInfo** for information on the format of this list). If *option* is specified with no *value*, then the command returns a list describing the one named option (this list will be identical to the corresponding sublist of the value returned if no *option* is specified). If one or more *option−value* pairs are specified, then the command modifies the given widget option(s) to have the given value(s) in each of the items given by *tagOrId*; in this case the command returns an empty string. The *option*s and *value*s are the same as those permissible in the **create** widget command when the item(s) were created; see the sections describing individual item types below for details on the legal options.

pathName **lower** *tagOrId* ?*belowThis*?
Move all of the items given by *tagOrId* to a new position in the display list just before the item given by *belowThis*. If *tagOrId* refers to more than one item then all are moved but the relative order of the moved items will not be changed. *BelowThis* is a tag or id; if it refers to more than one item then the first (lowest) of these items in the display list is used as the destination location for the moved items. Note: this command has no effect on window items. Window items always obscure other item types, and the stacking order of window items is determined by the **raise** and **lower** commands, not the **raise** and **lower** widget commands for canvases. This command returns an empty string.

pathName **move** *tagOrId xAmount yAmount*
Move each of the items given by *tagOrId* in the canvas coordinate space by adding *xAmount* to the

x-coordinate of each point associated with the item and *yAmount* to the y-coordinate of each point associated with the item. This command returns an empty string.

pathName **postscript** ?*option value option value ...*?

Generate a Postscript representation for part or all of the canvas. If the **−file** option is specified then the Postscript is written to a file and an empty string is returned; otherwise the Postscript is returned as the result of the command. If the interpreter that owns the canvas is marked as safe, the operation will fail because safe interpreters are not allowed to write files. If the **−channel** option is specified, the argument denotes the name of a channel already opened for writing. The Postscript is written to that channel, and the channel is left open for further writing at the end of the operation. The Postscript is created in Encapsulated Postscript form using version 3.0 of the Document Structuring Conventions. Note: by default Postscript is only generated for information that appears in the canvas's window on the screen. If the canvas is freshly created it may still have its initial size of 1x1 pixel so nothing will appear in the Postscript. To get around this problem either invoke the "update" command to wait for the canvas window to reach its final size, or else use the **−width** and **−height** options to specify the area of the canvas to print. The *option−value* argument pairs provide additional information to control the generation of Postscript. The following options are supported:

−colormap *varName*

VarName must be the name of an array variable that specifies a color mapping to use in the Postscript. Each element of *varName* must consist of Postscript code to set a particular color value (e.g. ''**1.0 1.0 0.0 setrgbcolor**''). When outputting color information in the Postscript, Tk checks to see if there is an element of *varName* with the same name as the color. If so, Tk uses the value of the element as the Postscript command to set the color. If this option hasn't been specified, or if there isn't an entry in *varName* for a given color, then Tk uses the red, green, and blue intensities from the X color.

−colormode *mode*

Specifies how to output color information. *Mode* must be either **color** (for full color output), **gray** (convert all colors to their gray-scale equivalents) or **mono** (convert all colors to black or white).

−file *fileName*

Specifies the name of the file in which to write the Postscript. If this option isn't specified then the Postscript is returned as the result of the command instead of being written to a file.

−fontmap *varName*

VarName must be the name of an array variable that specifies a font mapping to use in the Postscript. Each element of *varName* must consist of a Tcl list with two elements, which are the name and point size of a Postscript font. When outputting Postscript commands for a particular font, Tk checks to see if *varName* contains an element with the same name as the font. If there is such an element, then the font information contained in that element is used in the Postscript. Otherwise Tk attempts to guess what Postscript font to use. Tk's guesses generally only work for well-known fonts such as Times and Helvetica and Courier, and only if the X font name does not omit any dashes up through the point size. For example, **−∗−Courier−Bold−R−Normal−−∗−120−∗** will work but **∗Courier−Bold−R−Normal∗120∗** will not; Tk needs the dashes to parse the font name).

−height *size*

Specifies the height of the area of the canvas to print. Defaults to the height of the canvas window.

−pageanchor *anchor*

Specifies which point of the printed area of the canvas should appear over the positioning

point on the page (which is given by the **−pagex** and **−pagey** options). For example, **−pageanchor n** means that the top center of the area of the canvas being printed (as it appears in the canvas window) should be over the positioning point. Defaults to **center**.

−pageheight *size*
Specifies that the Postscript should be scaled in both x and y so that the printed area is *size* high on the Postscript page. *Size* consists of a floating-point number followed by **c** for centimeters, **i** for inches, **m** for millimeters, or **p** or nothing for printer's points (1/72 inch). Defaults to the height of the printed area on the screen. If both **−pageheight** and **−pagewidth** are specified then the scale factor from **−pagewidth** is used (non-uniform scaling is not implemented).

−pagewidth *size*
Specifies that the Postscript should be scaled in both x and y so that the printed area is *size* wide on the Postscript page. *Size* has the same form as for **−pageheight**. Defaults to the width of the printed area on the screen. If both **−pageheight** and **−pagewidth** are specified then the scale factor from **−pagewidth** is used (non-uniform scaling is not implemented).

−pagex *position*
Position gives the x-coordinate of the positioning point on the Postscript page, using any of the forms allowed for **−pageheight**. Used in conjunction with the **−pagey** and **−pageanchor** options to determine where the printed area appears on the Postscript page. Defaults to the center of the page.

−pagey *position*
Position gives the y-coordinate of the positioning point on the Postscript page, using any of the forms allowed for **−pageheight**. Used in conjunction with the **−pagex** and **−pageanchor** options to determine where the printed area appears on the Postscript page. Defaults to the center of the page.

−rotate *boolean*
Boolean specifies whether the printed area is to be rotated 90 degrees. In non-rotated output the x-axis of the printed area runs along the short dimension of the page (''portrait'' orientation); in rotated output the x-axis runs along the long dimension of the page (''landscape'' orientation). Defaults to non-rotated.

−width *size*
Specifies the width of the area of the canvas to print. Defaults to the width of the canvas window.

−x *position*
Specifies the x-coordinate of the left edge of the area of the canvas that is to be printed, in canvas coordinates, not window coordinates. Defaults to the coordinate of the left edge of the window.

−y *position*
Specifies the y-coordinate of the top edge of the area of the canvas that is to be printed, in canvas coordinates, not window coordinates. Defaults to the coordinate of the top edge of the window.

pathName **raise** *tagOrId* ?*aboveThis*?
Move all of the items given by *tagOrId* to a new position in the display list just after the item given by *aboveThis*. If *tagOrId* refers to more than one item then all are moved but the relative order of the moved items will not be changed. *AboveThis* is a tag or id; if it refers to more than one item then the last (topmost) of these items in the display list is used as the destination location for the moved items. Note: this command has no effect on window items. Window items always obscure

other item types, and the stacking order of window items is determined by the **raise** and **lower** commands, not the **raise** and **lower** widget commands for canvases. This command returns an empty string.

pathName **scale** *tagOrId xOrigin yOrigin xScale yScale*

Rescale all of the items given by *tagOrId* in canvas coordinate space. *XOrigin* and *yOrigin* identify the origin for the scaling operation and *xScale* and *yScale* identify the scale factors for x- and y-coordinates, respectively (a scale factor of 1.0 implies no change to that coordinate). For each of the points defining each item, the x-coordinate is adjusted to change the distance from *xOrigin* by a factor of *xScale*. Similarly, each y-coordinate is adjusted to change the distance from *yOrigin* by a factor of *yScale*. This command returns an empty string.

pathName **scan** *option args*

This command is used to implement scanning on canvases. It has two forms, depending on *option*:

pathName **scan mark** *x y*

Records *x* and *y* and the canvas's current view; used in conjunction with later **scan dragto** commands. Typically this command is associated with a mouse button press in the widget and *x* and *y* are the coordinates of the mouse. It returns an empty string.

pathName **scan dragto** *x y*.

This command computes the difference between its *x* and *y* arguments (which are typically mouse coordinates) and the *x* and *y* arguments to the last **scan mark** command for the widget. It then adjusts the view by 10 times the difference in coordinates. This command is typically associated with mouse motion events in the widget, to produce the effect of dragging the canvas at high speed through its window. The return value is an empty string.

pathName **select** *option* ?*tagOrId arg*?

Manipulates the selection in one of several ways, depending on *option*. The command may take any of the forms described below. In all of the descriptions below, *tagOrId* must refer to an item that supports indexing and selection; if it refers to multiple items then the first of these that supports indexing and the selection is used. *Index* gives a textual description of a position within *tagOrId*, as described in INDICES above.

pathName **select adjust** *tagOrId index*

Locate the end of the selection in *tagOrId* nearest to the character given by *index*, and adjust that end of the selection to be at *index* (i.e. including but not going beyond *index*). The other end of the selection is made the anchor point for future **select to** commands. If the selection isn't currently in *tagOrId* then this command behaves the same as the **select to** widget command. Returns an empty string.

pathName **select clear**

Clear the selection if it is in this widget. If the selection isn't in this widget then the command has no effect. Returns an empty string.

pathName **select from** *tagOrId index*

Set the selection anchor point for the widget to be just before the character given by *index* in the item given by *tagOrId*. This command doesn't change the selection; it just sets the fixed end of the selection for future **select to** commands. Returns an empty string.

pathName **select item**

Returns the id of the selected item, if the selection is in an item in this canvas. If the selection is not in this canvas then an empty string is returned.

pathName **select to** *tagOrId index*

Set the selection to consist of those characters of *tagOrId* between the selection anchor point and *index*. The new selection will include the character given by *index*; it will

include the character given by the anchor point only if *index* is greater than or equal to the anchor point. The anchor point is determined by the most recent **select adjust** or **select from** command for this widget. If the selection anchor point for the widget isn't currently in *tagOrId*, then it is set to the same character given by *index*. Returns an empty string.

pathName **type** *tagOrId*

Returns the type of the item given by *tagOrId*, such as **rectangle** or **text**. If *tagOrId* refers to more than one item, then the type of the first item in the display list is returned. If *tagOrId* doesn't refer to any items at all then an empty string is returned.

pathName **xview** ?*args*?

This command is used to query and change the horizontal position of the information displayed in the canvas's window. It can take any of the following forms:

pathName **xview**

Returns a list containing two elements. Each element is a real fraction between 0 and 1; together they describe the horizontal span that is visible in the window. For example, if the first element is .2 and the second element is .6, 20% of the canvas's area (as defined by the **−scrollregion** option) is off-screen to the left, the middle 40% is visible in the window, and 40% of the canvas is off-screen to the right. These are the same values passed to scrollbars via the **−xscrollcommand** option.

pathName **xview moveto** *fraction*

Adjusts the view in the window so that *fraction* of the total width of the canvas is off-screen to the left. *Fraction* must be a fraction between 0 and 1.

pathName **xview scroll** *number what*

This command shifts the view in the window left or right according to *number* and *what*. *Number* must be an integer. *What* must be either **units** or **pages** or an abbreviation of one of these. If *what* is **units**, the view adjusts left or right in units of the **xScrollIncrement** option, if it is greater than zero, or in units of one-tenth the window's width otherwise. If *what is* **pages** then the view adjusts in units of nine-tenths the window's width. If *number* is negative then information farther to the left becomes visible; if it is positive then information farther to the right becomes visible.

pathName **yview** *?args*?

This command is used to query and change the vertical position of the information displayed in the canvas's window. It can take any of the following forms:

pathName **yview**

Returns a list containing two elements. Each element is a real fraction between 0 and 1; together they describe the vertical span that is visible in the window. For example, if the first element is .6 and the second element is 1.0, the lowest 40% of the canvas's area (as defined by the **−scrollregion** option) is visible in the window. These are the same values passed to scrollbars via the **−yscrollcommand** option.

pathName **yview moveto** *fraction*

Adjusts the view in the window so that *fraction* of the canvas's area is off-screen to the top. *Fraction* is a fraction between 0 and 1.

pathName **yview scroll** *number what*

This command adjusts the view in the window up or down according to *number* and *what*. *Number* must be an integer. *What* must be either **units** or **pages**. If *what* is **units**, the view adjusts up or down in units of the **yScrollIncrement** option, if it is greater than zero, or in units of one-tenth the window's height otherwise. If *what* is **pages** then the view adjusts in units of nine-tenths the window's height. If *number* is negative then

higher information becomes visible; if it is positive then lower information becomes visible.

OVERVIEW OF ITEM TYPES

The sections below describe the various types of items supported by canvas widgets. Each item type is characterized by two things: first, the form of the **create** command used to create instances of the type; and second, a set of configuration options for items of that type, which may be used in the **create** and **itemconfigure** widget commands. Most items don't support indexing or selection or the commands related to them, such as **index** and **insert**. Where items do support these facilities, it is noted explicitly in the descriptions below (at present, only text items provide this support).

ARC ITEMS

Items of type **arc** appear on the display as arc-shaped regions. An arc is a section of an oval delimited by two angles (specified by the **−start** and **−extent** options) and displayed in one of several ways (specified by the **−style** option). Arcs are created with widget commands of the following form:

pathName **create arc** *x1 y1 x2 y2* ?*option value option value ...*?

The arguments *x1*, *y1*, *x2*, and *y2* give the coordinates of two diagonally opposite corners of a rectangular region enclosing the oval that defines the arc. After the coordinates there may be any number of *option−value* pairs, each of which sets one of the configuration options for the item. These same *option−value* pairs may be used in **itemconfigure** widget commands to change the item's configuration. The following options are supported for arcs:

−extent *degrees*

Specifies the size of the angular range occupied by the arc. The arc's range extends for *degrees* degrees counter-clockwise from the starting angle given by the **−start** option. *Degrees* may be negative. If it is greater than 360 or less than -360, then *degrees* modulo 360 is used as the extent.

−fill *color*

Fill the region of the arc with *color*. *Color* may have any of the forms accepted by **Tk_GetColor**. If *color* is an empty string (the default), then then the arc will not be filled.

−outline *color*

Color specifies a color to use for drawing the arc's outline; it may have any of the forms accepted by **Tk_GetColor**. This option defaults to **black**. If *color* is specified as an empty string then no outline is drawn for the arc.

−outlinestipple *bitmap*

Indicates that the outline for the arc should be drawn with a stipple pattern; *bitmap* specifies the stipple pattern to use, in any of the forms accepted by **Tk_GetBitmap**. If the **−outline** option hasn't been specified then this option has no effect. If *bitmap* is an empty string (the default), then the outline is drawn in a solid fashion.

−start *degrees*

Specifies the beginning of the angular range occupied by the arc. *Degrees* is given in units of degrees measured counter-clockwise from the 3-o'clock position; it may be either positive or negative.

−stipple *bitmap*

Indicates that the arc should be filled in a stipple pattern; *bitmap* specifies the stipple pattern to use, in any of the forms accepted by **Tk_GetBitmap**. If the **−fill** option hasn't been specified then this option has no effect. If *bitmap* is an empty string (the default), then filling is done in a solid fashion.

−style *type*

Specifies how to draw the arc. If *type* is **pieslice** (the default) then the arc's region is defined by a

section of the oval's perimeter plus two line segments, one between the center of the oval and each end of the perimeter section. If *type* is **chord** then the arc's region is defined by a section of the oval's perimeter plus a single line segment connecting the two end points of the perimeter section. If *type* is **arc** then the arc's region consists of a section of the perimeter alone. In this last case the **−fill** option is ignored.

−tags *tagList*
Specifies a set of tags to apply to the item. *TagList* consists of a list of tag names, which replace any existing tags for the item. *TagList* may be an empty list.

−width *outlineWidth*
Specifies the width of the outline to be drawn around the arc's region, in any of the forms described in the COORDINATES section above. If the **−outline** option has been specified as an empty string then this option has no effect. Wide outlines will be drawn centered on the edges of the arc's region. This option defaults to 1.0.

BITMAP ITEMS

Items of type **bitmap** appear on the display as images with two colors, foreground and background. Bitmaps are created with widget commands of the following form:

pathName **create bitmap** *x y* ?*option value option value ...*?

The arguments *x* and *y* specify the coordinates of a point used to position the bitmap on the display (see the **−anchor** option below for more information on how bitmaps are displayed). After the coordinates there may be any number of *option−value* pairs, each of which sets one of the configuration options for the item. These same *option−value* pairs may be used in **itemconfigure** widget commands to change the item's configuration. The following options are supported for bitmaps:

−anchor *anchorPos*
AnchorPos tells how to position the bitmap relative to the positioning point for the item; it may have any of the forms accepted by **Tk_GetAnchor**. For example, if *anchorPos* is **center** then the bitmap is centered on the point; if *anchorPos* is **n** then the bitmap will be drawn so that its top center point is at the positioning point. This option defaults to **center**.

−background *color*
Specifies a color to use for each of the bitmap pixels whose value is 0. *Color* may have any of the forms accepted by **Tk_GetColor**. If this option isn't specified, or if it is specified as an empty string, then nothing is displayed where the bitmap pixels are 0; this produces a transparent effect.

−bitmap *bitmap*
Specifies the bitmap to display in the item. *Bitmap* may have any of the forms accepted by **Tk_GetBitmap**.

−foreground *color*
Specifies a color to use for each of the bitmap pixels whose value is 1. *Color* may have any of the forms accepted by **Tk_GetColor** and defaults to **black**.

−tags *tagList*
Specifies a set of tags to apply to the item. *TagList* consists of a list of tag names, which replace any existing tags for the item. *TagList* may be an empty list.

IMAGE ITEMS

Items of type **image** are used to display images on a canvas. Images are created with widget commands of the following form:

pathName **create image** *x y* ?*option value option value ...*?

The arguments *x* and *y* specify the coordinates of a point used to position the image on the display (see the **−anchor** option below for more information). After the coordinates there may be any number of

option−value pairs, each of which sets one of the configuration options for the item. These same *option−value* pairs may be used in **itemconfigure** widget commands to change the item's configuration. The following options are supported for images:

−anchor *anchorPos*
AnchorPos tells how to position the image relative to the positioning point for the item; it may have any of the forms accepted by **Tk_GetAnchor**. For example, if *anchorPos* is **center** then the image is centered on the point; if *anchorPos* is **n** then the image will be drawn so that its top center point is at the positioning point. This option defaults to **center**.

−image *name*
Specifies the name of the image to display in the item. This image must have been created previously with the **image create** command.

−tags *tagList*
Specifies a set of tags to apply to the item. *TagList* consists of a list of tag names, which replace any existing tags for the item; it may be an empty list.

LINE ITEMS

Items of type **line** appear on the display as one or more connected line segments or curves. Lines are created with widget commands of the following form:

pathName **create line** *x1 y1... xn yn* ?*option value option value ...*?

The arguments *x1* through *yn* give the coordinates for a series of two or more points that describe a series of connected line segments. After the coordinates there may be any number of *option−value* pairs, each of which sets one of the configuration options for the item. These same *option−value* pairs may be used in **itemconfigure** widget commands to change the item's configuration. The following options are supported for lines:

−arrow *where*
Indicates whether or not arrowheads are to be drawn at one or both ends of the line. *Where* must have one of the values **none** (for no arrowheads), **first** (for an arrowhead at the first point of the line), **last** (for an arrowhead at the last point of the line), or **both** (for arrowheads at both ends). This option defaults to **none**.

−arrowshape *shape*
This option indicates how to draw arrowheads. The *shape* argument must be a list with three elements, each specifying a distance in any of the forms described in the COORDINATES section above. The first element of the list gives the distance along the line from the neck of the arrowhead to its tip. The second element gives the distance along the line from the trailing points of the arrowhead to the tip, and the third element gives the distance from the outside edge of the line to the trailing points. If this option isn't specified then Tk picks a ''reasonable'' shape.

−capstyle *style*
Specifies the ways in which caps are to be drawn at the endpoints of the line. *Style* may have any of the forms accepted by **Tk_GetCapStyle** (**butt**, **projecting**, or **round**). If this option isn't specified then it defaults to **butt**. Where arrowheads are drawn the cap style is ignored.

−fill *color*
Color specifies a color to use for drawing the line; it may have any of the forms acceptable to **Tk_GetColor**. It may also be an empty string, in which case the line will be transparent. This option defaults to **black**.

−joinstyle *style*
Specifies the ways in which joints are to be drawn at the vertices of the line. *Style* may have any of the forms accepted by **Tk_GetCapStyle** (**bevel**, **miter**, or **round**). If this option isn't specified then it defaults to **miter**. If the line only contains two points then this option is irrelevant.

−smooth *boolean*

Boolean must have one of the forms accepted by **Tk_GetBoolean**. It indicates whether or not the line should be drawn as a curve. If so, the line is rendered as a set of parabolic splines: one spline is drawn for the first and second line segments, one for the second and third, and so on. Straight-line segments can be generated within a curve by duplicating the end-points of the desired line segment.

−splinesteps *number*

Specifies the degree of smoothness desired for curves: each spline will be approximated with *number* line segments. This option is ignored unless the **−smooth** option is true.

−stipple *bitmap*

Indicates that the line should be filled in a stipple pattern; *bitmap* specifies the stipple pattern to use, in any of the forms accepted by **Tk_GetBitmap**. If *bitmap* is an empty string (the default), then filling is done in a solid fashion.

−tags *tagList*

Specifies a set of tags to apply to the item. *TagList* consists of a list of tag names, which replace any existing tags for the item. *TagList* may be an empty list.

−width *lineWidth*

LineWidth specifies the width of the line, in any of the forms described in the COORDINATES section above. Wide lines will be drawn centered on the path specified by the points. If this option isn't specified then it defaults to 1.0.

OVAL ITEMS

Items of type **oval** appear as circular or oval regions on the display. Each oval may have an outline, a fill, or both. Ovals are created with widget commands of the following form:

pathName **create oval** *x1 y1 x2 y2* ?*option value option value ...*?

The arguments *x1*, *y1*, *x2*, and *y2* give the coordinates of two diagonally opposite corners of a rectangular region enclosing the oval. The oval will include the top and left edges of the rectangle not the lower or right edges. If the region is square then the resulting oval is circular; otherwise it is elongated in shape. After the coordinates there may be any number of *option−value* pairs, each of which sets one of the configuration options for the item. These same *option−value* pairs may be used in **itemconfigure** widget commands to change the item's configuration. The following options are supported for ovals:

−fill *color*

Fill the area of the oval with *color*. *Color* may have any of the forms accepted by **Tk_GetColor**. If *color* is an empty string (the default), then then the oval will not be filled.

−outline *color*

Color specifies a color to use for drawing the oval's outline; it may have any of the forms accepted by **Tk_GetColor**. This option defaults to **black**. If *color* is an empty string then no outline will be drawn for the oval.

−stipple *bitmap*

Indicates that the oval should be filled in a stipple pattern; *bitmap* specifies the stipple pattern to use, in any of the forms accepted by **Tk_GetBitmap**. If the **−fill** option hasn't been specified then this option has no effect. If *bitmap* is an empty string (the default), then filling is done in a solid fashion.

−tags *tagList*

Specifies a set of tags to apply to the item. *TagList* consists of a list of tag names, which replace any existing tags for the item. *TagList* may be an empty list.

−width *outlineWidth*

outlineWidth specifies the width of the outline to be drawn around the oval, in any of the forms

described in the COORDINATES section above. If the **−outline** option hasn't been specified then this option has no effect. Wide outlines are drawn centered on the oval path defined by *x1*, *y1*, *x2*, and *y2*. This option defaults to 1.0.

POLYGON ITEMS

Items of type **polygon** appear as polygonal or curved filled regions on the display. Polygons are created with widget commands of the following form:

pathName **create polygon** *x1 y1 ... xn yn* ?*option value option value ...*?

The arguments *x1* through *yn* specify the coordinates for three or more points that define a closed polygon. The first and last points may be the same; whether they are or not, Tk will draw the polygon as a closed polygon. After the coordinates there may be any number of *option−value* pairs, each of which sets one of the configuration options for the item. These same *option−value* pairs may be used in **itemconfigure** widget commands to change the item's configuration. The following options are supported for polygons:

−fill *color*

Color specifies a color to use for filling the area of the polygon; it may have any of the forms acceptable to **Tk_GetColor**. If *color* is an empty string then the polygon will be transparent. This option defaults to **black**.

−outline *color*

Color specifies a color to use for drawing the polygon's outline; it may have any of the forms accepted by **Tk_GetColor**. If *color* is an empty string then no outline will be drawn for the polygon. This option defaults to empty (no outline).

−smooth *boolean*

Boolean must have one of the forms accepted by **Tk_GetBoolean** It indicates whether or not the polygon should be drawn with a curved perimeter. If so, the outline of the polygon becomes a set of parabolic splines, one spline for the first and second line segments, one for the second and third, and so on. Straight-line segments can be generated in a smoothed polygon by duplicating the endpoints of the desired line segment.

−splinesteps *number*

Specifies the degree of smoothness desired for curves: each spline will be approximated with *number* line segments. This option is ignored unless the **−smooth** option is true.

−stipple *bitmap*

Indicates that the polygon should be filled in a stipple pattern; *bitmap* specifies the stipple pattern to use, in any of the forms accepted by **Tk_GetBitmap**. If *bitmap* is an empty string (the default), then filling is done in a solid fashion.

−tags *tagList*

Specifies a set of tags to apply to the item. *TagList* consists of a list of tag names, which replace any existing tags for the item. *TagList* may be an empty list.

−width *outlineWidth*

OutlineWidth specifies the width of the outline to be drawn around the polygon, in any of the forms described in the COORDINATES section above. If the **−outline** option hasn't been specified then this option has no effect. This option defaults to 1.0.

Polygon items are different from other items such as rectangles, ovals and arcs in that interior points are considered to be ''inside'' a polygon (e.g. for purposes of the **find closest** and **find overlapping** widget commands) even if it is not filled. For most other item types, an interior point is considered to be inside the item only if the item is filled or if it has neither a fill nor an outline. If you would like an unfilled polygon whose interior points are not considered to be inside the polygon, use a line item instead.

RECTANGLE ITEMS

Items of type **rectangle** appear as rectangular regions on the display. Each rectangle may have an outline, a fill, or both. Rectangles are created with widget commands of the following form:

pathName **create rectangle** *x1 y1 x2 y2* ?*option value option value ...*?

The arguments *x1*, *y1*, *x2*, and *y2* give the coordinates of two diagonally opposite corners of the rectangle (the rectangle will include its upper and left edges but not its lower or right edges). After the coordinates there may be any number of *option−value* pairs, each of which sets one of the configuration options for the item. These same *option−value* pairs may be used in **itemconfigure** widget commands to change the item's configuration. The following options are supported for rectangles:

−fill *color*

Fill the area of the rectangle with *color*, which may be specified in any of the forms accepted by **Tk_GetColor**. If *color* is an empty string (the default), then the rectangle will not be filled.

−outline *color*

Draw an outline around the edge of the rectangle in *color*. *Color* may have any of the forms accepted by **Tk_GetColor**. This option defaults to **black**. If *color* is an empty string then no outline will be drawn for the rectangle.

−stipple *bitmap*

Indicates that the rectangle should be filled in a stipple pattern; *bitmap* specifies the stipple pattern to use, in any of the forms accepted by **Tk_GetBitmap**. If the **−fill** option hasn't been specified then this option has no effect. If *bitmap* is an empty string (the default), then filling is done in a solid fashion.

−tags *tagList*

Specifies a set of tags to apply to the item. *TagList* consists of a list of tag names, which replace any existing tags for the item. *TagList* may be an empty list.

−width *outlineWidth*

OutlineWidth specifies the width of the outline to be drawn around the rectangle, in any of the forms described in the COORDINATES section above. If the **−outline** option hasn't been specified then this option has no effect. Wide outlines are drawn centered on the rectangular path defined by *x1*, *y1*, *x2*, and *y2*. This option defaults to 1.0.

TEXT ITEMS

A text item displays a string of characters on the screen in one or more lines. Text items support indexing and selection, along with the following text-related canvas widget commands: **dchars**, **focus**, **icursor**, **index**, **insert**, **select**. Text items are created with widget commands of the following form:

pathName **create text** *x y* ?*option value option value ...*?

The arguments *x* and *y* specify the coordinates of a point used to position the text on the display (see the options below for more information on how text is displayed). After the coordinates there may be any number of *option−value* pairs, each of which sets one of the configuration options for the item. These same *option−value* pairs may be used in **itemconfigure** widget commands to change the item's configuration. The following options are supported for text items:

−anchor *anchorPos*

AnchorPos tells how to position the text relative to the positioning point for the text; it may have any of the forms accepted by **Tk_GetAnchor**. For example, if *anchorPos* is **center** then the text is centered on the point; if *anchorPos* is **n** then the text will be drawn such that the top center point of the rectangular region occupied by the text will be at the positioning point. This option defaults to **center**.

−fill *color*

Color specifies a color to use for filling the text characters; it may have any of the forms accepted

by **Tk_GetColor**. If this option isn't specified then it defaults to **black**.

−font *fontName*
Specifies the font to use for the text item. *FontName* may be any string acceptable to **Tk_GetFontStruct**. If this option isn't specified, it defaults to a system-dependent font.

−justify *how*
Specifies how to justify the text within its bounding region. *How* must be one of the values **left**, **right**, or **center**. This option will only matter if the text is displayed as multiple lines. If the option is omitted, it defaults to **left**.

−stipple *bitmap*
Indicates that the text should be drawn in a stippled pattern rather than solid; *bitmap* specifies the stipple pattern to use, in any of the forms accepted by **Tk_GetBitmap**. If *bitmap* is an empty string (the default) then the text is drawn in a solid fashion.

−tags *tagList*
Specifies a set of tags to apply to the item. *TagList* consists of a list of tag names, which replace any existing tags for the item. *TagList* may be an empty list.

−text *string*
String specifies the characters to be displayed in the text item. Newline characters cause line breaks. The characters in the item may also be changed with the **insert** and **delete** widget commands. This option defaults to an empty string.

−width *lineLength*
Specifies a maximum line length for the text, in any of the forms described in the COORDINATES section above. If this option is zero (the default) the text is broken into lines only at newline characters. However, if this option is non-zero then any line that would be longer than *lineLength* is broken just before a space character to make the line shorter than *lineLength*; the space character is treated as if it were a newline character.

WINDOW ITEMS

Items of type **window** cause a particular window to be displayed at a given position on the canvas. Window items are created with widget commands of the following form:

pathName **create window** *x y* ?*option value option value ...*?

The arguments *x* and *y* specify the coordinates of a point used to position the window on the display (see the **−anchor** option below for more information on how bitmaps are displayed). After the coordinates there may be any number of *option−value* pairs, each of which sets one of the configuration options for the item. These same *option−value* pairs may be used in **itemconfigure** widget commands to change the item's configuration. The following options are supported for window items:

−anchor *anchorPos*
AnchorPos tells how to position the window relative to the positioning point for the item; it may have any of the forms accepted by **Tk_GetAnchor**. For example, if *anchorPos* is **center** then the window is centered on the point; if *anchorPos* is **n** then the window will be drawn so that its top center point is at the positioning point. This option defaults to **center**.

−height *pixels*
Specifies the height to assign to the item's window. *Pixels* may have any of the forms described in the COORDINATES section above. If this option isn't specified, or if it is specified as an empty string, then the window is given whatever height it requests internally.

−tags *tagList*
Specifies a set of tags to apply to the item. *TagList* consists of a list of tag names, which replace any existing tags for the item. *TagList* may be an empty list.

−width *pixels*
Specifies the width to assign to the item's window. *Pixels* may have any of the forms described in the COORDINATES section above. If this option isn't specified, or if it is specified as an empty string, then the window is given whatever width it requests internally.

−window *pathName*
Specifies the window to associate with this item. The window specified by *pathName* must either be a child of the canvas widget or a child of some ancestor of the canvas widget. *PathName* may not refer to a top-level window.

Note: due to restrictions in the ways that windows are managed, it is not possible to draw other graphical items (such as lines and images) on top of window items. A window item always obscures any graphics that overlap it, regardless of their order in the display list.

APPLICATION-DEFINED ITEM TYPES

It is possible for individual applications to define new item types for canvas widgets using C code. See the documentation for **Tk_CreateItemType**.

BINDINGS

In the current implementation, new canvases are not given any default behavior: you'll have to execute explicit Tcl commands to give the canvas its behavior.

CREDITS

Tk's canvas widget is a blatant ripoff of ideas from Joel Bartlett's *ezd* program. *Ezd* provides structured graphics in a Scheme environment and preceded canvases by a year or two. Its simple mechanisms for placing and animating graphical objects inspired the functions of canvases.

KEYWORDS

canvas, widget

NAME

checkbutton − Create and manipulate checkbutton widgets

SYNOPSIS

checkbutton *pathName* ?*options*?

STANDARD OPTIONS

−activebackground	**−cursor**	**−highlightthickness**	**−takefocus**
−activeforeground	**−disabledforeground**	**−image**	**−text**
−anchor	**−font**	**−justify**	**−textvariable**
−background	**−foreground**	**−padx**	**−underline**
−bitmap	**−highlightbackground**	**−pady**	**−wraplength**
−borderwidth	**−highlightcolor**	**−relief**	

See the **options** manual entry for details on the standard options.

WIDGET-SPECIFIC OPTIONS

Command-Line Name: **−command**
Database Name: **command**
Database Class: **Command**

Specifies a Tcl command to associate with the button. This command is typically invoked when mouse button 1 is released over the button window. The button's global variable (**−variable** option) will be updated before the command is invoked.

Command-Line Name: **−height**
Database Name: **height**
Database Class: **Height**

Specifies a desired height for the button. If an image or bitmap is being displayed in the button then the value is in screen units (i.e. any of the forms acceptable to **Tk_GetPixels**); for text it is in lines of text. If this option isn't specified, the button's desired height is computed from the size of the image or bitmap or text being displayed in it.

Command-Line Name: **−indicatoron**
Database Name: **indicatorOn**
Database Class: **IndicatorOn**

Specifies whether or not the indicator should be drawn. Must be a proper boolean value. If false, the **relief** option is ignored and the widget's relief is always sunken if the widget is selected and raised otherwise.

Command-Line Name: **−offvalue**
Database Name: **offValue**
Database Class: **Value**

Specifies value to store in the button's associated variable whenever this button is deselected. Defaults to ''0''.

Command-Line Name: **−onvalue**
Database Name: **onValue**
Database Class: **Value**

Specifies value to store in the button's associated variable whenever this button is selected. Defaults to ''1''.

Command-Line Name: **−selectcolor**
Database Name: **selectColor**
Database Class: **Background**

Specifies a background color to use when the button is selected. If **indicatorOn** is true then the

color applies to the indicator. Under Windows, this color is used as the background for the indicator regardless of the select state. If **indicatorOn** is false, this color is used as the background for the entire widget, in place of **background** or **activeBackground**, whenever the widget is selected. If specified as an empty string then no special color is used for displaying when the widget is selected.

Command-Line Name: **−selectimage**
Database Name: **selectImage**
Database Class: **SelectImage**

Specifies an image to display (in place of the **image** option) when the checkbutton is selected. This option is ignored unless the **image** option has been specified.

Command-Line Name: **−state**
Database Name: **state**
Database Class: **State**

Specifies one of three states for the checkbutton: **normal**, **active**, or **disabled**. In normal state the checkbutton is displayed using the **foreground** and **background** options. The active state is typically used when the pointer is over the checkbutton. In active state the checkbutton is displayed using the **activeForeground** and **activeBackground** options. Disabled state means that the checkbutton should be insensitive: the default bindings will refuse to activate the widget and will ignore mouse button presses. In this state the **disabledForeground** and **background** options determine how the checkbutton is displayed.

Command-Line Name: **−variable**
Database Name: **variable**
Database Class: **Variable**

Specifies name of global variable to set to indicate whether or not this button is selected. Defaults to the name of the button within its parent (i.e. the last element of the button window's path name).

Command-Line Name: **−width**
Database Name: **width**
Database Class: **Width**

Specifies a desired width for the button. If an image or bitmap is being displayed in the button then the value is in screen units (i.e. any of the forms acceptable to **Tk_GetPixels**); for text it is in characters. If this option isn't specified, the button's desired width is computed from the size of the image or bitmap or text being displayed in it.

DESCRIPTION

The **checkbutton** command creates a new window (given by the *pathName* argument) and makes it into a checkbutton widget. Additional options, described above, may be specified on the command line or in the option database to configure aspects of the checkbutton such as its colors, font, text, and initial relief. The **checkbutton** command returns its *pathName* argument. At the time this command is invoked, there must not exist a window named *pathName*, but *pathName*'s parent must exist.

A checkbutton is a widget that displays a textual string, bitmap or image and a square called an *indicator*. If text is displayed, it must all be in a single font, but it can occupy multiple lines on the screen (if it contains newlines or if wrapping occurs because of the **wrapLength** option) and one of the characters may optionally be underlined using the **underline** option. A checkbutton has all of the behavior of a simple button, including the following: it can display itself in either of three different ways, according to the **state** option; it can be made to appear raised, sunken, or flat; it can be made to flash; and it invokes a Tcl command whenever mouse button 1 is clicked over the checkbutton.

In addition, checkbuttons can be *selected*. If a checkbutton is selected then the indicator is normally drawn with a selected appearance, and a Tcl variable associated with the checkbutton is set to a particular value (normally 1). Under Unix, the indicator is drawn with a sunken relief and a special color. Under Windows, the indicator is drawn with a check mark inside. If the checkbutton is not selected, then the indicator is drawn with a deselected appearance, and the associated variable is set to a different value (typically 0). Under Unix, the indicator is drawn with a raised relief and no special color. Under Windows, the indicator is drawn without a check mark inside. By default, the name of the variable associated with a checkbutton is the same as the *name* used to create the checkbutton. The variable name, and the ''on'' and ''off'' values stored in it, may be modified with options on the command line or in the option database. Configuration options may also be used to modify the way the indicator is displayed (or whether it is displayed at all). By default a checkbutton is configured to select and deselect itself on alternate button clicks. In addition, each checkbutton monitors its associated variable and automatically selects and deselects itself when the variables value changes to and from the button's ''on'' value.

WIDGET COMMAND

The **checkbutton** command creates a new Tcl command whose name is *pathName*. This command may be used to invoke various operations on the widget. It has the following general form:

pathName option ?*arg arg ...*?

Option and the *arg*s determine the exact behavior of the command. The following commands are possible for checkbutton widgets:

pathName **cget** *option*

Returns the current value of the configuration option given by *option*. *Option* may have any of the values accepted by the **checkbutton** command.

pathName **configure** ?*option*? ?*value option value ...*?

Query or modify the configuration options of the widget. If no *option* is specified, returns a list describing all of the available options for *pathName* (see **Tk_ConfigureInfo** for information on the format of this list). If *option* is specified with no *value*, then the command returns a list describing the one named option (this list will be identical to the corresponding sublist of the value returned if no *option* is specified). If one or more *option−value* pairs are specified, then the command modifies the given widget option(s) to have the given value(s); in this case the command returns an empty string. *Option* may have any of the values accepted by the **checkbutton** command.

pathName **deselect**

Deselects the checkbutton and sets the associated variable to its ''off'' value.

pathName **flash**

Flashes the checkbutton. This is accomplished by redisplaying the checkbutton several times, alternating between active and normal colors. At the end of the flash the checkbutton is left in the same normal/active state as when the command was invoked. This command is ignored if the checkbutton's state is **disabled**.

pathName **invoke**

Does just what would have happened if the user invoked the checkbutton with the mouse: toggle the selection state of the button and invoke the Tcl command associated with the checkbutton, if there is one. The return value is the return value from the Tcl command, or an empty string if there is no command associated with the checkbutton. This command is ignored if the checkbutton's state is **disabled**.

pathName **select**

Selects the checkbutton and sets the associated variable to its ''on'' value.

pathName **toggle**

Toggles the selection state of the button, redisplaying it and modifying its associated variable to reflect the new state.

BINDINGS

Tk automatically creates class bindings for checkbuttons that give them the following default behavior:

[1] On Unix systems, a checkbutton activates whenever the mouse passes over it and deactivates whenever the mouse leaves the checkbutton. On Mac and Windows systems, when mouse button 1 is pressed over a checkbutton, the button activates whenever the mouse pointer is inside the button, and deactivates whenever the mouse pointer leaves the button.

[2] When mouse button 1 is pressed over a checkbutton, it is invoked (its selection state toggles and the command associated with the button is invoked, if there is one).

[3] When a checkbutton has the input focus, the space key causes the checkbutton to be invoked. Under Windows, there are additional key bindings; plus (+) and equal (=) select the button, and minus (-) deselects the button.

If the checkbutton's state is **disabled** then none of the above actions occur: the checkbutton is completely non-responsive.

The behavior of checkbuttons can be changed by defining new bindings for individual widgets or by redefining the class bindings.

KEYWORDS

checkbutton, widget

NAME

tk_chooseColor − pops up a dialog box for the user to select a color.

SYNOPSIS

tk_chooseColor ?*option value ...*?

DESCRIPTION

The procedure **tk_chooseColor** pops up a dialog box for the user to select a color. The following *option−value* pairs are possible as command line arguments:

−initialcolor *color*

Specifies the color to display in the color dialog when it pops up. *color* must be in a form acceptable to the **Tk_GetColor** function.

−parent *window*

Makes *window* the logical parent of the color dialog. The color dialog is displayed on top of its parent window.

−title *titleString*

Specifies a string to display as the title of the dialog box. If this option is not specified, then a default title will be displayed.

If the user selects a color, **tk_chooseColor** will return the name of the color in a form acceptable to **Tk_GetColor**. If the user cancels the operation, both commands will return the empty string.

EXAMPLE

```
button .b −fg [tk_chooseColor −initialcolor gray −title "Choose color"]
```

KEYWORDS

color selection dialog

NAME

clipboard − Manipulate Tk clipboard

SYNOPSIS

clipboard *option* ?*arg arg ...*?

DESCRIPTION

This command provides a Tcl interface to the Tk clipboard, which stores data for later retrieval using the selection mechanism. In order to copy data into the clipboard, **clipboard clear** must be called, followed by a sequence of one or more calls to **clipboard append**. To ensure that the clipboard is updated atomically, all appends should be completed before returning to the event loop.

The first argument to **clipboard** determines the format of the rest of the arguments and the behavior of the command. The following forms are currently supported:

clipboard clear ?−**displayof** *window*?

Claims ownership of the clipboard on *window*'s display and removes any previous contents. *Window* defaults to ''.''. Returns an empty string.

clipboard append ?−**displayof** *window*? ?−**format** *format*? ?−**type** *type*? ?− −? *data*

Appends *data* to the clipboard on *window*'s display in the form given by *type* with the representation given by *format* and claims ownership of the clipboard on *window*'s display.

Type specifies the form in which the selection is to be returned (the desired ''target'' for conversion, in ICCCM terminology), and should be an atom name such as STRING or FILE_NAME; see the Inter-Client Communication Conventions Manual for complete details. *Type* defaults to STRING.

The *format* argument specifies the representation that should be used to transmit the selection to the requester (the second column of Table 2 of the ICCCM), and defaults to STRING. If *format* is STRING, the selection is transmitted as 8-bit ASCII characters. If *format* is ATOM, then the *data* is divided into fields separated by white space; each field is converted to its atom value, and the 32-bit atom value is transmitted instead of the atom name. For any other *format*, *data* is divided into fields separated by white space and each field is converted to a 32-bit integer; an array of integers is transmitted to the selection requester. Note that strings passed to **clipboard append** are concatenated before conversion, so the caller must take care to ensure appropriate spacing across string boundaries. All items appended to the clipboard with the same *type* must have the same *format*.

The *format* argument is needed only for compatibility with clipboard requesters that don't use Tk. If the Tk toolkit is being used to retrieve the CLIPBOARD selection then the value is converted back to a string at the requesting end, so *format* is irrelevant.

A − − argument may be specified to mark the end of options: the next argument will always be used as *data*. This feature may be convenient if, for example, *data* starts with a −.

KEYWORDS

clear, format, clipboard, append, selection, type

NAME

destroy − Destroy one or more windows

SYNOPSIS

destroy ?*window window* ...?

DESCRIPTION

This command deletes the windows given by the *window* arguments, plus all of their descendants. If a *window* ‘‘.’’ is deleted then the entire application will be destroyed. The *window*s are destroyed in order, and if an error occurs in destroying a window the command aborts without destroying the remaining windows. No error is returned if *window* does not exist.

KEYWORDS

application, destroy, window

NAME

tk_dialog − Create modal dialog and wait for response

SYNOPSIS

tk_dialog *window title text bitmap default string string ...*

DESCRIPTION

This procedure is part of the Tk script library. Its arguments describe a dialog box:

window Name of top-level window to use for dialog. Any existing window by this name is destroyed.

title Text to appear in the window manager's title bar for the dialog.

text Message to appear in the top portion of the dialog box.

bitmap If non-empty, specifies a bitmap to display in the top portion of the dialog, to the left of the text. If this is an empty string then no bitmap is displayed in the dialog.

default If this is an integer greater than or equal to zero, then it gives the index of the button that is to be the default button for the dialog (0 for the leftmost button, and so on). If less than zero or an empty string then there won't be any default button.

string There will be one button for each of these arguments. Each *string* specifies text to display in a button, in order from left to right.

After creating a dialog box, **tk_dialog** waits for the user to select one of the buttons either by clicking on the button with the mouse or by typing return to invoke the default button (if any). Then it returns the index of the selected button: 0 for the leftmost button, 1 for the button next to it, and so on. If the dialog's window is destroyed before the user selects one of the buttons, then -1 is returned.

While waiting for the user to respond, **tk_dialog** sets a local grab. This prevents the user from interacting with the application in any way except to invoke the dialog box.

KEYWORDS

bitmap, dialog, modal

NAME

entry − Create and manipulate entry widgets

SYNOPSIS

entry *pathName* ?*options*?

STANDARD OPTIONS

−background	**−highlightbackground**	**−insertontime**	**−selectforeground**
−borderwidth	**−highlightcolor**	**−insertwidth**	**−takefocus**
−cursor	**−highlightthickness**	**−justify**	**−textvariable**
−exportselection	**−insertbackground**	**−relief**	**−xscrollcommand**
−font	**−insertborderwidth**	**−selectbackground**	
−foreground	**−insertofftime**	**−selectborderwidth**	

See the **options** manual entry for details on the standard options.

WIDGET-SPECIFIC OPTIONS

Command-Line Name: **−show**
Database Name: **show**
Database Class: **Show**

If this option is specified, then the true contents of the entry are not displayed in the window. Instead, each character in the entry's value will be displayed as the first character in the value of this option, such as ''∗''. This is useful, for example, if the entry is to be used to enter a password. If characters in the entry are selected and copied elsewhere, the information copied will be what is displayed, not the true contents of the entry.

Command-Line Name: **−state**
Database Name: **state**
Database Class: **State**

Specifies one of two states for the entry: **normal** or **disabled**. If the entry is disabled then the value may not be changed using widget commands and no insertion cursor will be displayed, even if the input focus is in the widget.

Command-Line Name: **−width**
Database Name: **width**
Database Class: **Width**

Specifies an integer value indicating the desired width of the entry window, in average-size characters of the widget's font. If the value is less than or equal to zero, the widget picks a size just large enough to hold its current text.

DESCRIPTION

The **entry** command creates a new window (given by the *pathName* argument) and makes it into an entry widget. Additional options, described above, may be specified on the command line or in the option database to configure aspects of the entry such as its colors, font, and relief. The **entry** command returns its *pathName* argument. At the time this command is invoked, there must not exist a window named *pathName*, but *pathName*'s parent must exist.

An entry is a widget that displays a one-line text string and allows that string to be edited using widget commands described below, which are typically bound to keystrokes and mouse actions. When first created, an entry's string is empty. A portion of the entry may be selected as described below. If an entry is exporting its selection (see the **exportSelection** option), then it will observe the standard X11 protocols for handling the selection; entry selections are available as type **STRING**. Entries also observe the standard Tk rules for dealing with the input focus. When an entry has the input focus it displays an *insertion cursor*

to indicate where new characters will be inserted.

Entries are capable of displaying strings that are too long to fit entirely within the widget's window. In this case, only a portion of the string will be displayed; commands described below may be used to change the view in the window. Entries use the standard **xScrollCommand** mechanism for interacting with scrollbars (see the description of the **xScrollCommand** option for details). They also support scanning, as described below.

WIDGET COMMAND

The **entry** command creates a new Tcl command whose name is *pathName*. This command may be used to invoke various operations on the widget. It has the following general form:

pathName option ?*arg arg ...*?

Option and the *arg*s determine the exact behavior of the command.

Many of the widget commands for entries take one or more indices as arguments. An index specifies a particular character in the entry's string, in any of the following ways:

number Specifies the character as a numerical index, where 0 corresponds to the first character in the string.

anchor Indicates the anchor point for the selection, which is set with the **select from** and **select adjust** widget commands.

end Indicates the character just after the last one in the entry's string. This is equivalent to specifying a numerical index equal to the length of the entry's string.

insert Indicates the character adjacent to and immediately following the insertion cursor.

sel.first Indicates the first character in the selection. It is an error to use this form if the selection isn't in the entry window.

sel.last Indicates the character just after the last one in the selection. It is an error to use this form if the selection isn't in the entry window.

@*number* In this form, *number* is treated as an x-coordinate in the entry's window; the character spanning that x-coordinate is used. For example, "**@0**" indicates the left-most character in the window.

Abbreviations may be used for any of the forms above, e.g. "**e**" or "**sel.f**". In general, out-of-range indices are automatically rounded to the nearest legal value.

The following commands are possible for entry widgets:

pathName **bbox** *index*

Returns a list of four numbers describing the bounding box of the character given by *index*. The first two elements of the list give the x and y coordinates of the upper-left corner of the screen area covered by the character (in pixels relative to the widget) and the last two elements give the width and height of the character, in pixels. The bounding box may refer to a region outside the visible area of the window.

pathName **cget** *option*

Returns the current value of the configuration option given by *option*. *Option* may have any of the values accepted by the **entry** command.

pathName **configure** ?*option*? ?*value option value ...*?

Query or modify the configuration options of the widget. If no *option* is specified, returns a list describing all of the available options for *pathName* (see **Tk_ConfigureInfo** for information on the format of this list). If *option* is specified with no *value*, then the command returns a list describing the one named option (this list will be identical to the corresponding sublist of the value returned if no *option* is specified). If one or more *option−value* pairs are specified, then the

command modifies the given widget option(s) to have the given value(s); in this case the command returns an empty string. *Option* may have any of the values accepted by the **entry** command.

pathName **delete** *first* ?*last*?
Delete one or more elements of the entry. *First* is the index of the first character to delete, and *last* is the index of the character just after the last one to delete. If *last* isn't specified it defaults to *first*+1, i.e. a single character is deleted. This command returns an empty string.

pathName **get**
Returns the entry's string.

pathName **icursor** *index*
Arrange for the insertion cursor to be displayed just before the character given by *index*. Returns an empty string.

pathName **index** *index*
Returns the numerical index corresponding to *index*.

pathName **insert** *index string*
Insert the characters of *string* just before the character indicated by *index*. Returns an empty string.

pathName **scan** *option args*
This command is used to implement scanning on entries. It has two forms, depending on *option*:

pathName **scan mark** *x*
Records *x* and the current view in the entry window; used in conjunction with later **scan dragto** commands. Typically this command is associated with a mouse button press in the widget. It returns an empty string.

pathName **scan dragto** *x*
This command computes the difference between its *x* argument and the *x* argument to the last **scan mark** command for the widget. It then adjusts the view left or right by 10 times the difference in x-coordinates. This command is typically associated with mouse motion events in the widget, to produce the effect of dragging the entry at high speed through the window. The return value is an empty string.

pathName **selection** *option arg*
This command is used to adjust the selection within an entry. It has several forms, depending on *option*:

pathName **selection adjust** *index*
Locate the end of the selection nearest to the character given by *index*, and adjust that end of the selection to be at *index* (i.e including but not going beyond *index*). The other end of the selection is made the anchor point for future **select to** commands. If the selection isn't currently in the entry, then a new selection is created to include the characters between *index* and the most recent selection anchor point, inclusive. Returns an empty string.

pathName **selection clear**
Clear the selection if it is currently in this widget. If the selection isn't in this widget then the command has no effect. Returns an empty string.

pathName **selection from** *index*
Set the selection anchor point to just before the character given by *index*. Doesn't change the selection. Returns an empty string.

pathName **selection present**

Returns 1 if there is are characters selected in the entry, 0 if nothing is selected.

pathName **selection range** *start end*
Sets the selection to include the characters starting with the one indexed by *start* and ending with the one just before *end*. If *end* refers to the same character as *start* or an earlier one, then the entry's selection is cleared.

pathName **selection to** *index*
If *index* is before the anchor point, set the selection to the characters from *index* up to but not including the anchor point. If *index* is the same as the anchor point, do nothing. If *index* is after the anchor point, set the selection to the characters from the anchor point up to but not including *index*. The anchor point is determined by the most recent **select from** or **select adjust** command in this widget. If the selection isn't in this widget then a new selection is created using the most recent anchor point specified for the widget. Returns an empty string.

pathName **xview** *args*
This command is used to query and change the horizontal position of the text in the widget's window. It can take any of the following forms:

pathName **xview**
Returns a list containing two elements. Each element is a real fraction between 0 and 1; together they describe the horizontal span that is visible in the window. For example, if the first element is .2 and the second element is .6, 20% of the entry's text is off-screen to the left, the middle 40% is visible in the window, and 40% of the text is off-screen to the right. These are the same values passed to scrollbars via the **−xscrollcommand** option.

pathName **xview** *index*
Adjusts the view in the window so that the character given by *index* is displayed at the left edge of the window.

pathName **xview moveto** *fraction*
Adjusts the view in the window so that the character *fraction* of the way through the text appears at the left edge of the window. *Fraction* must be a fraction between 0 and 1.

pathName **xview scroll** *number what*
This command shifts the view in the window left or right according to *number* and *what*. *Number* must be an integer. *What* must be either **units** or **pages** or an abbreviation of one of these. If *what* is **units**, the view adjusts left or right by *number* average-width characters on the display; if it is **pages** then the view adjusts by *number* screenfuls. If *number* is negative then characters farther to the left become visible; if it is positive then characters farther to the right become visible.

DEFAULT BINDINGS

Tk automatically creates class bindings for entries that give them the following default behavior. In the descriptions below, ''word'' refers to a contiguous group of letters, digits, or ''_'' characters, or any single character other than these.

[1] Clicking mouse button 1 positions the insertion cursor just before the character underneath the mouse cursor, sets the input focus to this widget, and clears any selection in the widget. Dragging with mouse button 1 strokes out a selection between the insertion cursor and the character under the mouse.

[2] Double-clicking with mouse button 1 selects the word under the mouse and positions the insertion cursor at the beginning of the word. Dragging after a double click will stroke out a selection consisting of whole words.

[3] Triple-clicking with mouse button 1 selects all of the text in the entry and positions the insertion cursor before the first character.

[4] The ends of the selection can be adjusted by dragging with mouse button 1 while the Shift key is down; this will adjust the end of the selection that was nearest to the mouse cursor when button 1 was pressed. If the button is double-clicked before dragging then the selection will be adjusted in units of whole words.

[5] Clicking mouse button 1 with the Control key down will position the insertion cursor in the entry without affecting the selection.

[6] If any normal printing characters are typed in an entry, they are inserted at the point of the insertion cursor.

[7] The view in the entry can be adjusted by dragging with mouse button 2. If mouse button 2 is clicked without moving the mouse, the selection is copied into the entry at the position of the mouse cursor.

[8] If the mouse is dragged out of the entry on the left or right sides while button 1 is pressed, the entry will automatically scroll to make more text visible (if there is more text off-screen on the side where the mouse left the window).

[9] The Left and Right keys move the insertion cursor one character to the left or right; they also clear any selection in the entry and set the selection anchor. If Left or Right is typed with the Shift key down, then the insertion cursor moves and the selection is extended to include the new character. Control-Left and Control-Right move the insertion cursor by words, and Control-Shift-Left and Control-Shift-Right move the insertion cursor by words and also extend the selection. Control-b and Control-f behave the same as Left and Right, respectively. Meta-b and Meta-f behave the same as Control-Left and Control-Right, respectively.

[10] The Home key, or Control-a, will move the insertion cursor to the beginning of the entry and clear any selection in the entry. Shift-Home moves the insertion cursor to the beginning of the entry and also extends the selection to that point.

[11] The End key, or Control-e, will move the insertion cursor to the end of the entry and clear any selection in the entry. Shift-End moves the cursor to the end and extends the selection to that point.

[12] The Select key and Control-Space set the selection anchor to the position of the insertion cursor. They don’t affect the current selection. Shift-Select and Control-Shift-Space adjust the selection to the current position of the insertion cursor, selecting from the anchor to the insertion cursor if there was not any selection previously.

[13] Control-/ selects all the text in the entry.

[14] Control-\ clears any selection in the entry.

[15] The F16 key (labelled Copy on many Sun workstations) or Meta-w copies the selection in the widget to the clipboard, if there is a selection.

[16] The F20 key (labelled Cut on many Sun workstations) or Control-w copies the selection in the widget to the clipboard and deletes the selection. If there is no selection in the widget then these keys have no effect.

[17] The F18 key (labelled Paste on many Sun workstations) or Control-y inserts the contents of the clipboard at the position of the insertion cursor.

[18] The Delete key deletes the selection, if there is one in the entry. If there is no selection, it deletes the character to the right of the insertion cursor.

[19] The BackSpace key and Control-h delete the selection, if there is one in the entry. If there is no

selection, it deletes the character to the left of the insertion cursor.

[20] Control-d deletes the character to the right of the insertion cursor.

[21] Meta-d deletes the word to the right of the insertion cursor.

[22] Control-k deletes all the characters to the right of the insertion cursor.

[23] Control-t reverses the order of the two characters to the right of the insertion cursor.

If the entry is disabled using the **−state** option, then the entry's view can still be adjusted and text in the entry can still be selected, but no insertion cursor will be displayed and no text modifications will take place.

The behavior of entries can be changed by defining new bindings for individual widgets or by redefining the class bindings.

KEYWORDS

entry, widget

NAME

event − Miscellaneous event facilities: define virtual events and generate events

SYNOPSIS

event *option* ?*arg arg ...*?

DESCRIPTION

The **event** command provides several facilities for dealing with window system events, such as defining virtual events and synthesizing events. The command has several different forms, determined by the first argument. The following forms are currently supported:

event add <<*virtual*>> *sequence* ?*sequence ...*?

Associates the virtual event *virtual* with the physical event sequence(s) given by the *sequence* arguments, so that the virtual event will trigger whenever any one of the *sequence*s occurs. *Virtual* may be any string value and *sequence* may have any of the values allowed for the *sequence* argument to the **bind** command. If *virtual* is already defined, the new physical event sequences add to the existing sequences for the event.

event delete <<*virtual*>> ?*sequence sequence ...*?

Deletes each of the *sequence*s from those associated with the virtual event given by *virtual*. *Virtual* may be any string value and *sequence* may have any of the values allowed for the *sequence* argument to the **bind** command. Any *sequence*s not currently associated with *virtual* are ignored. If no *sequence* argument is provided, all physical event sequences are removed for *virtual*, so that the virtual event will not trigger anymore.

event generate *window event* ?*option value option value ...*?

Generates a window event and arranges for it to be processed just as if it had come from the window system. *Window* gives the path name of the window for which the event will be generated; it may also be an identifier (such as returned by **winfo id**) as long as it is for a window in the current application. *Event* provides a basic description of the event, such as **<Shift-Button-2>** or **<<Paste>>**. *Event* may have any of the forms allowed for the *sequence* argument of the **bind** command except that it must consist of a single event pattern, not a sequence. *Option-value* pairs may be used to specify additional attributes of the event, such as the x and y mouse position; see EVENT FIELDS below. If the **−when** option is not specified, the event is processed immediately: all of the handlers for the event will complete before the **event generate** command returns. If the **−when** option is specified then it determines when the event is processed.

event info ?<<*virtual*>>?

Returns information about virtual events. If the <<*virtual*>> argument is omitted, the return value is a list of all the virtual events that are currently defined. If <<*virtual*>> is specified then the return value is a list whose elements are the physical event sequences currently defined for the given virtual event; if the virtual event is not defined then an empty string is returned.

EVENT FIELDS

The following options are supported for the **event generate** command. These correspond to the ‘‘%’’ expansions allowed in binding scripts for the **bind** command.

−above *window*

Window specifies the *above* field for the event, either as a window path name or as an integer window id. Valid for **Configure** events. Corresponds to the **%a** substitution for binding scripts.

−borderwidth *size*

Size must be a screen distance; it specifies the *border_width* field for the event. Valid for **Configure** events. Corresponds to the **%B** substitution for binding scripts.

−button *number*
Number must be an integer; it specifies the *detail* field for a **ButtonPress** or **ButtonRelease** event, overriding any button number provided in the base *event* argument. Corresponds to the **%b** substitution for binding scripts.

−count *number*
Number must be an integer; it specifies the *count* field for the event. Valid for **Expose** events. Corresponds to the **%c** substitution for binding scripts.

−detail *detail*
Detail specifies the *detail* field for the event and must be one of the following:

NotifyAncestor	**NotifyNonlinearVirtual**
NotifyDetailNone	**NotifyPointer**
NotifyInferior	**NotifyPointerRoot**
NotifyNonlinear	**NotifyVirtual**

Valid for **Enter**, **Leave**, **FocusIn** and **FocusOut** events. Corresponds to the **%d** substitution for binding scripts.

−focus *boolean*
Boolean must be a boolean value; it specifies the *focus* field for the event. Valid for **Enter** and **Leave** events. Corresponds to the **%f** substitution for binding scripts.

−height *size*
Size must be a screen distance; it specifies the *height* field for the event. Valid for **Configure** events. Corresponds to the **%h** substitution for binding scripts.

−keycode *number*
Number must be an integer; it specifies the *keycode* field for the event. Valid for **KeyPress** and **KeyRelease** events. Corresponds to the **%k** substitution for binding scripts.

−keysym *name*
Name must be the name of a valid keysym, such as **g**, **space**, or **Return**; its corresponding keycode value is used as the *keycode* field for event, overriding any detail specified in the base *event* argument. Valid for **KeyPress** and **KeyRelease** events. Corresponds to the **%K** substitution for binding scripts.

−mode *notify*
Notify specifies the *mode* field for the event and must be one of **NotifyNormal**, **NotifyGrab**, **NotifyUngrab**, or **NotifyWhileGrabbed**. Valid for **Enter**, **Leave**, **FocusIn**, and **FocusOut** events. Corresponds to the **%m** substitution for binding scripts.

−override *boolean*
Boolean must be a boolean value; it specifies the *override_redirect* field for the event. Valid for **Map**, **Reparent**, and **Configure** events. Corresponds to the **%o** substitution for binding scripts.

−place *where*
Where specifies the *place* field for the event; it must be either **PlaceOnTop** or **PlaceOnBottom**. Valid for **Circulate** events. Corresponds to the **%p** substitution for binding scripts.

−root *window*
Window must be either a window path name or an integer window identifier; it specifies the *root* field for the event. Valid for **KeyPress**, **KeyRelease**, **ButtonPress**, **ButtonRelease**, **Enter**, **Leave**, and **Motion** events. Corresponds to the **%R** substitution for binding scripts.

−rootx *coord*
Coord must be a screen distance; it specifies the *x_root* field for the event. Valid for **KeyPress**,

KeyRelease, **ButtonPress**, **ButtonRelease**, **Enter**, **Leave**, and **Motion** events. Corresponds to the **%X** substitution for binding scripts.

−rooty *coord*
Coord must be a screen distance; it specifies th *y_root* field for the event. Valid for **KeyPress**, **KeyRelease**, **ButtonPress**, **ButtonRelease**, **Enter**, **Leave**, and **Motion** events. Corresponds to the **%Y** substitution for binding scripts.

−sendevent *boolean*
Boolean must be a boolean value; it specifies the *send_event* field for the event. Valid for all events. Corresponds to the **%E** substitution for binding scripts.

−serial *number*
Number must be an integer; it specifies the *serial* field for the event. Valid for all events. Corresponds to the **%#** substitution for binding scripts.

−state *state*
State specifies the *state* field for the event. For **KeyPress**, **KeyRelease**, **ButtonPress**, **ButtonRelease**, **Enter**, **Leave**, and **Motion** events it must be an integer value. For **Visibility** events it must be one of **VisibilityUnobscured**, **VisibilityPartiallyObscured**, or **VisibilityFullyObscured**. This option overrides any modifiers such as **Meta** or **Control** specified in the base *event*. Corresponds to the **%s** substitution for binding scripts.

−subwindow *window*
Window specifies the *subwindow* field for the event, either as a path name for a Tk widget or as an integer window identifier. Valid for **KeyPress**, **KeyRelease**, **ButtonPress**, **ButtonRelease**, **Enter**, **Leave**, and **Motion** events. Similar to **%S** substitution for binding scripts.

−time *integer*
Integer must be an integer value; it specifies the *time* field for the event. Valid for **KeyPress**, **KeyRelease**, **ButtonPress**, **ButtonRelease**, **Enter**, **Leave**, **Motion**, and **Property** events. Corresponds to the **%t** substitution for binding scripts.

−width *size*
Size must be a screen distance; it specifies the *width* field for the event. Valid for **Configure** events. Corresponds to the **%w** substitution for binding scripts.

−when *when*
When determines when the event will be processed; it must have one of the following values:

now Process the event immediately, before the command returns. This also happens if the **−when** option is omitted.

tail Place the event on Tcl's event queue behind any events already queued for this application.

head Place the event at the front of Tcl's event queue, so that it will be handled before any other events already queued.

mark Place the event at the front of Tcl's event queue but behind any other events already queued with **−when mark**. This option is useful when generating a series of events that should be processed in order but at the front of the queue.

−x *coord*
Coord must be a screen distance; it specifies the *x* field for the event. Valid for **KeyPress**, **KeyRelease**, **ButtonPress**, **ButtonRelease**, **Motion**, **Enter**, **Leave**, **Expose**, **Configure**, **Gravity**, and **Reparent** events. Corresponds to the the **%x** substitution for binding scripts.

−y *coord*
Coord must be a screen distance; it specifies the *y* field for the event. Valid for **KeyPress**,

KeyRelease, **ButtonPress**, **ButtonRelease**, **Motion**, **Enter**, **Leave**, **Expose**, **Configure**, **Gravity**, and **Reparent** events. Corresponds to the the **%y** substitution for binding scripts.

Any options that are not specified when generating an event are filled with the value 0, except for *serial*, which is filled with the next X event serial number.

VIRTUAL EVENT EXAMPLES

In order for a virtual event binding to trigger, two things must happen. First, the virtual event must be defined with the **event add** command. Second, a binding must be created for the virtual event with the **bind** command. Consider the following virtual event definitions:

```
event add <<Paste>> <Control-y>
event add <<Paste>> <Button-2>
event add <<Save>> <Control-X><Control-S>
event add <<Save>> <Shift-F12>
```

In the **bind** command, a virtual event can be bound like any other builtin event type as follows:

```
bind Entry <<Paste>> {%W insert [selection get]}
```

The double angle brackets are used to specify that a virtual event is being bound. If the user types Control-y or presses button 2, or if a **<<Paste>>** virtual event is synthesized with **event generate**, then the **<<Paste>>** binding will be invoked.

If a virtual binding has the exact same sequence as a separate physical binding, then the physical binding will take precedence. Consider the following example:

```
event add <<Paste>> <Control-y> <Meta-Control-y>
bind Entry <Control-y> {puts Control-y}
bind Entry <<Paste>> {puts Paste}
```

When the user types Control-y the **<Control-y>** binding will be invoked, because a physical event is considered more specific than a virtual event, all other things being equal. However, when the user types Meta-Control-y the **<<Paste>>** binding will be invoked, because the **Meta** modifier in the physical pattern associated with the virtual binding is more specific than the **<Control-y>** sequence for the physical event.

Bindings on a virtual event may be created before the virtual event exists. Indeed, the virtual event never actually needs to be defined, for instance, on platforms where the specific virtual event would meaningless or ungeneratable.

When a definition of a virtual event changes at run time, all windows will respond immediately to the new definition. Starting from the preceding example, if the following code is executed:

```
bind <Entry> <Control-y> {}
event add <<Paste>> <Key-F6>
```

the behavior will change such in two ways. First, the shadowed **<<Paste>>** binding will emerge. Typing Control-y will no longer invoke the **<Control-y>** binding, but instead invoke the virtual event **<<Paste>>**. Second, pressing the F6 key will now also invoke the **<<Paste>>** binding.

SEE ALSO

bind

KEYWORDS

event, binding, define, handle, virtual event

NAME

focus − Manage the input focus

SYNOPSIS

focus

focus *window*

focus *option* ?*arg arg ...*?

DESCRIPTION

The **focus** command is used to manage the Tk input focus. At any given time, one window on each display is designated as the *focus window*; any key press or key release events for the display are sent to that window. It is normally up to the window manager to redirect the focus among the top-level windows of a display. For example, some window managers automatically set the input focus to a top-level window whenever the mouse enters it; others redirect the input focus only when the user clicks on a window. Usually the window manager will set the focus only to top-level windows, leaving it up to the application to redirect the focus among the children of the top-level.

Tk remembers one focus window for each top-level (the most recent descendant of that top-level to receive the focus); when the window manager gives the focus to a top-level, Tk automatically redirects it to the remembered window. Within a top-level Tk uses an *explicit* focus model by default. Moving the mouse within a top-level does not normally change the focus; the focus changes only when a widget decides explicitly to claim the focus (e.g., because of a button click), or when the user types a key such as Tab that moves the focus.

The Tcl procedure **tk_focusFollowsMouse** may be invoked to create an *implicit* focus model: it reconfigures Tk so that the focus is set to a window whenever the mouse enters it. The Tcl procedures **tk_focusNext** and **tk_focusPrev** implement a focus order among the windows of a top-level; they are used in the default bindings for Tab and Shift-Tab, among other things.

The **focus** command can take any of the following forms:

focus Returns the path name of the focus window on the display containing the application's main window, or an empty string if no window in this application has the focus on that display. Note: it is better to specify the display explicitly using **−displayof** (see below) so that the code will work in applications using multiple displays.

focus *window*

If the application currently has the input focus on *window*'s display, this command resets the input focus for *window*'s display to *window* and returns an empty string. If the application doesn't currently have the input focus on *window*'s display, *window* will be remembered as the focus for its top-level; the next time the focus arrives at the top-level, Tk will redirect it to *window*. If *window* is an empty string then the command does nothing.

focus −displayof *window*

Returns the name of the focus window on the display containing *window*. If the focus window for *window*'s display isn't in this application, the return value is an empty string.

focus −force *window*

Sets the focus of *window*'s display to *window*, even if the application doesn't currently have the input focus for the display. This command should be used sparingly, if at all. In normal usage, an application should not claim the focus for itself; instead, it should wait for the window manager to give it the focus. If *window* is an empty string then the command does nothing.

focus −lastfor *window*

Returns the name of the most recent window to have the input focus among all the windows in the same top-level as *window*. If no window in that top-level has ever had the input focus, or if the most recent focus window has been deleted, then the name of the top-level is returned. The return value is the window that will receive the input focus the next time the window manager gives the focus to the top-level.

QUIRKS

When an internal window receives the input focus, Tk doesn't actually set the X focus to that window; as far as X is concerned, the focus will stay on the top-level window containing the window with the focus. However, Tk generates FocusIn and FocusOut events just as if the X focus were on the internal window. This approach gets around a number of problems that would occur if the X focus were actually moved; the fact that the X focus is on the top-level is invisible unless you use C code to query the X server directly.

KEYWORDS

events, focus, keyboard, top-level, window manager

NAME

tk_focusNext, tk_focusPrev, tk_focusFollowsMouse − Utility procedures for managing the input focus.

SYNOPSIS

tk_focusNext *window*

tk_focusPrev *window*

tk_focusFollowsMouse

DESCRIPTION

tk_focusNext is a utility procedure used for keyboard traversal. It returns the ‘‘next’’ window after *window* in focus order. The focus order is determined by the stacking order of windows and the structure of the window hierarchy. Among siblings, the focus order is the same as the stacking order, with the lowest window being first. If a window has children, the window is visited first, followed by its children (recursively), followed by its next sibling. Top-level windows other than *window* are skipped, so that **tk_focusNext** never returns a window in a different top-level from *window*.

After computing the next window, **tk_focusNext** examines the window’s **−takefocus** option to see whether it should be skipped. If so, **tk_focusNext** continues on to the next window in the focus order, until it eventually finds a window that will accept the focus or returns back to *window*.

tk_focusPrev is similar to **tk_focusNext** except that it returns the window just before *window* in the focus order.

tk_focusFollowsMouse changes the focus model for the application to an implicit one where the window under the mouse gets the focus. After this procedure is called, whenever the mouse enters a window Tk will automatically give it the input focus. The **focus** command may be used to move the focus to a window other than the one under the mouse, but as soon as the mouse moves into a new window the focus will jump to that window. Note: at present there is no built-in support for returning the application to an explicit focus model; to do this you’ll have to write a script that deletes the bindings created by **tk_focusFollowsMouse**.

KEYWORDS

focus, keyboard traversal, top-level

NAME

font − Create and inspect fonts.

SYNOPSIS

font *option* ?*arg arg ...*?

DESCRIPTION

The **font** command provides several facilities for dealing with fonts, such as defining named fonts and inspecting the actual attributes of a font. The command has several different forms, determined by the first argument. The following forms are currently supported:

font actual *font* ?**−displayof** *window*? ?*option*?

Returns information about the the actual attributes that are obtained when *font* is used on *window*'s display; the actual attributes obtained may differ from the attributes requested due to platform-dependant limitations, such as the availability of font families and pointsizes. *font* is a font description; see FONT DESCRIPTIONS below. If the *window* argument is omitted, it defaults to the main window. If *option* is specified, returns the value of that attribute; if it is omitted, the return value is a list of all the attributes and their values. See FONT OPTIONS below for a list of the possible attributes.

font configure *fontname* ?*option*? ?*value option value ...*?

Query or modify the desired attributes for the named font called *fontname*. If no *option* is specified, returns a list describing all the options and their values for *fontname*. If a single *option* is specified with no *value*, then returns the current value of that attribute. If one or more *option−value* pairs are specified, then the command modifies the given named font to have the given values; in this case, all widgets using that font will redisplay themselves using the new attributes for the font. See FONT OPTIONS below for a list of the possible attributes.

font create ?*fontname*? ?*option value ...*?

Creates a new named font and returns its name. *fontname* specifies the name for the font; if it is omitted, then Tk generates a new name of the form **font***x*, where *x* is an integer. There may be any number of *option−value* pairs, which provide the desired attributes for the new named font. See FONT OPTIONS below for a list of the possible attributes.

font delete *fontname* ?*fontname ...*?

Delete the specified named fonts. If there are widgets using the named font, the named font won't actually be deleted until all the instances are released. Those widgets will continue to display using the last known values for the named font. If a deleted named font is subsequently recreated with another call to **font create**, the widgets will use the new named font and redisplay themselves using the new attributes of that font.

font families ?**−displayof** *window*?

The return value is a list of the case-insensitive names of all font families that exist on *window*'s display. If the *window* argument is omitted, it defaults to the main window.

font measure *font* ?**−displayof** *window*? *text*

Measures the amount of space the string *text* would use in the given *font* when displayed in *window*. *font* is a font description; see FONT DESCRIPTIONS below. If the *window* argument is omitted, it defaults to the main window. The return value is the total width in pixels of *text*, not including the extra pixels used by highly exagerrated characters such as cursive ''f''. If the string contains newlines or tabs, those characters are not expanded or treated specially when measuring the string.

font metrics *font* ?**−displayof** *window*? ?*option*?

Returns information about the metrics (the font-specific data), for *font* when it is used on *window*'s display. *font* is a font description; see FONT DESCRIPTIONS below. If the *window* argument is

omitted, it defaults to the main window. If *option* is specified, returns the value of that metric; if it is omitted, the return value is a list of all the metrics and their values. See FONT METRICS below for a list of the possible metrics.

font names
The return value is a list of all the named fonts that are currently defined.

FONT DESCRIPTION

The following formats are accepted as a font description anywhere *font* is specified as an argument above; these same forms are also permitted when specifying the **−font** option for widgets.

[1] *fontname*
The name of a named font, created using the **font create** command. When a widget uses a named font, it is guaranteed that this will never cause an error, as long as the named font exists, no matter what potentially invalid or meaningless set of attributes the named font has. If the named font cannot be displayed with exactly the specified attributes, some other close font will be substituted automatically.

[2] *systemfont*
The platform-specific name of a font, interpreted by the graphics server. This also includes, under X, an XLFD (see [4]) for which a single ‘‘∗’’ character was used to elide more than one field in the middle of the name. See PLATFORM-SPECIFIC issues for a list of the system fonts.

[3] *family* ?*size*? ?*style*? ?*style* ...?
A properly formed list whose first element is the desired font *family* and whose optional second element is the desired *size*. The interpretation of the *size* attribute follows the same rules described for **−size** in FONT OPTIONS below. Any additional optional arguments following the *size* are font *style*s. Possible values for the *style* arguments are as follows:

normal	**bold**	**roman**	**italic**
underline	**overstrike**		

[4] X-font names (XLFD)
A Unix-centric font name of the form *-foundry-family-weight-slant-setwidth-addstyle-pixel-point-resx-resy-spacing-width-charset-encoding*. The ‘‘∗’’ character may be used to skip individual fields that the user does not care about. There must be exactly one ‘‘∗’’ for each field skipped, except that a ‘‘∗’’ at the end of the XLFD skips any remaining fields; the shortest valid XLFD is simply ‘‘∗’’, signifying all fields as defaults. Any fields that were skipped are given default values. For compatibility, an XLFD always chooses a font of the specified pixel size (not point size); although this interpretation is not strictly correct, all existing applications using XLFDs assumed that one ‘‘point’’ was in fact one pixel and would display incorrectly (generally larger) if the correct size font were actually used.

[5] *option value* ?*option value* ...?
A properly formed list of *option−value* pairs that specify the desired attributes of the font, in the same format used when defining a named font; see FONT OPTIONS below.

When font description *font* is used, the system attempts to parse the description according to each of the above five rules, in the order specified. Cases [1] and [2] must match the name of an existing named font or of a system font. Cases [3], [4], and [5] are accepted on all platforms and the closest available font will be used. In some situations it may not be possible to find any close font (e.g., the font family was a garbage value); in that case, some system-dependant default font is chosen. If the font description does not match any of the above patterns, an error is generated.

FONT METRICS

The following options are used by the **font metrics** command to query font-specific data determined when the font was created. These properties are for the whole font itself and not for individual characters drawn in that font. In the following definitions, the ''baseline'' of a font is the horizontal line where the bottom of most letters line up; certain letters, such as lower-case ''g'' stick below the baseline.

−ascent

The amount in pixels that the tallest letter sticks up above the baseline of the font, plus any extra blank space added by the designer of the font.

−descent

The largest amount in pixels that any letter sticks down below the baseline of the font, plus any extra blank space added by the designer of the font.

−linespace

Returns how far apart vertically in pixels two lines of text using the same font should be placed so that none of the characters in one line overlap any of the characters in the other line. This is generally the sum of the ascent above the baseline line plus the descent below the baseline.

−fixed

Returns a boolean flag that is ''**1**'' if this is a fixed-width font, where each normal character is the the same width as all the other characters, or is ''**0**'' if this is a proportionally-spaced font, where individual characters have different widths. The widths of control characters, tab characters, and other non-printing characters are not included when calculating this value.

FONT OPTIONS

The following options are supported on all platforms, and are used when constructing a named font or when specifying a font using style [5] as above:

−family *name*

The case-insensitive font family name. Tk guarantees to support the font families named **Courier** (a monospaced ''typewriter'' font), **Times** (a serifed ''newspaper'' font), and **Helvetica** (a sans-serif ''European'' font). The most closely matching native font family will automatically be substituted when one of the above font families is used. The *name* may also be the name of a native, platform-specific font family; in that case it will work as desired on one platform but may not display correctly on other platforms. If the family is unspecified or unrecognized, a platform-specific default font will be chosen.

−size *size*

The desired size of the font. If the *size* argument is a positive number, it is interpreted as a size in points. If *size* is a negative number, its absolute value is interpreted as a size in pixels. If a font cannot be displayed at the specified size, a nearby size will be chosen. If *size* is unspecified or zero, a platform-dependent default size will be chosen.

Sizes should normally be specified in points so the application will remain the same ruler size on the screen, even when changing screen resolutions or moving scripts across platforms. However, specifying pixels is useful in certain circumstances such as when a piece of text must line up with respect to a fixed-size bitmap. The mapping between points and pixels is set when the application starts, based on properties of the installed monitor, but it can be overridden by calling the **tk scaling** command.

−weight *weight*

The nominal thickness of the characters in the font. The value **normal** specifies a normal weight font, while **bold** specifies a bold font. The closest available weight to the one specified will be chosen. The default weight is **normal**.

−slant *slant*

The amount the characters in the font are slanted away from the vertical. Valid values for slant are

roman and **italic**. A roman font is the normal, upright appearance of a font, while an italic font is one that is tilted some number of degrees from upright. The closest available slant to the one specified will be chosen. The default slant is **roman**.

−underline *boolean*
The value is a boolean flag that specifies whether characters in this font should be underlined. The default value for underline is **false**.

−overstrike *boolean*
The value is a boolean flag that specifies whether a horizontal line should be drawn through the middle of characters in this font. The default value for overstrike is **false**.

PLATFORM-SPECIFIC ISSUES

The following named system fonts are supported:

X Windows:
All valid X font names, including those listed by xlsfonts(1), are available.

MS Windows:

system	**ansi**	**device**
systemfixed	**ansifixed**	**oemfixed**

Macintosh:

system	**application**

SEE ALSO

options

KEYWORDS

font

NAME

frame − Create and manipulate frame widgets

SYNOPSIS

frame *pathName* ?*options*?

STANDARD OPTIONS

−borderwidth	**−highlightbackground**	**−highlightthickness**	**−takefocus**
−cursor	**−highlightcolor**	**−relief**	

See the **options** manual entry for details on the standard options.

WIDGET-SPECIFIC OPTIONS

Command-Line Name: **−background**
Database Name: **background**
Database Class: **Background**

This option is the same as the standard **background** option except that its value may also be specified as an empty string. In this case, the widget will display no background or border, and no colors will be consumed from its colormap for its background and border.

Command-Line Name: **−class**
Database Name: **class**
Database Class: **Class**

Specifies a class for the window. This class will be used when querying the option database for the window's other options, and it will also be used later for other purposes such as bindings. The **class** option may not be changed with the **configure** widget command.

Command-Line Name: **−colormap**
Database Name: **colormap**
Database Class: **Colormap**

Specifies a colormap to use for the window. The value may be either **new**, in which case a new colormap is created for the window and its children, or the name of another window (which must be on the same screen and have the same visual as *pathName*), in which case the new window will use the colormap from the specified window. If the **colormap** option is not specified, the new window uses the same colormap as its parent. This option may not be changed with the **configure** widget command.

Command-Line Name: **−container**
Database Name: **container**
Database Class: **Container**

The value must be a boolean. If true, it means that this window will be used as a container in which some other application will be embedded (for example, a Tk toplevel can be embedded using the **−use** option). The window will support the appropriate window manager protocols for things like geometry requests. The window should not have any children of its own in this application. This option may not be changed with the **configure** widget command.

Command-Line Name: **−height**
Database Name: **height**
Database Class: **Height**

Specifies the desired height for the window in any of the forms acceptable to **Tk_GetPixels**. If this option is less than or equal to zero then the window will not request any size at all.

Command-Line Name: **−visual**
Database Name: **visual**
Database Class: **Visual**

Specifies visual information for the new window in any of the forms accepted by **Tk_GetVisual**. If this option is not specified, the new window will use the same visual as its parent. The **visual** option may not be modified with the **configure** widget command.

Command-Line Name: **−width**
Database Name: **width**
Database Class: **Width**

Specifies the desired width for the window in any of the forms acceptable to **Tk_GetPixels**. If this option is less than or equal to zero then the window will not request any size at all.

DESCRIPTION

The **frame** command creates a new window (given by the *pathName* argument) and makes it into a frame widget. Additional options, described above, may be specified on the command line or in the option database to configure aspects of the frame such as its background color and relief. The **frame** command returns the path name of the new window.

A frame is a simple widget. Its primary purpose is to act as a spacer or container for complex window layouts. The only features of a frame are its background color and an optional 3-D border to make the frame appear raised or sunken.

WIDGET COMMAND

The **frame** command creates a new Tcl command whose name is the same as the path name of the frame's window. This command may be used to invoke various operations on the widget. It has the following general form:

pathName option ?*arg arg ...*?

PathName is the name of the command, which is the same as the frame widget's path name. *Option* and the *arg*s determine the exact behavior of the command. The following commands are possible for frame widgets:

pathName **cget** *option*

Returns the current value of the configuration option given by *option*. *Option* may have any of the values accepted by the **frame** command.

pathName **configure** ?*option*? *?value option value ...*?

Query or modify the configuration options of the widget. If no *option* is specified, returns a list describing all of the available options for *pathName* (see **Tk_ConfigureInfo** for information on the format of this list). If *option* is specified with no *value*, then the command returns a list describing the one named option (this list will be identical to the corresponding sublist of the value returned if no *option* is specified). If one or more *option−value* pairs are specified, then the command modifies the given widget option(s) to have the given value(s); in this case the command returns an empty string. *Option* may have any of the values accepted by the **frame** command.

BINDINGS

When a new frame is created, it has no default event bindings: frames are not intended to be interactive.

KEYWORDS

frame, widget

NAME

tk_getOpenFile, tk_getSaveFile − pop up a dialog box for the user to select a file to open or save.

SYNOPSIS

tk_getOpenFile ?*option value ...*?
tk_getSaveFile ?*option value ...*?

DESCRIPTION

The procedures **tk_getOpenFile** and **tk_getSaveFile** pop up a dialog box for the user to select a file to open or save. The **tk_getOpenFile** command is usually associated with the **Open** command in the **File** menu. Its purpose is for the user to select an existing file *only*. If the user enters an non-existent file, the dialog box gives the user an error prompt and requires the user to give an alternative selection. If an application allows the user to create new files, it should do so by providing a separate **New** menu command.

The **tk_getSaveFile** command is usually associated with the **Save as** command in the **File** menu. If the user enters a file that already exists, the dialog box prompts the user for confirmation whether the existing file should be overwritten or not.

The following *option−value* pairs are possible as command line arguments to these two commands:

−defaultextension *extension*
Specifies a string that will be appended to the filename if the user enters a filename without an extension. The defaut value is the empty string, which means no extension will be appended to the filename in any case. This option is ignored on the Macintosh platform, which does not require extensions to filenames.

−filetypes *filePatternList*
If a **File types** listbox exists in the file dialog on the particular platform, this option gives the *filetype*s in this listbox. When the user choose a filetype in the listbox, only the files of that type are listed. If this option is unspecified, or if it is set to the empty list, or if the **File types** listbox is not supported by the particular platform then all files are listed regardless of their types. See the section SPECIFYING FILE PATTERNS below for a discussion on the contents of *filePatternList*.

−initialdir *directory*
Specifies that the files in *directory* should be displayed when the dialog pops up. If this parameter is not specified, then the files in the current working directory are displayed. If the parameter specifies a relative path, the return value will convert the relative path to an absolute path. This option may not always work on the Macintosh. This is not a bug. Rather, the *General Controls* control panel on the Mac allows the end user to override the application default directory.

−initialfile *filename*
Specifies a filename to be displayed in the dialog when it pops up. This option is ignored by the **tk_getOpenFile** command.

−parent *window*
Makes *window* the logical parent of the file dialog. The file dialog is displayed on top of its parent window.

−title *titleString*
Specifies a string to display as the title of the dialog box. If this option is not specified, then a default title is displayed. This option is ignored on the Macintosh platform.

If the user selects a file, both **tk_getOpenFile** and **tk_getSaveFile** return the full pathname of this file. If the user cancels the operation, both commands return the empty string.

SPECIFYING FILE PATTERNS

The *filePatternList* value given by the **−filetypes** option is a list of file patterns. Each file pattern is a list of the form

typeName {*extension* ?*extension* ...?} ?{*macType* ?*macType* ...?}?

typeName is the name of the file type described by this file pattern and is the text string that appears in the **File types** listbox. *extension* is a file extension for this file pattern. *macType* is a four-character Macintosh file type. The list of *macType*s is optional and may be omitted for applications that do not need to execute on the Macintosh platform.

Several file patterns may have the same *typeName*, in which case they refer to the same file type and share the same entry in the listbox. When the user selects an entry in the listbox, all the files that match at least one of the file patterns corresponding to that entry are listed. Usually, each file pattern corresponds to a distinct type of file. The use of more than one file patterns for one type of file is necessary on the Macintosh platform only.

On the Macintosh platform, a file matches a file pattern if its name matches at least one of the *extension*(s) AND it belongs to at least one of the *macType*(s) of the file pattern. For example, the **C Source Files** file pattern in the sample code matches with files that have a **.c** extension AND belong to the *macType* **TEXT**. To use the OR rule instead, you can use two file patterns, one with the *extensions* only and the other with the *macType* only. The **GIF Files** file type in the sample code matches files that EITHER have a **.gif** extension OR belong to the *macType* **GIFF**.

On the Unix and Windows platforms, a file matches a file pattern if its name matches at at least one of the *extension*(s) of the file pattern. The *macType*s are ignored.

SPECIFYING EXTENSIONS

On the Unix and Macintosh platforms, extensions are matched using glob-style pattern matching. On the Windows platforms, extensions are matched by the underlying operating system. The types of possible extensions are: (1) the special extension ∗ matches any file; (2) the special extension "" matches any files that do not have an extension (i.e., the filename contains no full stop character); (3) any character string that does not contain any wild card characters (∗ and ?).

Due to the different pattern matching rules on the various platforms, to ensure portability, wild card characters are not allowed in the extensions, except as in the special extension ∗. Extensions without a full stop character (e.g, ˜) are allowed but may not work on all platforms.

EXAMPLE

```
set types {
   {{Text Files}       {.txt}       }
   {{TCL Scripts}      {.tcl}       }
   {{C Source Files}   {.c}     TEXT}
   {{GIF Files}        {.gif}       }
   {{GIF Files}        {}       GIFF}
   {{All Files}        *            }
}
set filename [tk_getOpenFile -filetypes $types]

if {$filename != ""} {
   # Open the file ...
}
```

KEYWORDS

file selection dialog

NAME

grab − Confine pointer and keyboard events to a window sub-tree

SYNOPSIS

grab ?−**global**? *window*

grab *option* ?*arg arg* ...?

DESCRIPTION

This command implements simple pointer and keyboard grabs for Tk. Tk's grabs are different than the grabs described in the Xlib documentation. When a grab is set for a particular window, Tk restricts all pointer events to the grab window and its descendants in Tk's window hierarchy. Whenever the pointer is within the grab window's subtree, the pointer will behave exactly the same as if there had been no grab at all and all events will be reported in the normal fashion. When the pointer is outside *window*'s tree, button presses and releases and mouse motion events are reported to *window*, and window entry and window exit events are ignored. The grab subtree ''owns'' the pointer: windows outside the grab subtree will be visible on the screen but they will be insensitive until the grab is released. The tree of windows underneath the grab window can include top-level windows, in which case all of those top-level windows and their descendants will continue to receive mouse events during the grab.

Two forms of grabs are possible: local and global. A local grab affects only the grabbing application: events will be reported to other applications as if the grab had never occurred. Grabs are local by default. A global grab locks out all applications on the screen, so that only the given subtree of the grabbing application will be sensitive to pointer events (mouse button presses, mouse button releases, pointer motions, window entries, and window exits). During global grabs the window manager will not receive pointer events either.

During local grabs, keyboard events (key presses and key releases) are delivered as usual: the window manager controls which application receives keyboard events, and if they are sent to any window in the grabbing application then they are redirected to the focus window. During a global grab Tk grabs the keyboard so that all keyboard events are always sent to the grabbing application. The **focus** command is still used to determine which window in the application receives the keyboard events. The keyboard grab is released when the grab is released.

Grabs apply to particular displays. If an application has windows on multiple displays then it can establish a separate grab on each display. The grab on a particular display affects only the windows on that display. It is possible for different applications on a single display to have simultaneous local grabs, but only one application can have a global grab on a given display at once.

The **grab** command can take any of the following forms:

grab ?−**global**? *window*
: Same as **grab set**, described below.

grab current ?*window*?
: If *window* is specified, returns the name of the current grab window in this application for *window*'s display, or an empty string if there is no such window. If *window* is omitted, the command returns a list whose elements are all of the windows grabbed by this application for all displays, or an empty string if the application has no grabs.

grab release *window*
: Releases the grab on *window* if there is one, otherwise does nothing. Returns an empty string.

grab set ?−**global**? *window*
: Sets a grab on *window*. If **−global** is specified then the grab is global, otherwise it is local. If a grab was already in effect for this application on *window*'s display then it is automatically

released. If there is already a grab on *window* and it has the same global/local form as the requested grab, then the command does nothing. Returns an empty string.

grab status *window*

Returns **none** if no grab is currently set on *window*, **local** if a local grab is set on *window*, and **global** if a global grab is set.

BUGS

It took an incredibly complex and gross implementation to produce the simple grab effect described above. Given the current implementation, it isn't safe for applications to use the Xlib grab facilities at all except through the Tk grab procedures. If applications try to manipulate X's grab mechanisms directly, things will probably break.

If a single process is managing several different Tk applications, only one of those applications can have a local grab for a given display at any given time. If the applications are in different processes, this restriction doesn't exist.

KEYWORDS

grab, keyboard events, pointer events, window

NAME

grid − Geometry manager that arranges widgets in a grid

SYNOPSIS

grid *option arg* ?*arg ...*?

DESCRIPTION

The **grid** command is used to communicate with the grid geometry manager that arranges widgets in rows and columns inside of another window, called the geometry master (or master window). The **grid** command can have any of several forms, depending on the *option* argument:

grid *slave* ?*slave ...*? ?*options*?

If the first argument to **grid** is a window name (any value starting with ‘‘.’’), then the command is processed in the same way as **grid configure**.

grid bbox *master* ?*column row*? ?*column2 row2*?

With no arguments, the bounding box (in pixels) of the grid is returned. The return value consists of 4 integers. The first two are the pixel offset from the master window (x then y) of the top-left corner of the grid, and the second two integers are the width and height of the grid, also in pixels. If a single *column* and *row* is specified on the command line, then the bounding box for that cell is returned, where the top left cell is numbered from zero. If both *column* and *row* arguments are specified, then the bounding box spanning the rows and columns indicated is returned.

grid columnconfigure *master index* ?*−option value...*?

Query or set the column properties of the *index* column of the geometry master, *master*. The valid options are **−minsize**, **−weight** and **-pad**. If one or more options are provided, then *index* may be given as a list of column indeces to which the configuration options will operate on. The **−minsize** option sets the minimum size, in screen units, that will be permitted for this column. The **−weight** option (an integer value) sets the relative weight for apportioning any extra spaces among columns. A weight of zero (0) indicates the column will not deviate from its requested size. A column whose weight is two will grow at twice the rate as a column of weight one when extra space is allocated to the layout. The **-pad** option specifies the number of screen units that will be added to the largest window contained completely in that column when the grid geometry manager requests a size from the containing window. If only an option is specified, with no value, the current value of that option is returned. If only the master window and index is specified, all the current settings are returned in an list of "-option value" pairs.

grid configure *slave* ?*slave ...*? ?*options*?

The arguments consist of the names of one or more slave windows followed by pairs of arguments that specify how to manage the slaves. The characters **−**, **x** and **^**, can be specified instead of a window name to alter the default location of a *slave*, as described in the ‘‘RELATIVE PLACEMENT’’ section, below. The following options are supported:

−column *n*

Insert the slave so that it occupies the *n*th column in the grid. Column numbers start with 0. If this option is not supplied, then the slave is arranged just to the right of previous slave specified on this call to *grid*, or column "0" if it is the first slave. For each **x** that immediately precedes the *slave*, the column position is incremented by one. Thus the **x** represents a blank column for this row in the grid.

−columnspan *n*

Insert the slave so that it occupies *n* columns in the grid. The default is one column, unless the window name is followed by a **−**, in which case the columnspan is incremented once for each immediately following **−**.

−in *other*
Insert the slave(s) in the master window given by *other*. The default is the first slave's parent window.

−ipadx *amount*
The *amount* specifies how much horizontal internal padding to leave on each side of the slave(s). This is space is added inside the slave(s) border. The *amount* must be a valid screen distance, such as **2** or **.5c**. It defaults to 0.

−ipady *amount*
The *amount* specifies how much vertical internal padding to leave on on the top and bottom of the slave(s). This space is added inside the slave(s) border. The *amount* defaults to 0.

−padx *amount*
The *amount* specifies how much horizontal external padding to leave on each side of the slave(s), in screen units. The *amount* defaults to 0. This space is added outside the slave(s) border.

−pady *amount*
The *amount* specifies how much vertical external padding to leave on the top and bottom of the slave(s), in screen units. The *amount* defaults to 0. This space is added outside the slave(s) border.

−row *n* Insert the slave so that it occupies the *n*th row in the grid. Row numbers start with 0. If this option is not supplied, then the slave is arranged on the same row as the previous slave specified on this call to **grid**, or the first unoccupied row if this is the first slave.

−rowspan *n*
Insert the slave so that it occupies *n* rows in the grid. The default is one row. If the next **grid** command contains ^ characters instead of *slaves* that line up with the columns of this *slave*, then the **rowspan** of this *slave* is extended by one.

−sticky *style*
If a slave's cell is larger than its requested dimensions, this option may be used to position (or stretch) the slave within its cell. *Style* is a string that contains zero or more of the characters **n**, **s**, **e** or **w**. The string can optionally contains spaces or commas, but they are ignored. Each letter refers to a side (north, south, east, or west) that the slave will "stick" to. If both **n** and **s** (or **e** and **w**) are specified, the slave will be stretched to fill the entire height (or width) of its cavity. The **sticky** option subsumes the combination of **−anchor** and **−fill** that is used by **pack**. The default is **{}**, which causes the slave to be centered in its cavity, at its requested size.

If any of the slaves are already managed by the geometry manager then any unspecified options for them retain their previous values rather than receiving default values.

grid forget *slave* ?*slave* ...?
Removes each of the *slave*s from grid for its master and unmaps their windows. The slaves will no longer be managed by the grid geometry manager. The configuration options for that window are forgotten, so that if the slave is managed once more by the grid geometry manager, the initial default settings are used.

grid info *slave*
Returns a list whose elements are the current configuration state of the slave given by *slave* in the same option-value form that might be specified to **grid configure**. The first two elements of the list are ''**−in** *master*'' where *master* is the slave's master.

grid location *master x y*

Given *x* and *y* values in screen units relative to the master window, the column and row number at that *x* and *y* location is returned. For locations that are above or to the left of the grid, **-1** is returned.

grid propagate *master* ?*boolean*?

If *boolean* has a true boolean value such as **1** or **on** then propagation is enabled for *master*, which must be a window name (see ''GEOMETRY PROPAGATION'' below). If *boolean* has a false boolean value then propagation is disabled for *master*. In either of these cases an empty string is returned. If *boolean* is omitted then the command returns **0** or **1** to indicate whether propagation is currently enabled for *master*. Propagation is enabled by default.

grid rowconfigure *master index* ?−*option value...*?

Query or set the row properties of the *index* row of the geometry master, *master*. The valid options are **−minsize**, **−weight** and **-pad**. If one or more options are provided, then *index* may be given as a list of row indeces to which the configuration options will operate on. The **−minsize** option sets the minimum size, in screen units, that will be permitted for this row. The **−weight** option (an integer value) sets the relative weight for apportioning any extra spaces among rows. A weight of zero (0) indicates the row will not deviate from its requested size. A row whose weight is two will grow at twice the rate as a row of weight one when extra space is allocated to the layout. The **-pad** option specifies the number of screen units that will be added to the largest window contained completely in that row when the grid geometry manager requests a size from the containing window. If only an option is specified, with no value, the current value of that option is returned. If only the master window and index is specified, all the current settings are returned in an list of "-option value" pairs.

grid remove *slave* ?*slave ...*?

Removes each of the *slave*s from grid for its master and unmaps their windows. The slaves will no longer be managed by the grid geometry manager. However, the configuration options for that window are remembered, so that if the slave is managed once more by the grid geometry manager, the previous values are retained.

grid size *master*

Returns the size of the grid (in columns then rows) for *master*. The size is determined either by the *slave* occupying the largest row or column, or the largest column or row with a **minsize**, **weight**, or **pad** that is non-zero.

grid slaves *master* ?−*option value*?

If no options are supplied, a list of all of the slaves in *master* are returned, most recently manages first. *Option* can be either **−row** or **−column** which causes only the slaves in the row (or column) specified by *value* to be returned.

RELATIVE PLACEMENT

The **grid** command contains a limited set of capabilities that permit layouts to be created without specifying the row and column information for each slave. This permits slaves to be rearranged, added, or removed without the need to explicitly specify row and column information. When no column or row information is specified for a *slave*, default values are chosen for **column**, **row**, **columnspan** and **rowspan** at the time the *slave* is managed. The values are chosen based upon the current layout of the grid, the position of the *slave* relative to other *slave*s in the same grid command, and the presence of the characters **−**, **^**, and **^** in **grid** command where *slave* names are normally expected.

− This increases the columnspan of the *slave* to the left. Several −'s in a row will successively increase the columnspan. A − may not follow a **^** or a **x**.

x This leaves an empty column between the *slave* on the left and the *slave* on the right.

^ This extends the **rowspan** of the *slave* above the ^'s in the grid. The number of ^'s in a row must match the number of columns spanned by the *slave* above it.

THE GRID ALGORITHM

The grid geometry manager lays out its slaves in three steps. In the first step, the minimum size needed to fit all of the slaves is computed, then (if propagation is turned on), a request is made of the master window to become that size. In the second step, the requested size is compared against the actual size of the master. If the sizes are different, then spaces is added to or taken away from the layout as needed. For the final step, each slave is positioned in its row(s) and column(s) based on the setting of its *sticky* flag.

To compute the minimum size of a layout, the grid geometry manager first looks at all slaves whose columnspan and rowspan values are one, and computes the nominal size of each row or column to be either the *minsize* for that row or column, or the sum of the *pad*ding plus the size of the largest slave, whichever is greater. Then the slaves whose rowspans or columnspans are greater than one are examined. If a group of rows or columns need to be increased in size in order to accommodate these slaves, then extra space is added to each row or column in the group according to its *weight*. For each group whose weights are all zero, the additional space is apportioned equally.

For masters whose size is larger than the requested layout, the additional space is apportioned according to the row and column weights. If all of the weights are zero, the layout is centered within its master. For masters whose size is smaller than the requested layout, space is taken away from columns and rows according to their weights. However, once a column or row shrinks to its minsize, its weight is taken to be zero. If more space needs to be removed from a layout than would be permitted, as when all the rows or columns are at there minimum sizes, the layout is clipped on the bottom and right.

GEOMETRY PROPAGATION

The grid geometry manager normally computes how large a master must be to just exactly meet the needs of its slaves, and it sets the requested width and height of the master to these dimensions. This causes geometry information to propagate up through a window hierarchy to a top-level window so that the entire sub-tree sizes itself to fit the needs of the leaf windows. However, the **grid propagate** command may be used to turn off propagation for one or more masters. If propagation is disabled then grid will not set the requested width and height of the master window. This may be useful if, for example, you wish for a master window to have a fixed size that you specify.

RESTRICTIONS ON MASTER WINDOWS

The master for each slave must either be the slave's parent (the default) or a descendant of the slave's parent. This restriction is necessary to guarantee that the slave can be placed over any part of its master that is visible without danger of the slave being clipped by its parent. In addition, all slaves in one call to **grid** must have the same master.

STACKING ORDER

If the master for a slave is not its parent then you must make sure that the slave is higher in the stacking order than the master. Otherwise the master will obscure the slave and it will appear as if the slave hasn't been managed correctly. The easiest way to make sure the slave is higher than the master is to create the master window first: the most recently created window will be highest in the stacking order.

CREDITS

The **grid** command is based on ideas taken from the *GridBag* geometry manager written by Doug. Stein, and the **blt_table** geometry manager, written by George Howlett.

KEYWORDS

geometry manager, location, grid, cell, propagation, size, pack

NAME

image − Create and manipulate images

SYNOPSIS

image *option* ?*arg arg* ...?

DESCRIPTION

The **image** command is used to create, delete, and query images. It can take several different forms, depending on the *option* argument. The legal forms are:

image create *type* ?*name*? ?*option value* ...?

Creates a new image and returns its name. *type* specifies the type of the image, which must be one of the types currently defined (e.g., **bitmap**). *name* specifies the name for the image; if it is omitted then Tk picks a name of the form **image***x*, where *x* is an integer. There may be any number of *option−value* pairs, which provide configuration options for the new image. The legal set of options is defined separately for each image type; see below for details on the options for built-in image types. If an image already exists by the given name then it is replaced with the new image and any instances of that image will redisplay with the new contents.

image delete ?*name name* ...?

Deletes each of the named images and returns an empty string. If there are instances of the images displayed in widgets, the images won't actually be deleted until all of the instances are released. However, the association between the instances and the image manager will be dropped. Existing instances will retain their sizes but redisplay as empty areas. If a deleted image is recreated with another call to **image create**, the existing instances will use the new image.

image height *name*

Returns a decimal string giving the height of image *name* in pixels.

image names

Returns a list containing the names of all existing images.

image type *name*

Returns the type of image *name* (the value of the *type* argument to **image create** when the image was created).

image types

Returns a list whose elements are all of the valid image types (i.e., all of the values that may be supplied for the *type* argument to **image create**).

image width *name*

Returns a decimal string giving the width of image *name* in pixels.

BUILT-IN IMAGE TYPES

The following image types are defined by Tk so they will be available in any Tk application. Individual applications or extensions may define additional types.

bitmap Each pixel in the image displays a foreground color, a background color, or nothing. See the **bitmap** manual entry for more information.

photo Displays a variety of full-color images, using dithering to approximate colors on displays with limited color capabilities. See the **photo** manual entry for more information.

KEYWORDS

height, image, types of images, width

NAME

label − Create and manipulate label widgets

SYNOPSIS

label *pathName* ?*options*?

STANDARD OPTIONS

−anchor	**−font**	**−image**	**−takefocus**
−background	**−foreground**	**−justify**	**−text**
−bitmap	**−highlightbackground**	**−padx**	**−textvariable**
−borderwidth	**−highlightcolor**	**−pady**	**−underline**
−cursor	**−highlightthickness**	**−relief**	**−wraplength**

See the **options** manual entry for details on the standard options.

WIDGET-SPECIFIC OPTIONS

Command-Line Name: **−height**
Database Name: **height**
Database Class: **Height**

Specifies a desired height for the label. If an image or bitmap is being displayed in the label then the value is in screen units (i.e. any of the forms acceptable to **Tk_GetPixels**); for text it is in lines of text. If this option isn't specified, the label's desired height is computed from the size of the image or bitmap or text being displayed in it.

Command-Line Name: **−width**
Database Name: **width**
Database Class: **Width**

Specifies a desired width for the label. If an image or bitmap is being displayed in the label then the value is in screen units (i.e. any of the forms acceptable to **Tk_GetPixels**); for text it is in characters. If this option isn't specified, the label's desired width is computed from the size of the image or bitmap or text being displayed in it.

DESCRIPTION

The **label** command creates a new window (given by the *pathName* argument) and makes it into a label widget. Additional options, described above, may be specified on the command line or in the option database to configure aspects of the label such as its colors, font, text, and initial relief. The **label** command returns its *pathName* argument. At the time this command is invoked, there must not exist a window named *pathName*, but *pathName*'s parent must exist.

A label is a widget that displays a textual string, bitmap or image. If text is displayed, it must all be in a single font, but it can occupy multiple lines on the screen (if it contains newlines or if wrapping occurs because of the **wrapLength** option) and one of the characters may optionally be underlined using the **underline** option. The label can be manipulated in a few simple ways, such as changing its relief or text, using the commands described below.

WIDGET COMMAND

The **label** command creates a new Tcl command whose name is *pathName*. This command may be used to invoke various operations on the widget. It has the following general form:

pathName option ?*arg arg ...*?

Option and the *arg*s determine the exact behavior of the command. The following commands are possible for label widgets:

pathName **cget** *option*

Returns the current value of the configuration option given by *option*. *Option* may have any of the values accepted by the **label** command.

pathName **configure** ?*option*? ?*value option value ...*?
Query or modify the configuration options of the widget. If no *option* is specified, returns a list describing all of the available options for *pathName* (see **Tk_ConfigureInfo** for information on the format of this list). If *option* is specified with no *value*, then the command returns a list describing the one named option (this list will be identical to the corresponding sublist of the value returned if no *option* is specified). If one or more *option−value* pairs are specified, then the command modifies the given widget option(s) to have the given value(s); in this case the command returns an empty string. *Option* may have any of the values accepted by the **label** command.

BINDINGS

When a new label is created, it has no default event bindings: labels are not intended to be interactive.

KEYWORDS

label, widget

NAME

listbox − Create and manipulate listbox widgets

SYNOPSIS

listbox *pathName* ?*options*?

STANDARD OPTIONS

−background	**−foreground**	**−relief**	**−takefocus**
−borderwidth	**−height**	**−selectbackground**	**−width**
−cursor	**−highlightbackground**	**−selectborderwidth**	**−xscrollcommand**
−exportselection	**−highlightcolor**	**−selectforeground**	**−yscrollcommand**
−font	**−highlightthickness**	**−setgrid**	

See the **options** manual entry for details on the standard options.

WIDGET-SPECIFIC OPTIONS

Command-Line Name: **−height**
Database Name: **height**
Database Class: **Height**

Specifies the desired height for the window, in lines. If zero or less, then the desired height for the window is made just large enough to hold all the elements in the listbox.

Command-Line Name: **−selectmode**
Database Name: **selectMode**
Database Class: **SelectMode**

Specifies one of several styles for manipulating the selection. The value of the option may be arbitrary, but the default bindings expect it to be either **single**, **browse**, **multiple**, or **extended**; the default value is **browse**.

Command-Line Name: **−width**
Database Name: **width**
Database Class: **Width**

Specifies the desired width for the window in characters. If the font doesn't have a uniform width then the width of the character "0" is used in translating from character units to screen units. If zero or less, then the desired width for the window is made just large enough to hold all the elements in the listbox.

DESCRIPTION

The **listbox** command creates a new window (given by the *pathName* argument) and makes it into a listbox widget. Additional options, described above, may be specified on the command line or in the option database to configure aspects of the listbox such as its colors, font, text, and relief. The **listbox** command returns its *pathName* argument. At the time this command is invoked, there must not exist a window named *pathName*, but *pathName*'s parent must exist.

A listbox is a widget that displays a list of strings, one per line. When first created, a new listbox has no elements. Elements may be added or deleted using widget commands described below. In addition, one or more elements may be selected as described below. If a listbox is exporting its selection (see **exportSelection** option), then it will observe the standard X11 protocols for handling the selection. Listbox selections are available as type **STRING**; the value of the selection will be the text of the selected elements, with newlines separating the elements.

It is not necessary for all the elements to be displayed in the listbox window at once; commands described below may be used to change the view in the window. Listboxes allow scrolling in both directions using the standard **xScrollCommand** and **yScrollCommand** options. They also support scanning, as described

below.

INDICES

Many of the widget commands for listboxes take one or more indices as arguments. An index specifies a particular element of the listbox, in any of the following ways:

number Specifies the element as a numerical index, where 0 corresponds to the first element in the listbox.

active Indicates the element that has the location cursor. This element will be displayed with an underline when the listbox has the keyboard focus, and it is specified with the **activate** widget command.

anchor Indicates the anchor point for the selection, which is set with the **selection anchor** widget command.

end Indicates the end of the listbox. For most commands this refers to the last element in the listbox, but for a few commands such as **index** and **insert** it refers to the element just after the last one.

@x,y Indicates the element that covers the point in the listbox window specified by *x* and *y* (in pixel coordinates). If no element covers that point, then the closest element to that point is used.

In the widget command descriptions below, arguments named *index*, *first*, and *last* always contain text indices in one of the above forms.

WIDGET COMMAND

The **listbox** command creates a new Tcl command whose name is *pathName*. This command may be used to invoke various operations on the widget. It has the following general form:

pathName option ?*arg arg ...*?

Option and the *arg*s determine the exact behavior of the command. The following commands are possible for listbox widgets:

pathName **activate** *index*

Sets the active element to the one indicated by *index*. If *index* is outside the range of elements in the listbox then the closest element is activated. The active element is drawn with an underline when the widget has the input focus, and its index may be retrieved with the index **active**.

pathName **bbox** *index*

Returns a list of four numbers describing the bounding box of the text in the element given by *index*. The first two elements of the list give the x and y coordinates of the upper-left corner of the screen area covered by the text (specified in pixels relative to the widget) and the last two elements give the width and height of the area, in pixels. If no part of the element given by *index* is visible on the screen, or if *index* refers to a non-existent element, then the result is an empty string; if the element is partially visible, the result gives the full area of the element, including any parts that are not visible.

pathName **cget** *option*

Returns the current value of the configuration option given by *option*. *Option* may have any of the values accepted by the **listbox** command.

pathName **configure** ?*option*? ?*value option value ...*?

Query or modify the configuration options of the widget. If no *option* is specified, returns a list describing all of the available options for *pathName* (see **Tk_ConfigureInfo** for information on the format of this list). If *option* is specified with no *value*, then the command returns a list describing the one named option (this list will be identical to the corresponding sublist of the value

returned if no *option* is specified). If one or more *option−value* pairs are specified, then the command modifies the given widget option(s) to have the given value(s); in this case the command returns an empty string. *Option* may have any of the values accepted by the **listbox** command.

pathName **curselection**
Returns a list containing the numerical indices of all of the elements in the listbox that are currently selected. If there are no elements selected in the listbox then an empty string is returned.

pathName **delete** *first* ?*last*?
Deletes one or more elements of the listbox. *First* and *last* are indices specifying the first and last elements in the range to delete. If *last* isn't specified it defaults to *first*, i.e. a single element is deleted.

pathName **get** *first* ?*last*?
If *last* is omitted, returns the contents of the listbox element indicated by *first*, or an empty string if *first* refers to a non-existent element. If *last* is specified, the command returns a list whose elements are all of the listbox elements between *first* and *last*, inclusive. Both *first* and *last* may have any of the standard forms for indices.

pathName **index** *index*
Returns the integer index value that corresponds to *index*. If *index* is **end** the return value is a count of the number of elements in the listbox (not the index of the last element).

pathName **insert** *index* ?*element element ...*?
Inserts zero or more new elements in the list just before the element given by *index*. If *index* is specified as **end** then the new elements are added to the end of the list. Returns an empty string.

pathName **nearest** *y*
Given a y-coordinate within the listbox window, this command returns the index of the (visible) listbox element nearest to that y-coordinate.

pathName **scan** *option args*
This command is used to implement scanning on listboxes. It has two forms, depending on *option*:

pathName **scan mark** *x y*
Records *x* and *y* and the current view in the listbox window; used in conjunction with later **scan dragto** commands. Typically this command is associated with a mouse button press in the widget. It returns an empty string.

pathName **scan dragto** *x y*.
This command computes the difference between its *x* and *y* arguments and the *x* and *y* arguments to the last **scan mark** command for the widget. It then adjusts the view by 10 times the difference in coordinates. This command is typically associated with mouse motion events in the widget, to produce the effect of dragging the list at high speed through the window. The return value is an empty string.

pathName **see** *index*
Adjust the view in the listbox so that the element given by *index* is visible. If the element is already visible then the command has no effect; if the element is near one edge of the window then the listbox scrolls to bring the element into view at the edge; otherwise the listbox scrolls to center the element.

pathName **selection** *option arg*
This command is used to adjust the selection within a listbox. It has several forms, depending on *option*:

pathName **selection anchor** *index*
Sets the selection anchor to the element given by *index*. If *index* refers to a non-existent

element, then the closest element is used. The selection anchor is the end of the selection that is fixed while dragging out a selection with the mouse. The index **anchor** may be used to refer to the anchor element.

pathName **selection clear** *first* ?*last*?
If any of the elements between *first* and *last* (inclusive) are selected, they are deselected. The selection state is not changed for elements outside this range.

pathName **selection includes** *index*
Returns 1 if the element indicated by *index* is currently selected, 0 if it isn't.

pathName **selection set** *first* ?*last*?
Selects all of the elements in the range between *first* and *last*, inclusive, without affecting the selection state of elements outside that range.

pathName **size**
Returns a decimal string indicating the total number of elements in the listbox.

pathName **xview** *args*
This command is used to query and change the horizontal position of the information in the widget's window. It can take any of the following forms:

pathName **xview**
Returns a list containing two elements. Each element is a real fraction between 0 and 1; together they describe the horizontal span that is visible in the window. For example, if the first element is .2 and the second element is .6, 20% of the listbox's text is off-screen to the left, the middle 40% is visible in the window, and 40% of the text is off-screen to the right. These are the same values passed to scrollbars via the **−xscrollcommand** option.

pathName **xview** *index*
Adjusts the view in the window so that the character position given by *index* is displayed at the left edge of the window. Character positions are defined by the width of the character **0**.

pathName **xview moveto** *fraction*
Adjusts the view in the window so that *fraction* of the total width of the listbox text is off-screen to the left. *fraction* must be a fraction between 0 and 1.

pathName **xview scroll** *number what*
This command shifts the view in the window left or right according to *number* and *what*. *Number* must be an integer. *What* must be either **units** or **pages** or an abbreviation of one of these. If *what* is **units**, the view adjusts left or right by *number* character units (the width of the **0** character) on the display; if it is **pages** then the view adjusts by *number* screenfuls. If *number* is negative then characters farther to the left become visible; if it is positive then characters farther to the right become visible.

pathName **yview** *?args*?
This command is used to query and change the vertical position of the text in the widget's window. It can take any of the following forms:

pathName **yview**
Returns a list containing two elements, both of which are real fractions between 0 and 1. The first element gives the position of the listbox element at the top of the window, relative to the listbox as a whole (0.5 means it is halfway through the listbox, for example). The second element gives the position of the listbox element just after the last one in the window, relative to the listbox as a whole. These are the same values passed to scrollbars via the **−yscrollcommand** option.

pathName **yview** *index*
Adjusts the view in the window so that the element given by *index* is displayed at the top of the window.

pathName **yview moveto** *fraction*
Adjusts the view in the window so that the element given by *fraction* appears at the top of the window. *Fraction* is a fraction between 0 and 1; 0 indicates the first element in the listbox, 0.33 indicates the element one-third the way through the listbox, and so on.

pathName **yview scroll** *number what*
This command adjusts the view in the window up or down according to *number* and *what*. *Number* must be an integer. *What* must be either **units** or **pages**. If *what* is **units**, the view adjusts up or down by *number* lines; if it is **pages** then the view adjusts by *number* screenfuls. If *number* is negative then earlier elements become visible; if it is positive then later elements become visible.

DEFAULT BINDINGS

Tk automatically creates class bindings for listboxes that give them Motif-like behavior. Much of the behavior of a listbox is determined by its **selectMode** option, which selects one of four ways of dealing with the selection.

If the selection mode is **single** or **browse**, at most one element can be selected in the listbox at once. In both modes, clicking button 1 on an element selects it and deselects any other selected item. In **browse** mode it is also possible to drag the selection with button 1.

If the selection mode is **multiple** or **extended**, any number of elements may be selected at once, including discontiguous ranges. In **multiple** mode, clicking button 1 on an element toggles its selection state without affecting any other elements. In **extended** mode, pressing button 1 on an element selects it, deselects everything else, and sets the anchor to the element under the mouse; dragging the mouse with button 1 down extends the selection to include all the elements between the anchor and the element under the mouse, inclusive.

Most people will probably want to use **browse** mode for single selections and **extended** mode for multiple selections; the other modes appear to be useful only in special situations.

In addition to the above behavior, the following additional behavior is defined by the default bindings:

[1] In **extended** mode, the selected range can be adjusted by pressing button 1 with the Shift key down: this modifies the selection to consist of the elements between the anchor and the element under the mouse, inclusive. The un-anchored end of this new selection can also be dragged with the button down.

[2] In **extended** mode, pressing button 1 with the Control key down starts a toggle operation: the anchor is set to the element under the mouse, and its selection state is reversed. The selection state of other elements isn't changed. If the mouse is dragged with button 1 down, then the selection state of all elements between the anchor and the element under the mouse is set to match that of the anchor element; the selection state of all other elements remains what it was before the toggle operation began.

[3] If the mouse leaves the listbox window with button 1 down, the window scrolls away from the mouse, making information visible that used to be off-screen on the side of the mouse. The scrolling continues until the mouse re-enters the window, the button is released, or the end of the listbox is reached.

[4] Mouse button 2 may be used for scanning. If it is pressed and dragged over the listbox, the contents of the listbox drag at high speed in the direction the mouse moves.

[5] If the Up or Down key is pressed, the location cursor (active element) moves up or down one

element. If the selection mode is **browse** or **extended** then the new active element is also selected and all other elements are deselected. In **extended** mode the new active element becomes the selection anchor.

[6] In **extended** mode, Shift-Up and Shift-Down move the location cursor (active element) up or down one element and also extend the selection to that element in a fashion similar to dragging with mouse button 1.

[7] The Left and Right keys scroll the listbox view left and right by the width of the character **0**. Control-Left and Control-Right scroll the listbox view left and right by the width of the window. Control-Prior and Control-Next also scroll left and right by the width of the window.

[8] The Prior and Next keys scroll the listbox view up and down by one page (the height of the window).

[9] The Home and End keys scroll the listbox horizontally to the left and right edges, respectively.

[10] Control-Home sets the location cursor to the the first element in the listbox, selects that element, and deselects everything else in the listbox.

[11] Control-End sets the location cursor to the the last element in the listbox, selects that element, and deselects everything else in the listbox.

[12] In **extended** mode, Control-Shift-Home extends the selection to the first element in the listbox and Control-Shift-End extends the selection to the last element.

[13] In **multiple** mode, Control-Shift-Home moves the location cursor to the first element in the listbox and Control-Shift-End moves the location cursor to the last element.

[14] The space and Select keys make a selection at the location cursor (active element) just as if mouse button 1 had been pressed over this element.

[15] In **extended** mode, Control-Shift-space and Shift-Select extend the selection to the active element just as if button 1 had been pressed with the Shift key down.

[16] In **extended** mode, the Escape key cancels the most recent selection and restores all the elements in the selected range to their previous selection state.

[17] Control-slash selects everything in the widget, except in **single** and **browse** modes, in which case it selects the active element and deselects everything else.

[18] Control-backslash deselects everything in the widget, except in **browse** mode where it has no effect.

[19] The F16 key (labelled Copy on many Sun workstations) or Meta-w copies the selection in the widget to the clipboard, if there is a selection.

The behavior of listboxes can be changed by defining new bindings for individual widgets or by redefining the class bindings.

KEYWORDS

listbox, widget

NAME

loadTk − Load Tk into a safe interpreter.

SYNOPSIS

::safe::loadTk *slave* ?**−use** *windowId*? ?**−display** *displayName*?

Safe Tk is based on Safe Tcl, which provides a mechanism that allows restricted and mediated access to auto-loading and packages for safe interpreters. Safe Tk adds the ability to configure the interpreter for safe Tk operations and load Tk into safe interpreters.

DESCRIPTION

The **::safe::loadTk** command initializes the required data structures in the named safe interpreter and then loads Tk into it. The command returns the name of the safe interpreter. If **−use** is specified, the window identified by the specified system dependent identifier *windowId* is used to contain the **“.”** window of the safe interpreter; it can be any valid id, eventually referencing a window belonging to another application. As a convenience, if the window you plan to use is a Tk Window of the application you can use the window name (eg: **.x.y**) instead of its window Id (**[winfo id .x.y]**). When **−use** is not specified, a new toplevel window is created for the **“.”** window of the safe interpreter. On X11 if you want the embedded window to use another display than the default one, specify it with **−display**. See the **SECURITY ISSUES** section below for implementation details.

SECURITY ISSUES

Please read the **safe** manual page for Tcl to learn about the basic security considerations for Safe Tcl.

::safe::loadTk adds the value of **tk_library** taken from the master interpreter to the virtual access path of the safe interpreter so that auto-loading will work in the safe interpreter.

Tk initialization is now safe with respect to not trusting the slave’s state for startup. **::safe::loadTk** registers the slave’s name so when the Tk initialization (**Tk_SafeInit**) is called and in turn calls the master’s **::safe::InitTk** it will return the desired **argv** equivalent (**−use** *windowId*, correct **−display**, etc...).

When **−use** is not used, the new toplevel created is specially decorated so the user is always aware that the user interface presented comes from a potentially unsafe code and can easily delete the corresponding interpreter.

On X11, conflicting **−use** and **−display** are likely to generate a fatal X error.

SEE ALSO

safe(n), interp(n), library(n), load(n), package(n), source(n), unknown(n)

KEYWORDS

alias, auto−loading, auto_mkindex, load, master interpreter, safe interpreter, slave interpreter, source

NAME

lower − Change a window’s position in the stacking order

SYNOPSIS

lower *window* ?*belowThis*?

DESCRIPTION

If the *belowThis* argument is omitted then the command lowers *window* so that it is below all of its siblings in the stacking order (it will be obscured by any siblings that overlap it and will not obscure any siblings). If *belowThis* is specified then it must be the path name of a window that is either a sibling of *window* or the descendant of a sibling of *window*. In this case the **lower** command will insert *window* into the stacking order just below *belowThis* (or the ancestor of *belowThis* that is a sibling of *window*); this could end up either raising or lowering *window*.

SEE ALSO

raise

KEYWORDS

lower, obscure, stacking order

NAME

menu − Create and manipulate menu widgets

SYNOPSIS

menu *pathName* ?*options*?

STANDARD OPTIONS

−activebackground	**−background**	**−disabledforeground**	**−relief**
−activeborderwidth	**−borderwidth**	**−font**	**−takefocus**
−activeforeground	**−cursor**	**−foreground**	

See the **options** manual entry for details on the standard options.

WIDGET-SPECIFIC OPTIONS

Command-Line Name: **−postcommand**
Database Name: **postCommand**
Database Class: **Command**

If this option is specified then it provides a Tcl command to execute each time the menu is posted. The command is invoked by the **post** widget command before posting the menu. Note that in 8.0 on Macintosh and Windows, all commands in a menu systems are executed before any are posted. This is due to the limitations in the individual platforms' menu managers.

Command-Line Name: **−selectcolor**
Database Name: **selectColor**
Database Class: **Background**

For menu entries that are check buttons or radio buttons, this option specifies the color to display in the indicator when the check button or radio button is selected.

Command-Line Name: **−tearoff**
Database Name: **tearOff**
Database Class: **TearOff**

This option must have a proper boolean value, which specifies whether or not the menu should include a tear-off entry at the top. If so, it will exist as entry 0 of the menu and the other entries will number starting at 1. The default menu bindings arrange for the menu to be torn off when the tear-off entry is invoked.

Command-Line Name: **−tearoffcommand**
Database Name: **tearOffCommand**
Database Class: **TearOffCommand**

If this option has a non-empty value, then it specifies a Tcl command to invoke whenever the menu is torn off. The actual command will consist of the value of this option, followed by a space, followed by the name of the menu window, followed by a space, followed by the name of the name of the torn off menu window. For example, if the option's is "**a b**" and menu **.x.y** is torn off to create a new menu **.x.tearoff1**, then the command "**a b .x.y .x.tearoff1**" will be invoked.

Command-Line Name: **−title**
Database Name: **title**
Database Class: **Title**

The string will be used to title the window created when this menu is torn off. If the title is NULL, then the window will have the title of the menubutton or the text of the cascade item from which this menu was invoked.

Command-Line Name: **−type**
Database Name: **type**
Database Class: **Type**

This option can be one of **menubar**, **tearoff**, or **normal**, and is set when the menu is created. While the string returned by the configuration database will change if this option is changed, this does not affect the menu widget's behavior. This is used by the cloning mechanism and is not normally set outside of the Tk library.

INTRODUCTION

The **menu** command creates a new top-level window (given by the *pathName* argument) and makes it into a menu widget. Additional options, described above, may be specified on the command line or in the option database to configure aspects of the menu such as its colors and font. The **menu** command returns its *pathName* argument. At the time this command is invoked, there must not exist a window named *pathName*, but *pathName*'s parent must exist.

A menu is a widget that displays a collection of one-line entries arranged in one or more columns. There exist several different types of entries, each with different properties. Entries of different types may be combined in a single menu. Menu entries are not the same as entry widgets. In fact, menu entries are not even distinct widgets; the entire menu is one widget.

Menu entries are displayed with up to three separate fields. The main field is a label in the form of a text string, a bitmap, or an image, controlled by the **−label**, **−bitmap**, and **−image** options for the entry. If the **−accelerator** option is specified for an entry then a second textual field is displayed to the right of the label. The accelerator typically describes a keystroke sequence that may be typed in the application to cause the same result as invoking the menu entry. The third field is an *indicator*. The indicator is present only for checkbutton or radiobutton entries. It indicates whether the entry is selected or not, and is displayed to the left of the entry's string.

In normal use, an entry becomes active (displays itself differently) whenever the mouse pointer is over the entry. If a mouse button is released over the entry then the entry is *invoked*. The effect of invocation is different for each type of entry; these effects are described below in the sections on individual entries.

Entries may be *disabled*, which causes their labels and accelerators to be displayed with dimmer colors. The default menu bindings will not allow a disabled entry to be activated or invoked. Disabled entries may be re-enabled, at which point it becomes possible to activate and invoke them again.

Whenever a menu's active entry is changed, a <<MenuSelect>> virtual event is send to the menu. The active item can then be queried from the menu, and an action can be taken, such as setting context-sensitive help text for the entry.

COMMAND ENTRIES

The most common kind of menu entry is a command entry, which behaves much like a button widget. When a command entry is invoked, a Tcl command is executed. The Tcl command is specified with the **−command** option.

SEPARATOR ENTRIES

A separator is an entry that is displayed as a horizontal dividing line. A separator may not be activated or invoked, and it has no behavior other than its display appearance.

CHECKBUTTON ENTRIES

A checkbutton menu entry behaves much like a checkbutton widget. When it is invoked it toggles back and forth between the selected and deselected states. When the entry is selected, a particular value is stored in a particular global variable (as determined by the **−onvalue** and **−variable** options for the entry); when the entry is deselected another value (determined by the **−offvalue** option) is stored in the global variable. An indicator box is displayed to the left of the label in a checkbutton entry. If the entry is selected then the

indicator's center is displayed in the color given by the **-selectcolor** option for the entry; otherwise the indicator's center is displayed in the background color for the menu. If a **−command** option is specified for a checkbutton entry, then its value is evaluated as a Tcl command each time the entry is invoked; this happens after toggling the entry's selected state.

RADIOBUTTON ENTRIES

A radiobutton menu entry behaves much like a radiobutton widget. Radiobutton entries are organized in groups of which only one entry may be selected at a time. Whenever a particular entry becomes selected it stores a particular value into a particular global variable (as determined by the **−value** and **−variable** options for the entry). This action causes any previously-selected entry in the same group to deselect itself. Once an entry has become selected, any change to the entry's associated variable will cause the entry to deselect itself. Grouping of radiobutton entries is determined by their associated variables: if two entries have the same associated variable then they are in the same group. An indicator diamond is displayed to the left of the label in each radiobutton entry. If the entry is selected then the indicator's center is displayed in the color given by the **−selectcolor** option for the entry; otherwise the indicator's center is displayed in the background color for the menu. If a **−command** option is specified for a radiobutton entry, then its value is evaluated as a Tcl command each time the entry is invoked; this happens after selecting the entry.

CASCADE ENTRIES

A cascade entry is one with an associated menu (determined by the **−menu** option). Cascade entries allow the construction of cascading menus. The **postcascade** widget command can be used to post and unpost the associated menu just next to of the cascade entry. The associated menu must be a child of the menu containing the cascade entry (this is needed in order for menu traversal to work correctly).

A cascade entry posts its associated menu by invoking a Tcl command of the form

menu **post** *x y*

where *menu* is the path name of the associated menu, and *x* and *y* are the root-window coordinates of the upper-right corner of the cascade entry. On Unix, the lower-level menu is unposted by executing a Tcl command with the form

menu **unpost**

where *menu* is the name of the associated menu. On other platforms, the platform's native code takes care of unposting the menu.

If a **−command** option is specified for a cascade entry then it is evaluated as a Tcl command whenever the entry is invoked. This is not supported on Windows.

TEAR-OFF ENTRIES

A tear-off entry appears at the top of the menu if enabled with the **tearOff** option. It is not like other menu entries in that it cannot be created with the **add** widget command and cannot be deleted with the **delete** widget command. When a tear-off entry is created it appears as a dashed line at the top of the menu. Under the default bindings, invoking the tear-off entry causes a torn-off copy to be made of the menu and all of its submenus.

MENUBARS

Any menu can be set as a menubar for a toplevel window (see **toplevel** command for syntax). On the Macintosh, whenever the toplevel is in front, this menu's cascade items will appear in the menubar across the top of the main monitor. On Windows and Unix, this menu's items will be displayed in a menubar accross the top of the window. These menus will behave according to the interface guidelines of their platforms. For every menu set as a menubar, a clone menu is made. See the **CLONES** section for more information.

SPECIAL MENUS IN MENUBARS

Certain menus in a menubar will be treated specially. On the Macintosh, access to the special Apple and Help menus is provided. On Windows, access to the Windows System menu in each window is provided. On X Windows, a special right-justified help menu is provided. In all cases, these menus must be created with the command name of the menubar menu concatenated with the special name. So for a menubar named .menubar, on the Macintosh, the special menus would be .menubar.apple and .menubar.help; on Windows, the special menu would be .menubar.system; on X Windows, the help menu would be .menubar.help.

When Tk sees an Apple menu on the Macintosh, that menu's contents make up the first items of the Apple menu on the screen whenever the window containing the menubar is in front. The menu is the first one that the user sees and has a title which is an Apple logo. After all of the Tk-defined items, the menu will have a separator, followed by all of the items in the user's Apple Menu Items folder. Since the System uses a different menu definition procedure for the Apple menu than Tk uses for its menus, and the system APIs do not fully support everything Tk tries to do, the menu item will only have its text displayed. No font attributes, images, bitmaps, or colors will be displayed. In addition, a menu with a tearoff item will have the tearoff item displayed as "(TearOff)".

When Tk see a Help menu on the Macintosh, the menu's contents are appended to the standard help menu on the right of the user's menubar whenever the user's menubar is in front. The first items in the menu are provided by Apple. Similar to the Apple Menu, cusomization in this menu is limited to what the system provides.

When Tk sees a System menu on Windows, its items are appended to the system menu that the menubar is attached to. This menu has an icon representing a spacebar, and can be invoked with the mouse or by typing Alt+Spacebar. Due to limitations in the Windows API, any font changes, colors, images, bitmaps, or tearoff images will not appear in the system menu.

When Tk see a Help menu on X Windows, the menu is moved to be last in the menubar and is right justified.

CLONES

When a menu is set as a menubar for a toplevel window, or when a menu is torn off, a clone of the menu is made. This clone is a menu widget in its own right, but it is a child of the original. Changes in the configuration of the original are reflected in the clone. Additionally, any cascades that are pointed to are also cloned so that menu traversal will work right. Clones are destroyed when either the tearoff or menubar goes away, or when the original menu is destroyed.

WIDGET COMMAND

The **menu** command creates a new Tcl command whose name is *pathName*. This command may be used to invoke various operations on the widget. It has the following general form:

pathName option ?*arg arg ...*?

Option and the *arg*s determine the exact behavior of the command.

Many of the widget commands for a menu take as one argument an indicator of which entry of the menu to operate on. These indicators are called *index*es and may be specified in any of the following forms:

number Specifies the entry numerically, where 0 corresponds to the top-most entry of the menu, 1 to the entry below it, and so on.

active Indicates the entry that is currently active. If no entry is active then this form is equivalent to **none**. This form may not be abbreviated.

end Indicates the bottommost entry in the menu. If there are no entries in the menu then this form is equivalent to **none**. This form may not be abbreviated.

last Same as **end**.

none Indicates ‘‘no entry at all’’; this is used most commonly with the **activate** option to deactivate all the entries in the menu. In most cases the specification of **none** causes nothing to happen in the widget command. This form may not be abbreviated.

@number In this form, *number* is treated as a y-coordinate in the menu’s window; the entry closest to that y-coordinate is used. For example, ‘‘**@0**’’ indicates the top-most entry in the window.

pattern If the index doesn’t satisfy one of the above forms then this form is used. *Pattern* is pattern-matched against the label of each entry in the menu, in order from the top down, until a matching entry is found. The rules of **Tcl_StringMatch** are used.

The following widget commands are possible for menu widgets:

pathName **activate** *index*

Change the state of the entry indicated by *index* to **active** and redisplay it using its active colors. Any previously-active entry is deactivated. If *index* is specified as **none**, or if the specified entry is disabled, then the menu ends up with no active entry. Returns an empty string.

pathName **add** *type* ?*option value option value ...*?

Add a new entry to the bottom of the menu. The new entry’s type is given by *type* and must be one of **cascade**, **checkbutton**, **command**, **radiobutton**, or **separator**, or a unique abbreviation of one of the above. If additional arguments are present, they specify any of the following options:

−activebackground *value*

Specifies a background color to use for displaying this entry when it is active. If this option is specified as an empty string (the default), then the **activeBackground** option for the overall menu is used. If the **tk_strictMotif** variable has been set to request strict Motif compliance, then this option is ignored and the **−background** option is used in its place. This option is not available for separator or tear-off entries.

−activeforeground *value*

Specifies a foreground color to use for displaying this entry when it is active. If this option is specified as an empty string (the default), then the **activeForeground** option for the overall menu is used. This option is not available for separator or tear-off entries.

−accelerator *value*

Specifies a string to display at the right side of the menu entry. Normally describes an accelerator keystroke sequence that may be typed to invoke the same function as the menu entry. This option is not available for separator or tear-off entries.

−background *value*

Specifies a background color to use for displaying this entry when it is in the normal state (neither active nor disabled). If this option is specified as an empty string (the default), then the **background** option for the overall menu is used. This option is not available for separator or tear-off entries.

−bitmap *value*

Specifies a bitmap to display in the menu instead of a textual label, in any of the forms accepted by **Tk_GetBitmap**. This option overrides the **−label** option but may be reset to an empty string to enable a textual label to be displayed. If a **−image** option has been specified, it overrides **−bitmap**. This option is not available for separator or tear-off entries.

−columnbreak *value*

When this option is zero, the appears below the previous entry. When this option is one, the menu appears at the top of a new column in the menu.

−command *value*
Specifies a Tcl command to execute when the menu entry is invoked. Not available for separator or tear-off entries.

−font *value*
Specifies the font to use when drawing the label or accelerator string in this entry. If this option is specified as an empty string (the default) then the **font** option for the overall menu is used. This option is not available for separator or tear-off entries.

−foreground *value*
Specifies a foreground color to use for displaying this entry when it is in the normal state (neither active nor disabled). If this option is specified as an empty string (the default), then the **foreground** option for the overall menu is used. This option is not available for separator or tear-off entries.

−hidemargin *value*
Specifies whether the standard margins should be drawn for this menu entry. This is useful when creating palette with images in them, i.e., color palettes, pattern palettes, etc. 1 indicates that the margin for the entry is hidden; 0 means that the margin is used.

−image *value*
Specifies an image to display in the menu instead of a text string or bitmap The image must have been created by some previous invocation of **image create**. This option overrides the **−label** and **−bitmap** options but may be reset to an empty string to enable a textual or bitmap label to be displayed. This option is not available for separator or tear-off entries.

−indicatoron *value*
Available only for checkbutton and radiobutton entries. *Value* is a boolean that determines whether or not the indicator should be displayed.

−label *value*
Specifies a string to display as an identifying label in the menu entry. Not available for separator or tear-off entries.

−menu *value*
Available only for cascade entries. Specifies the path name of the submenu associated with this entry. The submenu must be a child of the menu.

−offvalue *value*
Available only for checkbutton entries. Specifies the value to store in the entry's associated variable when the entry is deselected.

−onvalue *value*
Available only for checkbutton entries. Specifies the value to store in the entry's associated variable when the entry is selected.

−selectcolor *value*
Available only for checkbutton and radiobutton entries. Specifies the color to display in the indicator when the entry is selected. If the value is an empty string (the default) then the **selectColor** option for the menu determines the indicator color.

−selectimage *value*
Available only for checkbutton and radiobutton entries. Specifies an image to display in the entry (in place of the **−image** option) when it is selected. *Value* is the name of an image, which must have been created by some previous invocation of **image create**. This option is ignored unless the **−image** option has been specified.

−state *value*

Specifies one of three states for the entry: **normal**, **active**, or **disabled**. In normal state the entry is displayed using the **foreground** option for the menu and the **background** option from the entry or the menu. The active state is typically used when the pointer is over the entry. In active state the entry is displayed using the **activeForeground** option for the menu along with the **activebackground** option from the entry. Disabled state means that the entry should be insensitive: the default bindings will refuse to activate or invoke the entry. In this state the entry is displayed according to the **disabledForeground** option for the menu and the **background** option from the entry. This option is not available for separator entries.

−underline *value*

Specifies the integer index of a character to underline in the entry. This option is also queried by the default bindings and used to implement keyboard traversal. 0 corresponds to the first character of the text displayed in the entry, 1 to the next character, and so on. If a bitmap or image is displayed in the entry then this option is ignored. This option is not available for separator or tear-off entries.

−value *value*

Available only for radiobutton entries. Specifies the value to store in the entry's associated variable when the entry is selected. If an empty string is specified, then the **−label** option for the entry as the value to store in the variable.

−variable *value*

Available only for checkbutton and radiobutton entries. Specifies the name of a global value to set when the entry is selected. For checkbutton entries the variable is also set when the entry is deselected. For radiobutton entries, changing the variable causes the currently-selected entry to deselect itself.

The **add** widget command returns an empty string.

pathName **cget** *option*

Returns the current value of the configuration option given by *option*. *Option* may have any of the values accepted by the **menu** command.

pathName **clone** *newPathname ?cloneType?*

Makes a clone of the current menu named *newPathName*. This clone is a menu in its own right, but any changes to the clone are propogated to the original menu and vice versa. *cloneType* can be **normal**, **menubar**, or **tearoff**. Should not normally be called outside of the Tk library. See the **CLONES** section for more information.

pathName **configure** ?*option*? ?*value option value ...*?

Query or modify the configuration options of the widget. If no *option* is specified, returns a list describing all of the available options for *pathName* (see **Tk_ConfigureInfo** for information on the format of this list). If *option* is specified with no *value*, then the command returns a list describing the one named option (this list will be identical to the corresponding sublist of the value returned if no *option* is specified). If one or more *option−value* pairs are specified, then the command modifies the given widget option(s) to have the given value(s); in this case the command returns an empty string. *Option* may have any of the values accepted by the **menu** command.

pathName **delete** *index1* ?*index2*?

Delete all of the menu entries between *index1* and *index2* inclusive. If *index2* is omitted then it defaults to *index1*. Attempts to delete a tear-off menu entry are ignored (instead, you should change the **tearOff** option to remove the tear-off entry).

pathName **entrycget** *index option*

Returns the current value of a configuration option for the entry given by *index*. *Option* may have any of the values accepted by the **add** widget command.

pathName **entryconfigure** *index* ?*options*?
This command is similar to the **configure** command, except that it applies to the options for an individual entry, whereas **configure** applies to the options for the menu as a whole. *Options* may have any of the values accepted by the **add** widget command. If *options* are specified, options are modified as indicated in the command and the command returns an empty string. If no *options* are specified, returns a list describing the current options for entry *index* (see **Tk_ConfigureInfo** for information on the format of this list).

pathName **index** *index*
Returns the numerical index corresponding to *index*, or **none** if *index* was specified as **none**.

pathName **insert** *index type* ?*option value option value ...*?
Same as the **add** widget command except that it inserts the new entry just before the entry given by *index*, instead of appending to the end of the menu. The *type*, *option*, and *value* arguments have the same interpretation as for the **add** widget command. It is not possible to insert new menu entries before the tear-off entry, if the menu has one.

pathName **invoke** *index*
Invoke the action of the menu entry. See the sections on the individual entries above for details on what happens. If the menu entry is disabled then nothing happens. If the entry has a command associated with it then the result of that command is returned as the result of the **invoke** widget command. Otherwise the result is an empty string. Note: invoking a menu entry does not automatically unpost the menu; the default bindings normally take care of this before invoking the **invoke** widget command.

pathName **post** *x y*
Arrange for the menu to be displayed on the screen at the root-window coordinates given by *x* and *y*. These coordinates are adjusted if necessary to guarantee that the entire menu is visible on the screen. This command normally returns an empty string. If the **postCommand** option has been specified, then its value is executed as a Tcl script before posting the menu and the result of that script is returned as the result of the **post** widget command. If an error returns while executing the command, then the error is returned without posting the menu.

pathName **postcascade** *index*
Posts the submenu associated with the cascade entry given by *index*, and unposts any previously posted submenu. If *index* doesn't correspond to a cascade entry, or if *pathName* isn't posted, the command has no effect except to unpost any currently posted submenu.

pathName **type** *index*
Returns the type of the menu entry given by *index*. This is the *type* argument passed to the **add** widget command when the entry was created, such as **command** or **separator**, or **tearoff** for a tear-off entry.

pathName **unpost**
Unmap the window so that it is no longer displayed. If a lower-level cascaded menu is posted, unpost that menu. Returns an empty string. This subcommand does not work on Windows and the Macintosh, as those platforms have their own way of unposting menus.

pathName **yposition** *index*
Returns a decimal string giving the y-coordinate within the menu window of the topmost pixel in the entry specified by *index*.

MENU CONFIGURATIONS

The default bindings support four different ways of using menus:

Pulldown Menus in Menubar
This is the most command case. You create a menu widget that will become the menu bar. You

then add cascade entries to this menu, specifying the pull down menus you wish to use in your menu bar. You then create all of the pulldowns. Once you have done this, specify the menu using the **-menu** option of the toplevel's widget command. See the **toplevel** manual entry for details.

Pulldown Menus in Menu Buttons

This is the compatable way to do menu bars. You create one menubutton widget for each top-level menu, and typically you arrange a series of menubuttons in a row in a menubar window. You also create the top-level menus and any cascaded submenus, and tie them together with **−menu** options in menubuttons and cascade menu entries. The top-level menu must be a child of the menubutton, and each submenu must be a child of the menu that refers to it. Once you have done this, the default bindings will allow users to traverse and invoke the tree of menus via its menubutton; see the **menubutton** manual entry for details.

Popup Menus

Popup menus typically post in response to a mouse button press or keystroke. You create the popup menus and any cascaded submenus, then you call the **tk_popup** procedure at the appropriate time to post the top-level menu.

Option Menus

An option menu consists of a menubutton with an associated menu that allows you to select one of several values. The current value is displayed in the menubutton and is also stored in a global variable. Use the **tk_optionMenu** procedure to create option menubuttons and their menus.

Torn-off Menus

You create a torn-off menu by invoking the tear-off entry at the top of an existing menu. The default bindings will create a new menu that is a copy of the original menu and leave it permanently posted as a top-level window. The torn-off menu behaves just the same as the original menu.

DEFAULT BINDINGS

Tk automatically creates class bindings for menus that give them the following default behavior:

[1] When the mouse enters a menu, the entry underneath the mouse cursor activates; as the mouse moves around the menu, the active entry changes to track the mouse.

[2] When the mouse leaves a menu all of the entries in the menu deactivate, except in the special case where the mouse moves from a menu to a cascaded submenu.

[3] When a button is released over a menu, the active entry (if any) is invoked. The menu also unposts unless it is a torn-off menu.

[4] The Space and Return keys invoke the active entry and unpost the menu.

[5] If any of the entries in a menu have letters underlined with with **−underline** option, then pressing one of the underlined letters (or its upper-case or lower-case equivalent) invokes that entry and unposts the menu.

[6] The Escape key aborts a menu selection in progress without invoking any entry. It also unposts the menu unless it is a torn-off menu.

[7] The Up and Down keys activate the next higher or lower entry in the menu. When one end of the menu is reached, the active entry wraps around to the other end.

[8] The Left key moves to the next menu to the left. If the current menu is a cascaded submenu, then the submenu is unposted and the current menu entry becomes the cascade entry in the parent. If the current menu is a top-level menu posted from a menubutton, then the current menubutton is unposted and the next menubutton to the left is posted. Otherwise the key has no effect. The left-right order of menubuttons is determined by their stacking order: Tk assumes that the lowest

menubutton (which by default is the first one created) is on the left.

[9] The Right key moves to the next menu to the right. If the current entry is a cascade entry, then the submenu is posted and the current menu entry becomes the first entry in the submenu. Otherwise, if the current menu was posted from a menubutton, then the current menubutton is unposted and the next menubutton to the right is posted.

Disabled menu entries are non-responsive: they don't activate and they ignore mouse button presses and releases.

The behavior of menus can be changed by defining new bindings for individual widgets or by redefining the class bindings.

BUGS

At present it isn't possible to use the option database to specify values for the options to individual entries.

KEYWORDS

menu, widget

NAME

tk_menuBar, tk_bindForTraversal − Obsolete support for menu bars

SYNOPSIS

tk_menuBar *frame* ?*menu menu ...*?

tk_bindForTraversal *arg arg ...*

DESCRIPTION

These procedures were used in Tk 3.6 and earlier releases to help manage pulldown menus and to implement keyboard traversal of menus. In Tk 4.0 and later releases they are no longer needed. Stubs for these procedures have been retained for backward compatibility, but they have no effect. You should remove calls to these procedures from your code, since eventually the procedures will go away.

KEYWORDS

keyboard traversal, menu, menu bar, post

NAME

menubutton − Create and manipulate menubutton widgets

SYNOPSIS

menubutton *pathName* ?*options*?

STANDARD OPTIONS

−activebackground	**−cursor**	**−highlightthickness**	**−takefocus**
−activeforeground	**−disabledforeground**	**−image**	**−text**
−anchor	**−font**	**−justify**	**−textvariable**
−background	**−foreground**	**−padx**	**−underline**
−bitmap	**−highlightbackground**	**−pady**	**−wraplength**
−borderwidth	**−highlightcolor**	**−relief**	

See the **options** manual entry for details on the standard options.

WIDGET-SPECIFIC OPTIONS

Command-Line Name: **−direction**
Database Name: **direction**
Database Class: **Height**

Specifies where the menu is going to be popup up. **above** tries to pop the menu above the menubutton. **below** tries to pop the menu below the menubutton. **left** tries to pop the menu to the left of the menubutton. **right** tries to pop the menu to the right of the menu button. **flush** pops the menu directly over the menubutton.

Command-Line Name: **−height**
Database Name: **height**
Database Class: **Height**

Specifies a desired height for the menubutton. If an image or bitmap is being displayed in the menubutton then the value is in screen units (i.e. any of the forms acceptable to **Tk_GetPixels**); for text it is in lines of text. If this option isn't specified, the menubutton's desired height is computed from the size of the image or bitmap or text being displayed in it.

Command-Line Name: **−indicatoron**
Database Name: **indicatorOn**
Database Class: **IndicatorOn**

The value must be a proper boolean value. If it is true then a small indicator rectangle will be displayed on the right side of the menubutton and the default menu bindings will treat this as an option menubutton. If false then no indicator will be displayed.

Command-Line Name: **−menu**
Database Name: **menu**
Database Class: **MenuName**

Specifies the path name of the menu associated with this menubutton. The menu must be a child of the menubutton.

Command-Line Name: **−state**
Database Name: **state**
Database Class: **State**

Specifies one of three states for the menubutton: **normal**, **active**, or **disabled**. In normal state the menubutton is displayed using the **foreground** and **background** options. The active state is typically used when the pointer is over the menubutton. In active state the menubutton is displayed using the **activeForeground** and **activeBackground** options. Disabled state means that the menubutton should be insensitive: the default bindings will refuse to activate the widget and will

ignore mouse button presses. In this state the **disabledForeground** and **background** options determine how the button is displayed.

Command-Line Name: **−width**
Database Name: **width**
Database Class: **Width**

Specifies a desired width for the menubutton. If an image or bitmap is being displayed in the menubutton then the value is in screen units (i.e. any of the forms acceptable to **Tk_GetPixels**); for text it is in characters. If this option isn't specified, the menubutton's desired width is computed from the size of the image or bitmap or text being displayed in it.

INTRODUCTION

The **menubutton** command creates a new window (given by the *pathName* argument) and makes it into a menubutton widget. Additional options, described above, may be specified on the command line or in the option database to configure aspects of the menubutton such as its colors, font, text, and initial relief. The **menubutton** command returns its *pathName* argument. At the time this command is invoked, there must not exist a window named *pathName*, but *pathName*'s parent must exist.

A menubutton is a widget that displays a textual string, bitmap, or image and is associated with a menu widget. If text is displayed, it must all be in a single font, but it can occupy multiple lines on the screen (if it contains newlines or if wrapping occurs because of the **wrapLength** option) and one of the characters may optionally be underlined using the **underline** option. In normal usage, pressing mouse button 1 over the menubutton causes the associated menu to be posted just underneath the menubutton. If the mouse is moved over the menu before releasing the mouse button, the button release causes the underlying menu entry to be invoked. When the button is released, the menu is unposted.

Menubuttons are typically organized into groups called menu bars that allow scanning: if the mouse button is pressed over one menubutton (causing it to post its menu) and the mouse is moved over another menubutton in the same menu bar without releasing the mouse button, then the menu of the first menubutton is unposted and the menu of the new menubutton is posted instead.

There are several interactions between menubuttons and menus; see the **menu** manual entry for information on various menu configurations, such as pulldown menus and option menus.

WIDGET COMMAND

The **menubutton** command creates a new Tcl command whose name is *pathName*. This command may be used to invoke various operations on the widget. It has the following general form:

pathName option ?*arg arg ...*?

Option and the *arg*s determine the exact behavior of the command. The following commands are possible for menubutton widgets:

pathName **cget** *option*

Returns the current value of the configuration option given by *option*. *Option* may have any of the values accepted by the **menubutton** command.

pathName **configure** ?*option*? ?*value option value ...*?

Query or modify the configuration options of the widget. If no *option* is specified, returns a list describing all of the available options for *pathName* (see **Tk_ConfigureInfo** for information on the format of this list). If *option* is specified with no *value*, then the command returns a list describing the one named option (this list will be identical to the corresponding sublist of the value returned if no *option* is specified). If one or more *option−value* pairs are specified, then the command modifies the given widget option(s) to have the given value(s); in this case the command returns an empty string. *Option* may have any of the values accepted by the **menubutton**

command.

DEFAULT BINDINGS

Tk automatically creates class bindings for menubuttons that give them the following default behavior:

[1] A menubutton activates whenever the mouse passes over it and deactivates whenever the mouse leaves it.

[2] Pressing mouse button 1 over a menubutton posts the menubutton: its relief changes to raised and its associated menu is posted under the menubutton. If the mouse is dragged down into the menu with the button still down, and if the mouse button is then released over an entry in the menu, the menubutton is unposted and the menu entry is invoked.

[3] If button 1 is pressed over a menubutton and then released over that menubutton, the menubutton stays posted: you can still move the mouse over the menu and click button 1 on an entry to invoke it. Once a menu entry has been invoked, the menubutton unposts itself.

[4] If button 1 is pressed over a menubutton and then dragged over some other menubutton, the original menubutton unposts itself and the new menubutton posts.

[5] If button 1 is pressed over a menubutton and released outside any menubutton or menu, the menubutton unposts without invoking any menu entry.

[6] When a menubutton is posted, its associated menu claims the input focus to allow keyboard traversal of the menu and its submenus. See the **menu** manual entry for details on these bindings.

[7] If the **underline** option has been specified for a menubutton then keyboard traversal may be used to post the menubutton: Alt+*x*, where *x* is the underlined character (or its lower-case or upper-case equivalent), may be typed in any window under the menubutton's toplevel to post the menubutton.

[8] The F10 key may be typed in any window to post the first menubutton under its toplevel window that isn't disabled.

[9] If a menubutton has the input focus, the space and return keys post the menubutton.

If the menubutton's state is **disabled** then none of the above actions occur: the menubutton is completely non-responsive.

The behavior of menubuttons can be changed by defining new bindings for individual widgets or by redefining the class bindings.

KEYWORDS

menubutton, widget

NAME

message − Create and manipulate message widgets

SYNOPSIS

message *pathName* ?*options*?

STANDARD OPTIONS

−anchor	**−font**	**−highlightthickness**	**−takefocus**
−background	**−foreground**	**−padx**	**−text**
−borderwidth	**−highlightbackground**	**−pady**	**−textvariable**
−cursor	**−highlightcolor**	**−relief**	**−width**

See the **options** manual entry for details on the standard options.

WIDGET-SPECIFIC OPTIONS

Command-Line Name: **−aspect**
Database Name: **aspect**
Database Class: **Aspect**

Specifies a non-negative integer value indicating desired aspect ratio for the text. The aspect ratio is specified as 100*width/height. 100 means the text should be as wide as it is tall, 200 means the text should be twice as wide as it is tall, 50 means the text should be twice as tall as it is wide, and so on. Used to choose line length for text if **width** option isn't specified. Defaults to 150.

Command-Line Name: **−justify**
Database Name: **justify**
Database Class: **Justify**

Specifies how to justify lines of text. Must be one of **left**, **center**, or **right**. Defaults to **left**. This option works together with the **anchor**, **aspect**, **padX**, **padY**, and **width** options to provide a variety of arrangements of the text within the window. The **aspect** and **width** options determine the amount of screen space needed to display the text. The **anchor**, **padX**, and **padY** options determine where this rectangular area is displayed within the widget's window, and the **justify** option determines how each line is displayed within that rectangular region. For example, suppose **anchor** is **e** and **justify** is **left**, and that the message window is much larger than needed for the text. The the text will displayed so that the left edges of all the lines line up and the right edge of the longest line is **padX** from the right side of the window; the entire text block will be centered in the vertical span of the window.

Command-Line Name: **−width**
Database Name: **width**
Database Class: **Width**

Specifies the length of lines in the window. The value may have any of the forms acceptable to **Tk_GetPixels**. If this option has a value greater than zero then the **aspect** option is ignored and the **width** option determines the line length. If this option has a value less than or equal to zero, then the **aspect** option determines the line length.

DESCRIPTION

The **message** command creates a new window (given by the *pathName* argument) and makes it into a message widget. Additional options, described above, may be specified on the command line or in the option database to configure aspects of the message such as its colors, font, text, and initial relief. The **message** command returns its *pathName* argument. At the time this command is invoked, there must not exist a window named *pathName*, but *pathName*'s parent must exist.

A message is a widget that displays a textual string. A message widget has three special features. First, it breaks up its string into lines in order to produce a given aspect ratio for the window. The line breaks are chosen at word boundaries wherever possible (if not even a single word would fit on a line, then the word will be split across lines). Newline characters in the string will force line breaks; they can be used, for example, to leave blank lines in the display.

The second feature of a message widget is justification. The text may be displayed left-justified (each line starts at the left side of the window), centered on a line-by-line basis, or right-justified (each line ends at the right side of the window).

The third feature of a message widget is that it handles control characters and non-printing characters specially. Tab characters are replaced with enough blank space to line up on the next 8-character boundary. Newlines cause line breaks. Other control characters (ASCII code less than 0x20) and characters not defined in the font are displayed as a four-character sequence **x***hh* where *hh* is the two-digit hexadecimal number corresponding to the character. In the unusual case where the font doesn't contain all of the characters in ''0123456789abcdef\x'' then control characters and undefined characters are not displayed at all.

WIDGET COMMAND

The **message** command creates a new Tcl command whose name is *pathName*. This command may be used to invoke various operations on the widget. It has the following general form:

pathName option ?*arg arg ...*?

Option and the *arg*s determine the exact behavior of the command. The following commands are possible for message widgets:

pathName **cget** *option*

Returns the current value of the configuration option given by *option*. *Option* may have any of the values accepted by the **message** command.

pathName **configure** ?*option*? ?*value option value ...*?

Query or modify the configuration options of the widget. If no *option* is specified, returns a list describing all of the available options for *pathName* (see **Tk_ConfigureInfo** for information on the format of this list). If *option* is specified with no *value*, then the command returns a list describing the one named option (this list will be identical to the corresponding sublist of the value returned if no *option* is specified). If one or more *option*−*value* pairs are specified, then the command modifies the given widget option(s) to have the given value(s); in this case the command returns an empty string. *Option* may have any of the values accepted by the **message** command.

DEFAULT BINDINGS

When a new message is created, it has no default event bindings: messages are intended for output purposes only.

BUGS

Tabs don't work very well with text that is centered or right-justified. The most common result is that the line is justified wrong.

KEYWORDS

message, widget

NAME

tk_messageBox − pops up a message window and waits for user response.

SYNOPSIS

tk_messageBox ?*option value ...*?

DESCRIPTION

This procedure creates and displays a message window with an application-specified message, an icon and a set of buttons. Each of the buttons in the message window is identified by a unique symbolic name (see the **−type** options). After the message window is popped up, **tk_messageBox** waits for the user to select one of the buttons. Then it returns the symbolic name of the selected button.

The following option-value pairs are supported:

−default *name*

Name gives the symbolic name of the default button for this message window (’ok’, ’cancel’, and so on). See **−type** for a list of the symbolic names. If the message box has just one button it will automatically be made the default, otherwise if this option is not specified, there won’t be any default button.

−icon *iconImage*

Specifies an icon to display. *IconImage* must be one of the following: **error**, **info**, **question** or **warning**. If this option is not specified, then no icon will be displayed.

−message *string*

Specifies the message to display in this message box.

−parent *window*

Makes *window* the logical parent of the message box. The message box is displayed on top of its parent window.

−title *titleString*

Specifies a string to display as the title of the message box. The default value is an empty string.

−type *predefinedType*

Arranges for a predefined set of buttons to be displayed. The following values are possible for *predefinedType*:

abortretryignore	Displays three buttons whose symbolic names are **abort**, **retry** and **ignore**.
ok	Displays one button whose symbolic name is **ok**.
okcancel	Displays two buttons whose symbolic names are **ok** and **cancel**.
retrycancel	Displays two buttons whose symbolic names are **retry** and **cancel**.
yesno	Displays two buttons whose symbolic names are **yes** and **no**.
yesnocancel	Displays three buttons whose symbolic names are **yes**, **no** and **cancel**.

EXAMPLE

```
set answer [tk_messageBox −message "Really quit?" −type yesno −icon question]
case $answer {
   yes exit
   no {tk_messageBox −message "I know you like this application!" −type ok}
}
```

KEYWORDS

message box

NAME

option − Add/retrieve window options to/from the option database

SYNOPSIS

option add *pattern value* ?*priority*?

option clear

option get *window name class*

option readfile *fileName* ?*priority*?

DESCRIPTION

The **option** command allows you to add entries to the Tk option database or to retrieve options from the database. The **add** form of the command adds a new option to the database. *Pattern* contains the option being specified, and consists of names and/or classes separated by asterisks or dots, in the usual X format. *Value* contains a text string to associate with *pattern*; this is the value that will be returned in calls to **Tk_GetOption** or by invocations of the **option get** command. If *priority* is specified, it indicates the priority level for this option (see below for legal values); it defaults to **interactive**. This command always returns an empty string.

The **option clear** command clears the option database. Default options (from the **RESOURCE_MANAGER** property or the **.Xdefaults** file) will be reloaded automatically the next time an option is added to the database or removed from it. This command always returns an empty string.

The **option get** command returns the value of the option specified for *window* under *name* and *class*. If several entries in the option database match *window*, *name*, and *class*, then the command returns whichever was created with highest *priority* level. If there are several matching entries at the same priority level, then it returns whichever entry was most recently entered into the option database. If there are no matching entries, then the empty string is returned.

The **readfile** form of the command reads *fileName*, which should have the standard format for an X resource database such as **.Xdefaults**, and adds all the options specified in that file to the option database. If *priority* is specified, it indicates the priority level at which to enter the options; *priority* defaults to **interactive**.

The *priority* arguments to the **option** command are normally specified symbolically using one of the following values:

widgetDefault
Level 20. Used for default values hard-coded into widgets.

startupFile
Level 40. Used for options specified in application-specific startup files.

userDefault
Level 60. Used for options specified in user-specific defaults files, such as **.Xdefaults**, resource databases loaded into the X server, or user-specific startup files.

interactive
Level 80. Used for options specified interactively after the application starts running. If *priority* isn't specified, it defaults to this level.

Any of the above keywords may be abbreviated. In addition, priorities may be specified numerically using integers between 0 and 100, inclusive. The numeric form is probably a bad idea except for new priority levels other than the ones given above.

KEYWORDS

database, option, priority, retrieve

NAME

tk_optionMenu − Create an option menubutton and its menu

SYNOPSIS

tk_optionMenu *w varName value* ?*value value* ...?

DESCRIPTION

This procedure creates an option menubutton whose name is *w*, plus an associated menu. Together they allow the user to select one of the values given by the *value* arguments. The current value will be stored in the global variable whose name is given by *varName* and it will also be displayed as the label in the option menubutton. The user can click on the menubutton to display a menu containing all of the *value*s and thereby select a new value. Once a new value is selected, it will be stored in the variable and appear in the option menubutton. The current value can also be changed by setting the variable.

The return value from **tk_optionMenu** is the name of the menu associated with *w*, so that the caller can change its configuration options or manipulate it in other ways.

KEYWORDS

option menu

NAME

options − Standard options supported by widgets

DESCRIPTION

This manual entry describes the common configuration options supported by widgets in the Tk toolkit. Every widget does not necessarily support every option (see the manual entries for individual widgets for a list of the standard options supported by that widget), but if a widget does support an option with one of the names listed below, then the option has exactly the effect described below.

In the descriptions below, ''Command-Line Name'' refers to the switch used in class commands and **configure** widget commands to set this value. For example, if an option's command-line switch is **−foreground** and there exists a widget **.a.b.c**, then the command

.a.b.c configure −foreground black

may be used to specify the value **black** for the option in the the widget **.a.b.c**. Command-line switches may be abbreviated, as long as the abbreviation is unambiguous. ''Database Name'' refers to the option's name in the option database (e.g. in .Xdefaults files). ''Database Class'' refers to the option's class value in the option database.

Command-Line Name: **−activebackground**
Database Name: **activeBackground**
Database Class: **Foreground**

Specifies background color to use when drawing active elements. An element (a widget or portion of a widget) is active if the mouse cursor is positioned over the element and pressing a mouse button will cause some action to occur. If strict Motif compliance has been requested by setting the **tk_strictMotif** variable, this option will normally be ignored; the normal background color will be used instead. For some elements on Windows and Macintosh systems, the active color will only be used while mouse button 1 is pressed over the element.

Command-Line Name: **−activeborderwidth**
Database Name: **activeBorderWidth**
Database Class: **BorderWidth**

Specifies a non-negative value indicating the width of the 3-D border drawn around active elements. See above for definition of active elements. The value may have any of the forms acceptable to **Tk_GetPixels**. This option is typically only available in widgets displaying more than one element at a time (e.g. menus but not buttons).

Command-Line Name: **−activeforeground**
Database Name: **activeForeground**
Database Class: **Background**

Specifies foreground color to use when drawing active elements. See above for definition of active elements.

Command-Line Name: **−anchor**
Database Name: **anchor**
Database Class: **Anchor**

Specifies how the information in a widget (e.g. text or a bitmap) is to be displayed in the widget. Must be one of the values **n**, **ne**, **e**, **se**, **s**, **sw**, **w**, **nw**, or **center**. For example, **nw** means display the information such that its top-left corner is at the top-left corner of the widget.

Command-Line Name: **−background or −bg**
Database Name: **background**
Database Class: **Background**

Specifies the normal background color to use when displaying the widget.

Command-Line Name: **−bitmap**
Database Name: **bitmap**
Database Class: **Bitmap**

Specifies a bitmap to display in the widget, in any of the forms acceptable to **Tk_GetBitmap**. The exact way in which the bitmap is displayed may be affected by other options such as **anchor** or **justify**. Typically, if this option is specified then it overrides other options that specify a textual value to display in the widget; the **bitmap** option may be reset to an empty string to re-enable a text display. In widgets that support both **bitmap** and **image** options, **image** will usually override **bitmap**.

Command-Line Name: **−borderwidth or −bd**
Database Name: **borderWidth**
Database Class: **BorderWidth**

Specifies a non-negative value indicating the width of the 3-D border to draw around the outside of the widget (if such a border is being drawn; the **relief** option typically determines this). The value may also be used when drawing 3-D effects in the interior of the widget. The value may have any of the forms acceptable to **Tk_GetPixels**.

Command-Line Name: **−cursor**
Database Name: **cursor**
Database Class: **Cursor**

Specifies the mouse cursor to be used for the widget. The value may have any of the forms acceptable to **Tk_GetCursor**.

Command-Line Name: **−disabledforeground**
Database Name: **disabledForeground**
Database Class: **DisabledForeground**

Specifies foreground color to use when drawing a disabled element. If the option is specified as an empty string (which is typically the case on monochrome displays), disabled elements are drawn with the normal foreground color but they are dimmed by drawing them with a stippled fill pattern.

Command-Line Name: **−exportselection**
Database Name: **exportSelection**
Database Class: **ExportSelection**

Specifies whether or not a selection in the widget should also be the X selection. The value may have any of the forms accepted by **Tcl_GetBoolean**, such as **true**, **false**, **0**, **1**, **yes**, or **no**. If the selection is exported, then selecting in the widget deselects the current X selection, selecting outside the widget deselects any widget selection, and the widget will respond to selection retrieval requests when it has a selection. The default is usually for widgets to export selections.

Command-Line Name: **−font**
Database Name: **font**
Database Class: **Font**

Specifies the font to use when drawing text inside the widget.

Command-Line Name: **−foreground or −fg**
Database Name: **foreground**
Database Class: **Foreground**

Specifies the normal foreground color to use when displaying the widget.

Command-Line Name: **−highlightbackground**
Database Name: **highlightBackground**
Database Class: **HighlightBackground**

Specifies the color to display in the traversal highlight region when the widget does not have the input focus.

Command-Line Name: **−highlightcolor**
Database Name: **highlightColor**
Database Class: **HighlightColor**

Specifies the color to use for the traversal highlight rectangle that is drawn around the widget when it has the input focus.

Command-Line Name: **−highlightthickness**
Database Name: **highlightThickness**
Database Class: **HighlightThickness**

Specifies a non-negative value indicating the width of the highlight rectangle to draw around the outside of the widget when it has the input focus. The value may have any of the forms acceptable to **Tk_GetPixels**. If the value is zero, no focus highlight is drawn around the widget.

Command-Line Name: **−image**
Database Name: **image**
Database Class: **Image**

Specifies an image to display in the widget, which must have been created with the **image create** command. Typically, if the **image** option is specified then it overrides other options that specify a bitmap or textual value to display in the widget; the **image** option may be reset to an empty string to re-enable a bitmap or text display.

Command-Line Name: **−insertbackground**
Database Name: **insertBackground**
Database Class: **Foreground**

Specifies the color to use as background in the area covered by the insertion cursor. This color will normally override either the normal background for the widget (or the selection background if the insertion cursor happens to fall in the selection).

Command-Line Name: **−insertborderwidth**
Database Name: **insertBorderWidth**
Database Class: **BorderWidth**

Specifies a non-negative value indicating the width of the 3-D border to draw around the insertion cursor. The value may have any of the forms acceptable to **Tk_GetPixels**.

Command-Line Name: **−insertofftime**
Database Name: **insertOffTime**
Database Class: **OffTime**

Specifies a non-negative integer value indicating the number of milliseconds the insertion cursor should remain ‘‘off’’ in each blink cycle. If this option is zero then the cursor doesn’t blink: it is on all the time.

Command-Line Name: **−insertontime**
Database Name: **insertOnTime**
Database Class: **OnTime**

Specifies a non-negative integer value indicating the number of milliseconds the insertion cursor should remain ‘‘on’’ in each blink cycle.

Command-Line Name: **−insertwidth**
Database Name: **insertWidth**
Database Class: **InsertWidth**

Specifies a value indicating the total width of the insertion cursor. The value may have any of the forms acceptable to **Tk_GetPixels**. If a border has been specified for the insertion cursor (using the **insertBorderWidth** option), the border will be drawn inside the width specified by the **insertWidth** option.

Command-Line Name: **−jump**
Database Name: **jump**
Database Class: **Jump**

For widgets with a slider that can be dragged to adjust a value, such as scrollbars, this option determines when notifications are made about changes in the value. The option's value must be a boolean of the form accepted by **Tcl_GetBoolean**. If the value is false, updates are made continuously as the slider is dragged. If the value is true, updates are delayed until the mouse button is released to end the drag; at that point a single notification is made (the value ''jumps'' rather than changing smoothly).

Command-Line Name: **−justify**
Database Name: **justify**
Database Class: **Justify**

When there are multiple lines of text displayed in a widget, this option determines how the lines line up with each other. Must be one of **left**, **center**, or **right**. **Left** means that the lines' left edges all line up, **center** means that the lines' centers are aligned, and **right** means that the lines' right edges line up.

Command-Line Name: **−orient**
Database Name: **orient**
Database Class: **Orient**

For widgets that can lay themselves out with either a horizontal or vertical orientation, such as scrollbars, this option specifies which orientation should be used. Must be either **horizontal** or **vertical** or an abbreviation of one of these.

Command-Line Name: **−padx**
Database Name: **padX**
Database Class: **Pad**

Specifies a non-negative value indicating how much extra space to request for the widget in the X-direction. The value may have any of the forms acceptable to **Tk_GetPixels**. When computing how large a window it needs, the widget will add this amount to the width it would normally need (as determined by the width of the things displayed in the widget); if the geometry manager can satisfy this request, the widget will end up with extra internal space to the left and/or right of what it displays inside. Most widgets only use this option for padding text: if they are displaying a bitmap or image, then they usually ignore padding options.

Command-Line Name: **−pady**
Database Name: **padY**
Database Class: **Pad**

Specifies a non-negative value indicating how much extra space to request for the widget in the Y-direction. The value may have any of the forms acceptable to **Tk_GetPixels**. When computing how large a window it needs, the widget will add this amount to the height it would normally need (as determined by the height of the things displayed in the widget); if the geometry manager can satisfy this request, the widget will end up with extra internal space above and/or below what it

displays inside. Most widgets only use this option for padding text: if they are displaying a bitmap or image, then they usually ignore padding options.

Command-Line Name: **−relief**
Database Name: **relief**
Database Class: **Relief**

Specifies the 3-D effect desired for the widget. Acceptable values are **raised**, **sunken**, **flat**, **ridge**, **solid**, and **groove**. The value indicates how the interior of the widget should appear relative to its exterior; for example, **raised** means the interior of the widget should appear to protrude from the screen, relative to the exterior of the widget.

Command-Line Name: **−repeatdelay**
Database Name: **repeatDelay**
Database Class: **RepeatDelay**

Specifies the number of milliseconds a button or key must be held down before it begins to auto-repeat. Used, for example, on the up- and down-arrows in scrollbars.

Command-Line Name: **−repeatinterval**
Database Name: **repeatInterval**
Database Class: **RepeatInterval**

Used in conjunction with **repeatDelay**: once auto-repeat begins, this option determines the number of milliseconds between auto-repeats.

Command-Line Name: **−selectbackground**
Database Name: **selectBackground**
Database Class: **Foreground**

Specifies the background color to use when displaying selected items.

Command-Line Name: **−selectborderwidth**
Database Name: **selectBorderWidth**
Database Class: **BorderWidth**

Specifies a non-negative value indicating the width of the 3-D border to draw around selected items. The value may have any of the forms acceptable to **Tk_GetPixels**.

Command-Line Name: **−selectforeground**
Database Name: **selectForeground**
Database Class: **Background**

Specifies the foreground color to use when displaying selected items.

Command-Line Name: **−setgrid**
Database Name: **setGrid**
Database Class: **SetGrid**

Specifies a boolean value that determines whether this widget controls the resizing grid for its top-level window. This option is typically used in text widgets, where the information in the widget has a natural size (the size of a character) and it makes sense for the window's dimensions to be integral numbers of these units. These natural window sizes form a grid. If the **setGrid** option is set to true then the widget will communicate with the window manager so that when the user interactively resizes the top-level window that contains the widget, the dimensions of the window will be displayed to the user in grid units and the window size will be constrained to integral numbers of grid units. See the section GRIDDED GEOMETRY MANAGEMENT in the **wm** manual entry for more details.

Command-Line Name: **−takefocus**
Database Name: **takeFocus**
Database Class: **TakeFocus**

Determines whether the window accepts the focus during keyboard traversal (e.g., Tab and Shift-Tab). Before setting the focus to a window, the traversal scripts consult the value of the **takeFocus** option. A value of **0** means that the window should be skipped entirely during keyboard traversal. **1** means that the window should receive the input focus as long as it is viewable (it and all of its ancestors are mapped). An empty value for the option means that the traversal scripts make the decision about whether or not to focus on the window: the current algorithm is to skip the window if it is disabled, if it has no key bindings, or if it is not viewable. If the value has any other form, then the traversal scripts take the value, append the name of the window to it (with a separator space), and evaluate the resulting string as a Tcl script. The script must return **0**, **1**, or an empty string: a **0** or **1** value specifies whether the window will receive the input focus, and an empty string results in the default decision described above. Note: this interpretation of the option is defined entirely by the Tcl scripts that implement traversal: the widget implementations ignore the option entirely, so you can change its meaning if you redefine the keyboard traversal scripts.

Command-Line Name: **−text**
Database Name: **text**
Database Class: **Text**

Specifies a string to be displayed inside the widget. The way in which the string is displayed depends on the particular widget and may be determined by other options, such as **anchor** or **justify**.

Command-Line Name: **−textvariable**
Database Name: **textVariable**
Database Class: **Variable**

Specifies the name of a variable. The value of the variable is a text string to be displayed inside the widget; if the variable value changes then the widget will automatically update itself to reflect the new value. The way in which the string is displayed in the widget depends on the particular widget and may be determined by other options, such as **anchor** or **justify**.

Command-Line Name: **−troughcolor**
Database Name: **troughColor**
Database Class: **Background**

Specifies the color to use for the rectangular trough areas in widgets such as scrollbars and scales.

Command-Line Name: **−underline**
Database Name: **underline**
Database Class: **Underline**

Specifies the integer index of a character to underline in the widget. This option is used by the default bindings to implement keyboard traversal for menu buttons and menu entries. 0 corresponds to the first character of the text displayed in the widget, 1 to the next character, and so on.

Command-Line Name: **−wraplength**
Database Name: **wrapLength**
Database Class: **WrapLength**

For widgets that can perform word-wrapping, this option specifies the maximum line length. Lines that would exceed this length are wrapped onto the next line, so that no line is longer than the specified length. The value may be specified in any of the standard forms for screen distances. If this value is less than or equal to 0 then no wrapping is done: lines will break only at newline characters in the text.

Command-Line Name: **−xscrollcommand**
Database Name: **xScrollCommand**
Database Class: **ScrollCommand**

Specifies the prefix for a command used to communicate with horizontal scrollbars. When the view in the widget's window changes (or whenever anything else occurs that could change the display in a scrollbar, such as a change in the total size of the widget's contents), the widget will generate a Tcl command by concatenating the scroll command and two numbers. Each of the numbers is a fraction between 0 and 1, which indicates a position in the document. 0 indicates the beginning of the document, 1 indicates the end, .333 indicates a position one third the way through the document, and so on. The first fraction indicates the first information in the document that is visible in the window, and the second fraction indicates the information just after the last portion that is visible. The command is then passed to the Tcl interpreter for execution. Typically the **xScrollCommand** option consists of the path name of a scrollbar widget followed by ''set'', e.g. ''.x.scrollbar set'': this will cause the scrollbar to be updated whenever the view in the window changes. If this option is not specified, then no command will be executed.

Command-Line Name: **−yscrollcommand**
Database Name: **yScrollCommand**
Database Class: **ScrollCommand**

Specifies the prefix for a command used to communicate with vertical scrollbars. This option is treated in the same way as the **xScrollCommand** option, except that it is used for vertical scrollbars and is provided by widgets that support vertical scrolling. See the description of **xScrollCommand** for details on how this option is used.

KEYWORDS

class, name, standard option, switch

NAME

pack − Obsolete syntax for packer geometry manager

SYNOPSIS

pack after *sibling window options* ?*window options ...*?

pack append *parent window options* ?*window options ...*?

pack before *sibling window options* ?*window options ...*?

pack unpack *window*

DESCRIPTION

Note: this manual entry describes the syntax for the **pack** *command as it existed before Tk version 3.3. Although this syntax continues to be supported for backward compatibility, it is obsolete and should not be used anymore. At some point in the future it may cease to be supported.*

The packer is a geometry manager that arranges the children of a parent by packing them in order around the edges of the parent. The first child is placed against one side of the window, occupying the entire span of the window along that side. This reduces the space remaining for other children as if the side had been moved in by the size of the first child. Then the next child is placed against one side of the remaining cavity, and so on until all children have been placed or there is no space left in the cavity.

The **before**, **after**, and **append** forms of the **pack** command are used to insert one or more children into the packing order for their parent. The **before** form inserts the children before window *sibling* in the order; all of the other windows must be siblings of *sibling*. The **after** form inserts the windows after *sibling*, and the **append** form appends one or more windows to the end of the packing order for *parent*. If a *window* named in any of these commands is already packed in its parent, it is removed from its current position in the packing order and repositioned as indicated by the command. All of these commands return an empty string as result.

The **unpack** form of the **pack** command removes *window* from the packing order of its parent and unmaps it. After the execution of this command the packer will no longer manage *window*'s geometry.

The placement of each child is actually a four-step process; the *options* argument following each *window* consists of a list of one or more fields that govern the placement of that window. In the discussion below, the term *cavity* refers to the space left in a parent when a particular child is placed (i.e. all the space that wasn't claimed by earlier children in the packing order). The term *parcel* refers to the space allocated to a particular child; this is not necessarily the same as the child window's final geometry.

The first step in placing a child is to determine which side of the cavity it will lie against. Any one of the following options may be used to specify a side:

top Position the child's parcel against the top of the cavity, occupying the full width of the cavity.

bottom Position the child's parcel against the bottom of the cavity, occupying the full width of the cavity.

left Position the child's parcel against the left side of the cavity, occupying the full height of the cavity.

right Position the child's parcel against the right side of the cavity, occupying the full height of the cavity.

At most one of these options should be specified for any given window. If no side is specified, then the default is **top**.

The second step is to decide on a parcel for the child. For **top** and **bottom** windows, the desired parcel width is normally the cavity width and the desired parcel height is the window's requested height, as passed

to **Tk_GeometryRequest**. For **left** and **right** windows, the desired parcel height is normally the cavity height and the desired width is the window's requested width. However, extra space may be requested for the window using any of the following options:

padx *num* Add *num* pixels to the window's requested width before computing the parcel size as described above.

pady *num* Add *num* pixels to the window's requested height before computing the parcel size as described above.

expand This option requests that the window's parcel absorb any extra space left over in the parent's cavity after packing all the children. The amount of space left over depends on the sizes requested by the other children, and may be zero. If several windows have all specified **expand** then the extra width will be divided equally among all the **left** and **right** windows that specified **expand** and the extra height will be divided equally among all the **top** and **bottom** windows that specified **expand**.

If the desired width or height for a parcel is larger than the corresponding dimension of the cavity, then the cavity's dimension is used instead.

The third step in placing the window is to decide on the window's width and height. The default is for the window to receive either its requested width and height or the those of the parcel, whichever is smaller. If the parcel is larger than the window's requested size, then the following options may be used to expand the window to partially or completely fill the parcel:

fill Set the window's size to equal the parcel size.

fillx Increase the window's width to equal the parcel's width, but retain the window's requested height.

filly Increase the window's height to equal the parcel's height, but retain the window's requested width.

The last step is to decide the window's location within its parcel. If the window's size equals the parcel's size, then the window simply fills the entire parcel. If the parcel is larger than the window, then one of the following options may be used to specify where the window should be positioned within its parcel:

frame center Center the window in its parcel. This is the default if no framing option is specified.

frame n Position the window with its top edge centered on the top edge of the parcel.

frame ne Position the window with its upper-right corner at the upper-right corner of the parcel.

frame e Position the window with its right edge centered on the right edge of the parcel.

frame se Position the window with its lower-right corner at the lower-right corner of the parcel.

frame s Position the window with its bottom edge centered on the bottom edge of the parcel.

frame sw Position the window with its lower-left corner at the lower-left corner of the parcel.

frame w Position the window with its left edge centered on the left edge of the parcel.

frame nw Position the window with its upper-left corner at the upper-left corner of the parcel.

The packer manages the mapped/unmapped state of all the packed children windows. It automatically maps the windows when it packs them, and it unmaps any windows for which there was no space left in the cavity.

The packer makes geometry requests on behalf of the parent windows it manages. For each parent window it requests a size large enough to accommodate all the options specified by all the packed children, such that zero space would be leftover for **expand** options.

KEYWORDS

geometry manager, location, packer, parcel, size

NAME

pack − Geometry manager that packs around edges of cavity

SYNOPSIS

pack *option arg* ?*arg ...*?

DESCRIPTION

The **pack** command is used to communicate with the packer, a geometry manager that arranges the children of a parent by packing them in order around the edges of the parent. The **pack** command can have any of several forms, depending on the *option* argument:

pack *slave* ?*slave ...*? ?*options*?

If the first argument to **pack** is a window name (any value starting with ‘‘.’’), then the command is processed in the same way as **pack configure**.

pack configure *slave* ?*slave ...*? ?*options*?

The arguments consist of the names of one or more slave windows followed by pairs of arguments that specify how to manage the slaves. See ‘‘THE PACKER ALGORITHM’’ below for details on how the options are used by the packer. The following options are supported:

−after *other*

Other must the name of another window. Use its master as the master for the slaves, and insert the slaves just after *other* in the packing order.

−anchor *anchor*

Anchor must be a valid anchor position such as **n** or **sw**; it specifies where to position each slave in its parcel. Defaults to **center**.

−before *other*

Other must the name of another window. Use its master as the master for the slaves, and insert the slaves just before *other* in the packing order.

−expand *boolean*

Specifies whether the slaves should be expanded to consume extra space in their master. *Boolean* may have any proper boolean value, such as **1** or **no**. Defaults to 0.

−fill *style*

If a slave’s parcel is larger than its requested dimensions, this option may be used to stretch the slave. *Style* must have one of the following values:

none Give the slave its requested dimensions plus any internal padding requested with **−ipadx** or **−ipady**. This is the default.

x Stretch the slave horizontally to fill the entire width of its parcel (except leave external padding as specified by **−padx**).

y Stretch the slave vertically to fill the entire height of its parcel (except leave external padding as specified by **−pady**).

both Stretch the slave both horizontally and vertically.

−in *other*

Insert the slave(s) at the end of the packing order for the master window given by *other*.

−ipadx *amount*

Amount specifies how much horizontal internal padding to leave on each side of the slave(s). *Amount* must be a valid screen distance, such as **2** or **.5c**. It defaults to 0.

−ipady *amount*

Amount specifies how much vertical internal padding to leave on each side of the slave(s). *Amount* defaults to 0.

−padx *amount*
Amount specifies how much horizontal external padding to leave on each side of the slave(s). *Amount* defaults to 0.

−pady *amount*
Amount specifies how much vertical external padding to leave on each side of the slave(s). *Amount* defaults to 0.

−side *side*
Specifies which side of the master the slave(s) will be packed against. Must be **left**, **right**, **top**, or **bottom**. Defaults to **top**.

If no **−in**, **−after** or **−before** option is specified then each of the slaves will be inserted at the end of the packing list for its parent unless it is already managed by the packer (in which case it will be left where it is). If one of these options is specified then all the slaves will be inserted at the specified point. If any of the slaves are already managed by the geometry manager then any unspecified options for them retain their previous values rather than receiving default values.

pack forget *slave* ?*slave* ...?
Removes each of the *slave*s from the packing order for its master and unmaps their windows. The slaves will no longer be managed by the packer.

pack info *slave*
Returns a list whose elements are the current configuration state of the slave given by *slave* in the same option-value form that might be specified to **pack configure**. The first two elements of the list are ''**−in** *master*'' where *master* is the slave's master.

pack propagate *master* ?*boolean*?
If *boolean* has a true boolean value such as **1** or **on** then propagation is enabled for *master*, which must be a window name (see ''GEOMETRY PROPAGATION'' below). If *boolean* has a false boolean value then propagation is disabled for *master*. In either of these cases an empty string is returned. If *boolean* is omitted then the command returns **0** or **1** to indicate whether propagation is currently enabled for *master*. Propagation is enabled by default.

pack slaves *master*
Returns a list of all of the slaves in the packing order for *master*. The order of the slaves in the list is the same as their order in the packing order. If *master* has no slaves then an empty string is returned.

THE PACKER ALGORITHM

For each master the packer maintains an ordered list of slaves called the *packing list*. The **−in**, **−after**, and **−before** configuration options are used to specify the master for each slave and the slave's position in the packing list. If none of these options is given for a slave then the slave is added to the end of the packing list for its parent.

The packer arranges the slaves for a master by scanning the packing list in order. At the time it processes each slave, a rectangular area within the master is still unallocated. This area is called the *cavity*; for the first slave it is the entire area of the master.

For each slave the packer carries out the following steps:

[1] The packer allocates a rectangular *parcel* for the slave along the side of the cavity given by the slave's **−side** option. If the side is top or bottom then the width of the parcel is the width of the cavity and its height is the requested height of the slave plus the **−ipady** and **−pady** options. For the left or right side the height of the parcel is the height of the cavity and the width is the

requested width of the slave plus the **−ipadx** and **−padx** options. The parcel may be enlarged further because of the **−expand** option (see ‘‘EXPANSION’’ below)

[2] The packer chooses the dimensions of the slave. The width will normally be the slave’s requested width plus twice its **−ipadx** option and the height will normally be the slave’s requested height plus twice its **−ipady** option. However, if the **−fill** option is **x** or **both** then the width of the slave is expanded to fill the width of the parcel, minus twice the **−padx** option. If the **−fill** option is **y** or **both** then the height of the slave is expanded to fill the width of the parcel, minus twice the **−pady** option.

[3] The packer positions the slave over its parcel. If the slave is smaller than the parcel then the **−anchor** option determines where in the parcel the slave will be placed. If **−padx** or **−pady** is non-zero, then the given amount of external padding will always be left between the slave and the edges of the parcel.

Once a given slave has been packed, the area of its parcel is subtracted from the cavity, leaving a smaller rectangular cavity for the next slave. If a slave doesn’t use all of its parcel, the unused space in the parcel will not be used by subsequent slaves. If the cavity should become too small to meet the needs of a slave then the slave will be given whatever space is left in the cavity. If the cavity shrinks to zero size, then all remaining slaves on the packing list will be unmapped from the screen until the master window becomes large enough to hold them again.

EXPANSION

If a master window is so large that there will be extra space left over after all of its slaves have been packed, then the extra space is distributed uniformly among all of the slaves for which the **−expand** option is set. Extra horizontal space is distributed among the expandable slaves whose **−side** is **left** or **right**, and extra vertical space is distributed among the expandable slaves whose **−side** is **top** or **bottom**.

GEOMETRY PROPAGATION

The packer normally computes how large a master must be to just exactly meet the needs of its slaves, and it sets the requested width and height of the master to these dimensions. This causes geometry information to propagate up through a window hierarchy to a top-level window so that the entire sub-tree sizes itself to fit the needs of the leaf windows. However, the **pack propagate** command may be used to turn off propagation for one or more masters. If propagation is disabled then the packer will not set the requested width and height of the packer. This may be useful if, for example, you wish for a master window to have a fixed size that you specify.

RESTRICTIONS ON MASTER WINDOWS

The master for each slave must either be the slave’s parent (the default) or a descendant of the slave’s parent. This restriction is necessary to guarantee that the slave can be placed over any part of its master that is visible without danger of the slave being clipped by its parent.

PACKING ORDER

If the master for a slave is not its parent then you must make sure that the slave is higher in the stacking order than the master. Otherwise the master will obscure the slave and it will appear as if the slave hasn’t been packed correctly. The easiest way to make sure the slave is higher than the master is to create the master window first: the most recently created window will be highest in the stacking order. Or, you can use the **raise** and **lower** commands to change the stacking order of either the master or the slave.

KEYWORDS

geometry manager, location, packer, parcel, propagation, size

NAME

tk_setPalette, tk_bisque − Modify the Tk color palette

SYNOPSIS

tk_setPalette *background*

tk_setPalette *name value* ?*name value* ...?

tk_bisque

DESCRIPTION

The **tk_setPalette** procedure changes the color scheme for Tk. It does this by modifying the colors of existing widgets and by changing the option database so that future widgets will use the new color scheme. If **tk_setPalette** is invoked with a single argument, the argument is the name of a color to use as the normal background color; **tk_setPalette** will compute a complete color palette from this background color. Alternatively, the arguments to **tk_setPalette** may consist of any number of *name−value* pairs, where the first argument of the pair is the name of an option in the Tk option database and the second argument is the new value to use for that option. The following database names are currently supported:

activeBackground	**foreground**	**selectColor**
activeForeground	**highlightBackground**	**selectBackground**
background	**highlightColor**	**selectForeground**
disabledForeground	**insertBackground**	**troughColor**

tk_setPalette tries to compute reasonable defaults for any options that you don't specify. You can specify options other than the above ones and Tk will change those options on widgets as well. This feature may be useful if you are using custom widgets with additional color options.

Once it has computed the new value to use for each of the color options, **tk_setPalette** scans the widget hierarchy to modify the options of all existing widgets. For each widget, it checks to see if any of the above options is defined for the widget. If so, and if the option's current value is the default, then the value is changed; if the option has a value other than the default, **tk_setPalette** will not change it. The default for an option is the one provided by the widget (**[lindex [$w configure $option] 3]**) unless **tk_setPalette** has been run previously, in which case it is the value specified in the previous invocation of **tk_setPalette**.

After modifying all the widgets in the application, **tk_setPalette** adds options to the option database to change the defaults for widgets created in the future. The new options are added at priority **widgetDefault**, so they will be overridden by options from the .Xdefaults file or options specified on the command-line that creates a widget.

The procedure **tk_bisque** is provided for backward compatibility: it restores the application's colors to the light brown (''bisque'') color scheme used in Tk 3.6 and earlier versions.

KEYWORDS

bisque, color, palette

NAME

photo − Full-color images

SYNOPSIS

image create photo ?*name*? ?*options*?

DESCRIPTION

A photo is an image whose pixels can display any color or be transparent. A photo image is stored internally in full color (24 bits per pixel), and is displayed using dithering if necessary. Image data for a photo image can be obtained from a file or a string, or it can be supplied from C code through a procedural interface. At present, only GIF and PPM/PGM formats are supported, but an interface exists to allow additional image file formats to be added easily. A photo image is transparent in regions where no image data has been supplied.

CREATING PHOTOS

Like all images, photos are created using the **image create** command. Photos support the following *options*:

−data *string*

Specifies the contents of the image as a string. The format of the string must be one of those for which there is an image file format handler that will accept string data. If both the **−data** and **−file** options are specified, the **−file** option takes precedence.

−format *format-name*

Specifies the name of the file format for the data specified with the **−data** or **−file** option.

−file *name*

name gives the name of a file that is to be read to supply data for the photo image. The file format must be one of those for which there is an image file format handler that can read data.

−gamma *value*

Specifies that the colors allocated for displaying this image in a window should be corrected for a non-linear display with the specified gamma exponent value. (The intensity produced by most CRT displays is a power function of the input value, to a good approximation; gamma is the exponent and is typically around 2). The value specified must be greater than zero. The default value is one (no correction). In general, values greater than one will make the image lighter, and values less than one will make it darker.

−height *number*

Specifies the height of the image, in pixels. This option is useful primarily in situations where the user wishes to build up the contents of the image piece by piece. A value of zero (the default) allows the image to expand or shrink vertically to fit the data stored in it.

−palette *palette-spec*

Specifies the resolution of the color cube to be allocated for displaying this image, and thus the number of colors used from the colormaps of the windows where it is displayed. The *palette-spec* string may be either a single decimal number, specifying the number of shades of gray to use, or three decimal numbers separated by slashes (/), specifying the number of shades of red, green and blue to use, respectively. If the first form (a single number) is used, the image will be displayed in monochrome (i.e., grayscale).

−width *number*

Specifies the width of the image, in pixels. This option is useful primarily in situations where the user wishes to build up the contents of the image piece by piece. A value of zero (the default) allows the image to expand or shrink horizontally to fit the data stored in it.

IMAGE COMMAND

When a photo image is created, Tk also creates a new command whose name is the same as the image. This command may be used to invoke various operations on the image. It has the following general form:

imageName option ?*arg arg ...*?

Option and the *arg*s determine the exact behavior of the command.

Those options that write data to the image generally expand the size of the image, if necessary, to accommodate the data written to the image, unless the user has specified non-zero values for the **−width** and/or **−height** configuration options, in which case the width and/or height, respectively, of the image will not be changed.

The following commands are possible for photo images:

imageName **blank**

Blank the image; that is, set the entire image to have no data, so it will be displayed as transparent, and the background of whatever window it is displayed in will show through.

imageName **cget** *option*

Returns the current value of the configuration option given by *option*. *Option* may have any of the values accepted by the **image create photo** command.

imageName **configure** ?*option*? ?*value option value ...*?

Query or modify the configuration options for the image. If no *option* is specified, returns a list describing all of the available options for *imageName* (see **Tk_ConfigureInfo** for information on the format of this list). If *option* is specified with no *value*, then the command returns a list describing the one named option (this list will be identical to the corresponding sublist of the value returned if no *option* is specified). If one or more *option−value* pairs are specified, then the command modifies the given option(s) to have the given value(s); in this case the command returns an empty string. *Option* may have any of the values accepted by the **image create photo** command.

imageName **copy** *sourceImage* ?*option value(s) ...*?

Copies a region from the image called *sourceImage* (which must be a photo image) to the image called *imageName*, possibly with pixel zooming and/or subsampling. If no options are specified, this command copies the whole of *sourceImage* into *imageName*, starting at coordinates (0,0) in *imageName*. The following options may be specified:

−from *x1 y1 x2 y2*

Specifies a rectangular sub-region of the source image to be copied. (*x1*,*y1*) and (*x2*,*y2*) specify diagonally opposite corners of the rectangle. If *x2* and *y2* are not specified, the default value is the bottom-right corner of the source image. The pixels copied will include the left and top edges of the specified rectangle but not the bottom or right edges. If the **−from** option is not given, the default is the whole source image.

−to *x1 y1 x2 y2*

Specifies a rectangular sub-region of the destination image to be affected. (*x1*,*y1*) and (*x2*,*y2*) specify diagonally opposite corners of the rectangle. If *x2* and *y2* are not specified, the default value is (*x1*,*y1*) plus the size of the source region (after subsampling and zooming, if specified). If *x2* and *y2* are specified, the source region will be replicated if necessary to fill the destination region in a tiled fashion.

−shrink

Specifies that the size of the destination image should be reduced, if necessary, so that the region being copied into is at the bottom-right corner of the image. This option will not affect the width or height of the image if the user has specified a non-zero value for the **−width** or **−height** configuration option, respectively.

−zoom *x y*

Specifies that the source region should be magnified by a factor of *x* in the X direction

and *y* in the Y direction. If *y* is not given, the default value is the same as *x*. With this option, each pixel in the source image will be expanded into a block of *x* x *y* pixels in the destination image, all the same color. *x* and *y* must be greater than 0.

−subsample *x y*
Specifies that the source image should be reduced in size by using only every *x*th pixel in the X direction and *y*th pixel in the Y direction. Negative values will cause the image to be flipped about the Y or X axes, respectively. If *y* is not given, the default value is the same as *x*.

imageName **get** *x y*
Returns the color of the pixel at coordinates (*x*,*y*) in the image as a list of three integers between 0 and 255, representing the red, green and blue components respectively.

imageName **put** *data* ?**−to** *x1 y1 x2 y2*?
Sets pixels in *imageName* to the colors specified in *data*. *data* is used to form a two-dimensional array of pixels that are then copied into the *imageName*. *data* is structured as a list of horizontal rows, from top to bottom, each of which is a list of colors, listed from left to right. Each color may be specified by name (e.g., blue) or in hexadecimal form (e.g., #2376af). The **−to** option can be used to specify the area of *imageName* to be affected. If only *x1* and *y1* are given, the area affected has its top-left corner at (*x1*,*y1*) and is the same size as the array given in *data*. If all four coordinates are given, they specify diagonally opposite corners of the affected rectangle, and the array given in *data* will be replicated as necessary in the X and Y directions to fill the rectangle.

imageName **read** *filename* ?*option value(s)* ...?
Reads image data from the file named *filename* into the image. This command first searches the list of image file format handlers for a handler that can interpret the data in *filename*, and then reads the image in *filename* into *imageName* (the destination image). The following options may be specified:

−format *format-name*
Specifies the format of the image data in *filename*. Specifically, only image file format handlers whose names begin with *format-name* will be used while searching for an image data format handler to read the data.

−from *x1 y1 x2 y2*
Specifies a rectangular sub-region of the image file data to be copied to the destination image. If only *x1* and *y1* are specified, the region extends from (*x1*,*y1*) to the bottom-right corner of the image in the image file. If all four coordinates are specified, they specify diagonally opposite corners or the region. The default, if this option is not specified, is the whole of the image in the image file.

−shrink
If this option, the size of *imageName* will be reduced, if necessary, so that the region into which the image file data are read is at the bottom-right corner of the *imageName*. This option will not affect the width or height of the image if the user has specified a non-zero value for the **−width** or **−height** configuration option, respectively.

−to *x y* Specifies the coordinates of the top-left corner of the region of *imageName* into which data from *filename* are to be read. The default is (0,0).

imageName **redither**
The dithering algorithm used in displaying photo images propagates quantization errors from one pixel to its neighbors. If the image data for *imageName* is supplied in pieces, the dithered image may not be exactly correct. Normally the difference is not noticeable, but if it is a problem, this command can be used to recalculate the dithered image in each window where the image is displayed.

imageName **write** *filename* ?*option value(s)* ...?

Writes image data from *imageName* to a file named *filename*. The following options may be specified:

−format *format-name*

Specifies the name of the image file format handler to be used to write the data to the file. Specifically, this subcommand searches for the first handler whose name matches a initial substring of *format-name* and which has the capability to write an image file. If this option is not given, this subcommand uses the first handler that has the capability to write an image file.

−from *x1 y1 x2 y2*

Specifies a rectangular region of *imageName* to be written to the image file. If only *x1* and *y1* are specified, the region extends from *(x1,y1)* to the bottom-right corner of *imageName*. If all four coordinates are given, they specify diagonally opposite corners of the rectangular region. The default, if this option is not given, is the whole image.

IMAGE FORMATS

The photo image code is structured to allow handlers for additional image file formats to be added easily. The photo image code maintains a list of these handlers. Handlers are added to the list by registering them with a call to **Tk_CreatePhotoImageFormat**. The standard Tk distribution comes with handlers for PPM/PGM and GIF formats, which are automatically registered on initialization.

When reading an image file or processing string data specified with the **−data** configuration option, the photo image code invokes each handler in turn until one is found that claims to be able to read the data in the file or string. Usually this will find the correct handler, but if it doesn't, the user may give a format name with the **−format** option to specify which handler to use. In fact the photo image code will try those handlers whose names begin with the string specified for the **−format** option (the comparison is case-insensitive). For example, if the user specifies **−format gif**, then a handler named GIF87 or GIF89 may be invoked, but a handler named JPEG may not (assuming that such handlers had been registered).

When writing image data to a file, the processing of the **−format** option is slightly different: the string value given for the **−format** option must begin with the complete name of the requested handler, and may contain additional information following that, which the handler can use, for example, to specify which variant to use of the formats supported by the handler.

COLOR ALLOCATION

When a photo image is displayed in a window, the photo image code allocates colors to use to display the image and dithers the image, if necessary, to display a reasonable approximation to the image using the colors that are available. The colors are allocated as a color cube, that is, the number of colors allocated is the product of the number of shades of red, green and blue.

Normally, the number of colors allocated is chosen based on the depth of the window. For example, in an 8-bit PseudoColor window, the photo image code will attempt to allocate seven shades of red, seven shades of green and four shades of blue, for a total of 198 colors. In a 1-bit StaticGray (monochrome) window, it will allocate two colors, black and white. In a 24-bit DirectColor or TrueColor window, it will allocate 256 shades each of red, green and blue. Fortunately, because of the way that pixel values can be combined in DirectColor and TrueColor windows, this only requires 256 colors to be allocated. If not all of the colors can be allocated, the photo image code reduces the number of shades of each primary color and tries again.

The user can exercise some control over the number of colors that a photo image uses with the **−palette** configuration option. If this option is used, it specifies the maximum number of shades of each primary color to try to allocate. It can also be used to force the image to be displayed in shades of gray, even on a color display, by giving a single number rather than three numbers separated by slashes.

CREDITS

The photo image type was designed and implemented by Paul Mackerras, based on his earlier photo widget and some suggestions from John Ousterhout.

KEYWORDS

photo, image, color

NAME

place − Geometry manager for fixed or rubber-sheet placement

SYNOPSIS

place *window option value* ?*option value* ...?

place configure *window option value* ?*option value* ...?

place forget *window*

place info *window*

place slaves *window*

DESCRIPTION

The placer is a geometry manager for Tk. It provides simple fixed placement of windows, where you specify the exact size and location of one window, called the *slave*, within another window, called the *master*. The placer also provides rubber-sheet placement, where you specify the size and location of the slave in terms of the dimensions of the master, so that the slave changes size and location in response to changes in the size of the master. Lastly, the placer allows you to mix these styles of placement so that, for example, the slave has a fixed width and height but is centered inside the master.

If the first argument to the **place** command is a window path name or **configure** then the command arranges for the placer to manage the geometry of a slave whose path name is *window*. The remaining arguments consist of one or more *option−value* pairs that specify the way in which *window*'s geometry is managed. If the placer is already managing *window*, then the *option−value* pairs modify the configuration for *window*. In this form the **place** command returns an empty string as result. The following *option−value* pairs are supported:

−in *master*

Master specifes the path name of the window relative to which *window* is to be placed. *Master* must either be *window*'s parent or a descendant of *window*'s parent. In addition, *master* and *window* must both be descendants of the same top-level window. These restrictions are necessary to guarantee that *window* is visible whenever *master* is visible. If this option isn't specified then the master defaults to *window*'s parent.

−x *location*

Location specifies the x-coordinate within the master window of the anchor point for *window*. The location is specified in screen units (i.e. any of the forms accepted by **Tk_GetPixels**) and need not lie within the bounds of the master window.

−relx *location*

Location specifies the x-coordinate within the master window of the anchor point for *window*. In this case the location is specified in a relative fashion as a floating-point number: 0.0 corresponds to the left edge of the master and 1.0 corresponds to the right edge of the master. *Location* need not be in the range 0.0−1.0. If both **−x** and **−relx** are specified for a slave then their values are summed. For example, **−relx 0.5 −x −2** positions the left edge of the slave 2 pixels to the left of the center of its master.

−y *location*

Location specifies the y-coordinate within the master window of the anchor point for *window*. The location is specified in screen units (i.e. any of the forms accepted by **Tk_GetPixels**) and need not lie within the bounds of the master window.

−rely *location*

Location specifies the y-coordinate within the master window of the anchor point for *window*. In this case the value is specified in a relative fashion as a floating-point number: 0.0 corresponds to the top edge of the master and 1.0 corresponds to the bottom edge of the master. *Location* need not be in the range 0.0−1.0. If both **−y** and **−rely** are specified for a slave then their values are summed. For example, **−rely 0.5 −x 3** positions the top edge of the slave 3 pixels below the center of its master.

−anchor *where*

Where specifies which point of *window* is to be positioned at the (x,y) location selected by the **−x**, **−y**, **−relx**, and **−rely** options. The anchor point is in terms of the outer area of *window* including its border, if any. Thus if *where* is **se** then the lower-right corner of *window*'s border will appear at the given (x,y) location in the master. The anchor position defaults to **nw**.

−width *size*

Size specifies the width for *window* in screen units (i.e. any of the forms accepted by **Tk_GetPixels**). The width will be the outer width of *window* including its border, if any. If *size* is an empty string, or if no **−width** or **−relwidth** option is specified, then the width requested internally by the window will be used.

−relwidth *size*

Size specifies the width for *window*. In this case the width is specified as a floating-point number relative to the width of the master: 0.5 means *window* will be half as wide as the master, 1.0 means *window* will have the same width as the master, and so on. If both **−width** and **−relwidth** are specified for a slave, their values are summed. For example, **−relwidth 1.0 −width 5** makes the slave 5 pixels wider than the master.

−height *size*

Size specifies the height for *window* in screen units (i.e. any of the forms accepted by **Tk_GetPixels**). The height will be the outer dimension of *window* including its border, if any. If *size* is an empty string, or if no **−height** or **−relheight** option is specified, then the height requested internally by the window will be used.

−relheight *size*

Size specifies the height for *window*. In this case the height is specified as a floating-point number relative to the height of the master: 0.5 means *window* will be half as high as the master, 1.0 means *window* will have the same height as the master, and so on. If both **−height** and **−relheight** are specified for a slave, their values are summed. For example, **−relheight 1.0 −height −2** makes the slave 2 pixels shorter than the master.

−bordermode *mode*

Mode determines the degree to which borders within the master are used in determining the placement of the slave. The default and most common value is **inside**. In this case the placer considers the area of the master to be the innermost area of the master, inside any border: an option of **−x 0** corresponds to an x-coordinate just inside the border and an option of **−relwidth 1.0** means *window* will fill the area inside the master's border. If *mode* is **outside** then the placer considers the area of the master to include its border; this mode is typically used when placing *window* outside its master, as with the options **−x 0 −y 0 −anchor ne**. Lastly, *mode* may be specified as **ignore**, in which case borders are ignored: the area of the master is considered to be its official X area, which includes any internal border but no external border. A bordermode of **ignore** is probably not very useful.

If the same value is specified separately with two different options, such as **−x** and **−relx**, then the most recent option is used and the older one is ignored.

The **place slaves** command returns a list of all the slave windows for which *window* is the master. If there are no slaves for *window* then an empty string is returned.

The **place forget** command causes the placer to stop managing the geometry of *window*. As a side effect of this command *window* will be unmapped so that it doesn't appear on the screen. If *window* isn't currently managed by the placer then the command has no effect. **Place forget** returns an empty string as result.

The **place info** command returns a list giving the current configuration of *window*. The list consists of *option−value* pairs in exactly the same form as might be specified to the **place configure** command. If the configuration of a window has been retrieved with **place info**, that configuration can be restored later by first using **place forget** to erase any existing information for the window and then invoking **place configure** with the saved information.

FINE POINTS

It is not necessary for the master window to be the parent of the slave window. This feature is useful in at least two situations. First, for complex window layouts it means you can create a hierarchy of subwindows whose only purpose is to assist in the layout of the parent. The ''real children'' of the parent (i.e. the windows that are significant for the application's user interface) can be children of the parent yet be placed inside the windows of the geometry-management hierarchy. This means that the path names of the ''real children'' don't reflect the geometry-management hierarchy and users can specify options for the real children without being aware of the structure of the geometry-management hierarchy.

A second reason for having a master different than the slave's parent is to tie two siblings together. For example, the placer can be used to force a window always to be positioned centered just below one of its siblings by specifying the configuration

−in *sibling* **−relx 0.5 −rely 1.0 −anchor n −bordermode outside**

Whenever the sibling is repositioned in the future, the slave will be repositioned as well.

Unlike many other geometry managers (such as the packer) the placer does not make any attempt to manipulate the geometry of the master windows or the parents of slave windows (i.e. it doesn't set their requested sizes). To control the sizes of these windows, make them windows like frames and canvases that provide configuration options for this purpose.

KEYWORDS

geometry manager, height, location, master, place, rubber sheet, slave, width

NAME

tk_popup − Post a popup menu

SYNOPSIS

tk_popup *menu x y* ?*entry*?

DESCRIPTION

This procedure posts a menu at a given position on the screen and configures Tk so that the menu and its cascaded children can be traversed with the mouse or the keyboard. *Menu* is the name of a menu widget and *x* and *y* are the root coordinates at which to display the menu. If *entry* is omitted or an empty string, the menu's upper left corner is positioned at the given point. Otherwise *entry* gives the index of an entry in *menu* and the menu will be positioned so that the entry is positioned over the given point.

KEYWORDS

menu, popup

NAME

radiobutton − Create and manipulate radiobutton widgets

SYNOPSIS

radiobutton *pathName* ?*options*?

STANDARD OPTIONS

−activebackground	**−cursor**	**−highlightthickness**	**−takefocus**
−activeforeground	**−disabledforeground**	**−image**	**−text**
−anchor	**−font**	**−justify**	**−textvariable**
−background	**−foreground**	**−padx**	**−underline**
−bitmap	**−highlightbackground**	**−pady**	**−wraplength**
−borderwidth	**−highlightcolor**	**−relief**	

See the **options** manual entry for details on the standard options.

WIDGET-SPECIFIC OPTIONS

Command-Line Name: **−command**
Database Name: **command**
Database Class: **Command**

Specifies a Tcl command to associate with the button. This command is typically invoked when mouse button 1 is released over the button window. The button's global variable (**−variable** option) will be updated before the command is invoked.

Command-Line Name: **−height**
Database Name: **height**
Database Class: **Height**

Specifies a desired height for the button. If an image or bitmap is being displayed in the button then the value is in screen units (i.e. any of the forms acceptable to **Tk_GetPixels**); for text it is in lines of text. If this option isn't specified, the button's desired height is computed from the size of the image or bitmap or text being displayed in it.

Command-Line Name: **−indicatoron**
Database Name: **indicatorOn**
Database Class: **IndicatorOn**

Specifies whether or not the indicator should be drawn. Must be a proper boolean value. If false, the **relief** option is ignored and the widget's relief is always sunken if the widget is selected and raised otherwise.

Command-Line Name: **−selectcolor**
Database Name: **selectColor**
Database Class: **Background**

Specifies a background color to use when the button is selected. If **indicatorOn** is true then the color applies to the indicator. Under Windows, this color is used as the background for the indicator regardless of the select state. If **indicatorOn** is false, this color is used as the background for the entire widget, in place of **background** or **activeBackground**, whenever the widget is selected. If specified as an empty string then no special color is used for displaying when the widget is selected.

Command-Line Name: **−selectimage**
Database Name: **selectImage**
Database Class: **SelectImage**

Specifies an image to display (in place of the **image** option) when the radiobutton is selected. This option is ignored unless the **image** option has been specified.

Command-Line Name: **−state**
Database Name: **state**
Database Class: **State**

Specifies one of three states for the radiobutton: **normal**, **active**, or **disabled**. In normal state the radiobutton is displayed using the **foreground** and **background** options. The active state is typically used when the pointer is over the radiobutton. In active state the radiobutton is displayed using the **activeForeground** and **activeBackground** options. Disabled state means that the radiobutton should be insensitive: the default bindings will refuse to activate the widget and will ignore mouse button presses. In this state the **disabledForeground** and **background** options determine how the radiobutton is displayed.

Command-Line Name: **−value**
Database Name: **value**
Database Class: **Value**

Specifies value to store in the button's associated variable whenever this button is selected.

Command-Line Name: **−variable**
Database Name: **variable**
Database Class: **Variable**

Specifies name of global variable to set whenever this button is selected. Changes in this variable also cause the button to select or deselect itself. Defaults to the value **selectedButton**.

Command-Line Name: **−width**
Database Name: **width**
Database Class: **Width**

Specifies a desired width for the button. If an image or bitmap is being displayed in the button, the value is in screen units (i.e. any of the forms acceptable to **Tk_GetPixels**); for text it is in characters. If this option isn't specified, the button's desired width is computed from the size of the image or bitmap or text being displayed in it.

DESCRIPTION

The **radiobutton** command creates a new window (given by the *pathName* argument) and makes it into a radiobutton widget. Additional options, described above, may be specified on the command line or in the option database to configure aspects of the radiobutton such as its colors, font, text, and initial relief. The **radiobutton** command returns its *pathName* argument. At the time this command is invoked, there must not exist a window named *pathName*, but *pathName*'s parent must exist.

A radiobutton is a widget that displays a textual string, bitmap or image and a diamond or circle called an *indicator*. If text is displayed, it must all be in a single font, but it can occupy multiple lines on the screen (if it contains newlines or if wrapping occurs because of the **wrapLength** option) and one of the characters may optionally be underlined using the **underline** option. A radiobutton has all of the behavior of a simple button: it can display itself in either of three different ways, according to the **state** option; it can be made to appear raised, sunken, or flat; it can be made to flash; and it invokes a Tcl command whenever mouse button 1 is clicked over the check button.

In addition, radiobuttons can be *selected*. If a radiobutton is selected, the indicator is normally drawn with a selected appearance, and a Tcl variable associated with the radiobutton is set to a particular value (normally 1). Under Unix, the indicator is drawn with a sunken relief and a special color. Under Windows, the indicator is drawn with a round mark inside. If the radiobutton is not selected, then the indicator is drawn with a deselected appearance, and the associated variable is set to a different value (typically 0). Under Unix, the indicator is drawn with a raised relief and no special color. Under Windows, the indicator is drawn without a round mark inside. Typically, several radiobuttons share a single variable and the value of

the variable indicates which radiobutton is to be selected. When a radiobutton is selected it sets the value of the variable to indicate that fact; each radiobutton also monitors the value of the variable and automatically selects and deselects itself when the variable's value changes. By default the variable **selectedButton** is used; its contents give the name of the button that is selected, or the empty string if no button associated with that variable is selected. The name of the variable for a radiobutton, plus the variable to be stored into it, may be modified with options on the command line or in the option database. Configuration options may also be used to modify the way the indicator is displayed (or whether it is displayed at all). By default a radiobutton is configured to select itself on button clicks.

WIDGET COMMAND

The **radiobutton** command creates a new Tcl command whose name is *pathName*. This command may be used to invoke various operations on the widget. It has the following general form:

pathName option ?*arg arg* ...?

Option and the *arg*s determine the exact behavior of the command. The following commands are possible for radiobutton widgets:

pathName **cget** *option*

Returns the current value of the configuration option given by *option*. *Option* may have any of the values accepted by the **radiobutton** command.

pathName **configure** ?*option*? ?*value option value* ...?

Query or modify the configuration options of the widget. If no *option* is specified, returns a list describing all of the available options for *pathName* (see **Tk_ConfigureInfo** for information on the format of this list). If *option* is specified with no *value*, the command returns a list describing the one named option (this list will be identical to the corresponding sublist of the value returned if no *option* is specified). If one or more *option−value* pairs are specified, the command modifies the given widget option(s) to have the given value(s); in this case the command returns an empty string. *Option* may have any of the values accepted by the **radiobutton** command.

pathName **deselect**

Deselects the radiobutton and sets the associated variable to an empty string. If this radiobutton was not currently selected, the command has no effect.

pathName **flash**

Flashes the radiobutton. This is accomplished by redisplaying the radiobutton several times, alternating between active and normal colors. At the end of the flash the radiobutton is left in the same normal/active state as when the command was invoked. This command is ignored if the radiobutton's state is **disabled**.

pathName **invoke**

Does just what would have happened if the user invoked the radiobutton with the mouse: selects the button and invokes its associated Tcl command, if there is one. The return value is the return value from the Tcl command, or an empty string if there is no command associated with the radiobutton. This command is ignored if the radiobutton's state is **disabled**.

pathName **select**

Selects the radiobutton and sets the associated variable to the value corresponding to this widget.

BINDINGS

Tk automatically creates class bindings for radiobuttons that give them the following default behavior:

[1] On Unix systems, a radiobutton activates whenever the mouse passes over it and deactivates whenever the mouse leaves the radiobutton. On Mac and Windows systems, when mouse button 1 is pressed over a radiobutton, the button activates whenever the mouse pointer is inside the button, and deactivates whenever the mouse pointer leaves the button.

[2] When mouse button 1 is pressed over a radiobutton it is invoked (it becomes selected and the command associated with the button is invoked, if there is one).

[3] When a radiobutton has the input focus, the space key causes the radiobutton to be invoked.

If the radiobutton's state is **disabled** then none of the above actions occur: the radiobutton is completely non-responsive.

The behavior of radiobuttons can be changed by defining new bindings for individual widgets or by redefining the class bindings.

KEYWORDS

radiobutton, widget

NAME

raise − Change a window’s position in the stacking order

SYNOPSIS

raise *window* ?*aboveThis*?

DESCRIPTION

If the *aboveThis* argument is omitted then the command raises *window* so that it is above all of its siblings in the stacking order (it will not be obscured by any siblings and will obscure any siblings that overlap it). If *aboveThis* is specified then it must be the path name of a window that is either a sibling of *window* or the descendant of a sibling of *window*. In this case the **raise** command will insert *window* into the stacking order just above *aboveThis* (or the ancestor of *aboveThis* that is a sibling of *window*); this could end up either raising or lowering *window*.

SEE ALSO

lower

KEYWORDS

obscure, raise, stacking order

NAME

scale − Create and manipulate scale widgets

SYNOPSIS

scale *pathName* ?*options*?

STANDARD OPTIONS

−activebackground	**−font**	**−highlightthickness**	**−repeatinterval**
−background	**−foreground**	**−orient**	**−takefocus**
−borderwidth	**−highlightbackground**	**−relief**	**−troughcolor**
−cursor	**−highlightcolor**	**−repeatdelay**	

See the **options** manual entry for details on the standard options.

WIDGET-SPECIFIC OPTIONS

Command-Line Name: **−bigincrement**
Database Name: **bigIncrement**
Database Class: **BigIncrement**

Some interactions with the scale cause its value to change by ‘‘large’’ increments; this option specifies the size of the large increments. If specified as 0, the large increments default to 1/10 the range of the scale.

Command-Line Name: **−command**
Database Name: **command**
Database Class: **Command**

Specifies the prefix of a Tcl command to invoke whenever the scale’s value is changed via a widget command. The actual command consists of this option followed by a space and a real number indicating the new value of the scale.

Command-Line Name: **−digits**
Database Name: **digits**
Database Class: **Digits**

An integer specifying how many significant digits should be retained when converting the value of the scale to a string. If the number is less than or equal to zero, then the scale picks the smallest value that guarantees that every possible slider position prints as a different string.

Command-Line Name: **−from**
Database Name: **from**
Database Class: **From**

A real value corresponding to the left or top end of the scale.

Command-Line Name: **−label**
Database Name: **label**
Database Class: **Label**

A string to display as a label for the scale. For vertical scales the label is displayed just to the right of the top end of the scale. For horizontal scales the label is displayed just above the left end of the scale. If the option is specified as an empty string, no label is displayed.

Command-Line Name: **−length**
Database Name: **length**
Database Class: **Length**

Specifies the desired long dimension of the scale in screen units (i.e. any of the forms acceptable to **Tk_GetPixels**). For vertical scales this is the scale’s height; for horizontal scales it is the scale’s width.

Command-Line Name: **−resolution**
Database Name: **resolution**
Database Class: **Resolution**

A real value specifying the resolution for the scale. If this value is greater than zero then the scale's value will always be rounded to an even multiple of this value, as will tick marks and the endpoints of the scale. If the value is less than zero then no rounding occurs. Defaults to 1 (i.e., the value will be integral).

Command-Line Name: **−showvalue**
Database Name: **showValue**
Database Class: **ShowValue**

Specifies a boolean value indicating whether or not the current value of the scale is to be displayed.

Command-Line Name: **−sliderlength**
Database Name: **sliderLength**
Database Class: **SliderLength**

Specfies the size of the slider, measured in screen units along the slider's long dimension. The value may be specified in any of the forms acceptable to **Tk_GetPixels**.

Command-Line Name: **−sliderrelief**
Database Name: **sliderRelief**
Database Class: **SliderRelief**

Specifies the relief to use when drawing the slider, such as **raised** or **sunken**.

Command-Line Name: **−state**
Database Name: **state**
Database Class: **State**

Specifies one of three states for the scale: **normal**, **active**, or **disabled**. If the scale is disabled then the value may not be changed and the scale won't activate. If the scale is active, the slider is displayed using the color specified by the **activeBackground** option.

Command-Line Name: **−tickinterval**
Database Name: **tickInterval**
Database Class: **TickInterval**

Must be a real value. Determines the spacing between numerical tick marks displayed below or to the left of the slider. If 0, no tick marks will be displayed.

Command-Line Name: **−to**
Database Name: **to**
Database Class: **To**

Specifies a real value corresponding to the right or bottom end of the scale. This value may be either less than or greater than the **from** option.

Command-Line Name: **−variable**
Database Name: **variable**
Database Class: **Variable**

Specifies the name of a global variable to link to the scale. Whenever the value of the variable changes, the scale will update to reflect this value. Whenever the scale is manipulated interactively, the variable will be modified to reflect the scale's new value.

Command-Line Name: **−width**
Database Name: **width**
Database Class: **Width**

Specifies the desired narrow dimension of the trough in screen units (i.e. any of the forms acceptable to **Tk_GetPixels**). For vertical scales this is the trough's width; for horizontal scales this is the trough's height.

DESCRIPTION

The **scale** command creates a new window (given by the *pathName* argument) and makes it into a scale widget. Additional options, described above, may be specified on the command line or in the option database to configure aspects of the scale such as its colors, orientation, and relief. The **scale** command returns its *pathName* argument. At the time this command is invoked, there must not exist a window named *pathName*, but *pathName*'s parent must exist.

A scale is a widget that displays a rectangular *trough* and a small *slider*. The trough corresponds to a range of real values (determined by the **from**, **to**, and **resolution** options), and the position of the slider selects a particular real value. The slider's position (and hence the scale's value) may be adjusted with the mouse or keyboard as described in the BINDINGS section below. Whenever the scale's value is changed, a Tcl command is invoked (using the **command** option) to notify other interested widgets of the change. In addition, the value of the scale can be linked to a Tcl variable (using the **variable** option), so that changes in either are reflected in the other.

Three annotations may be displayed in a scale widget: a label appearing at the top right of the widget (top left for horizontal scales), a number displayed just to the left of the slider (just above the slider for horizontal scales), and a collection of numerical tick marks just to the left of the current value (just below the trough for horizontal scales). Each of these three annotations may be enabled or disabled using the configuration options.

WIDGET COMMAND

The **scale** command creates a new Tcl command whose name is *pathName*. This command may be used to invoke various operations on the widget. It has the following general form:

pathName option ?*arg arg ...*?

Option and the *arg*s determine the exact behavior of the command. The following commands are possible for scale widgets:

pathName **cget** *option*

Returns the current value of the configuration option given by *option*. *Option* may have any of the values accepted by the **scale** command.

pathName **configure** ?*option*? ?*value option value ...*?

Query or modify the configuration options of the widget. If no *option* is specified, returns a list describing all of the available options for *pathName* (see **Tk_ConfigureInfo** for information on the format of this list). If *option* is specified with no *value*, then the command returns a list describing the one named option (this list will be identical to the corresponding sublist of the value returned if no *option* is specified). If one or more *option−value* pairs are specified, then the command modifies the given widget option(s) to have the given value(s); in this case the command returns an empty string. *Option* may have any of the values accepted by the **scale** command.

pathName **coords** ?*value*?

Returns a list whose elements are the x and y coordinates of the point along the centerline of the trough that corresponds to *value*. If *value* is omitted then the scale's current value is used.

pathName **get** ?*x y*?

If *x* and *y* are omitted, returns the current value of the scale. If *x* and *y* are specified, they give pixel coordinates within the widget; the command returns the scale value corresponding to the given pixel. Only one of *x* or *y* is used: for horizontal scales *y* is ignored, and for vertical scales *x* is ignored.

pathName **identify** *x y*

Returns a string indicating what part of the scale lies under the coordinates given by *x* and *y*. A return value of **slider** means that the point is over the slider; **trough1** means that the point is over the portion of the slider above or to the left of the slider; and **trough2** means that the point is over the portion of the slider below or to the right of the slider. If the point isn't over one of these elements, an empty string is returned.

pathName **set** *value*

This command is invoked to change the current value of the scale, and hence the position at which the slider is displayed. *Value* gives the new value for the scale. The command has no effect if the scale is disabled.

BINDINGS

Tk automatically creates class bindings for scales that give them the following default behavior. Where the behavior is different for vertical and horizontal scales, the horizontal behavior is described in parentheses.

[1] If button 1 is pressed in the trough, the scale's value will be incremented or decremented by the value of the **resolution** option so that the slider moves in the direction of the cursor. If the button is held down, the action auto-repeats.

[2] If button 1 is pressed over the slider, the slider can be dragged with the mouse.

[3] If button 1 is pressed in the trough with the Control key down, the slider moves all the way to the end of its range, in the direction towards the mouse cursor.

[4] If button 2 is pressed, the scale's value is set to the mouse position. If the mouse is dragged with button 2 down, the scale's value changes with the drag.

[5] The Up and Left keys move the slider up (left) by the value of the **resolution** option.

[6] The Down and Right keys move the slider down (right) by the value of the **resolution** option.

[7] Control-Up and Control-Left move the slider up (left) by the value of the **bigIncrement** option.

[8] Control-Down and Control-Right move the slider down (right) by the value of the **bigIncrement** option.

[9] Home moves the slider to the top (left) end of its range.

[10] End moves the slider to the bottom (right) end of its range.

If the scale is disabled using the **state** option then none of the above bindings have any effect.

The behavior of scales can be changed by defining new bindings for individual widgets or by redefining the class bindings.

KEYWORDS

scale, slider, trough, widget

NAME

scrollbar − Create and manipulate scrollbar widgets

SYNOPSIS

scrollbar *pathName* ?*options*?

STANDARD OPTIONS

−activebackground	**−highlightbackground**	**−orient**	**−takefocus**
−background	**−highlightcolor**	**−relief**	**−troughcolor**
−borderwidth	**−highlightthickness**	**−repeatdelay**	
−cursor	**−jump**	**−repeatinterval**	

See the **options** manual entry for details on the standard options.

WIDGET-SPECIFIC OPTIONS

Command-Line Name: **−activerelief**
Database Name: **activeRelief**
Database Class: **ActiveRelief**

Specifies the relief to use when displaying the element that is active, if any. Elements other than the active element are always displayed with a raised relief.

Command-Line Name: **−command**
Database Name: **command**
Database Class: **Command**

Specifies the prefix of a Tcl command to invoke to change the view in the widget associated with the scrollbar. When a user requests a view change by manipulating the scrollbar, a Tcl command is invoked. The actual command consists of this option followed by additional information as described later. This option almost always has a value such as **.t xview** or **.t yview**, consisting of the name of a widget and either **xview** (if the scrollbar is for horizontal scrolling) or **yview** (for vertical scrolling). All scrollable widgets have **xview** and **yview** commands that take exactly the additional arguments appended by the scrollbar as described in SCROLLING COMMANDS below.

Command-Line Name: **−elementborderwidth**
Database Name: **elementBorderWidth**
Database Class: **BorderWidth**

Specifies the width of borders drawn around the internal elements of the scrollbar (the two arrows and the slider). The value may have any of the forms acceptable to **Tk_GetPixels**. If this value is less than zero, the value of the **borderWidth** option is used in its place.

Command-Line Name: **−width**
Database Name: **width**
Database Class: **Width**

Specifies the desired narrow dimension of the scrollbar window, not including 3-D border, if any. For vertical scrollbars this will be the width and for horizontal scrollbars this will be the height. The value may have any of the forms acceptable to **Tk_GetPixels**.

DESCRIPTION

The **scrollbar** command creates a new window (given by the *pathName* argument) and makes it into a scrollbar widget. Additional options, described above, may be specified on the command line or in the option database to configure aspects of the scrollbar such as its colors, orientation, and relief. The **scrollbar** command returns its *pathName* argument. At the time this command is invoked, there must not exist a window named *pathName*, but *pathName*'s parent must exist.

A scrollbar is a widget that displays two arrows, one at each end of the scrollbar, and a *slider* in the middle portion of the scrollbar. It provides information about what is visible in an *associated window* that displays an document of some sort (such as a file being edited or a drawing). The position and size of the slider indicate which portion of the document is visible in the associated window. For example, if the slider in a vertical scrollbar covers the top third of the area between the two arrows, it means that the associated window displays the top third of its document.

Scrollbars can be used to adjust the view in the associated window by clicking or dragging with the mouse. See the BINDINGS section below for details.

ELEMENTS

A scrollbar displays five elements, which are referred to in the widget commands for the scrollbar:

arrow1 The top or left arrow in the scrollbar.

trough1 The region between the slider and **arrow1**.

slider The rectangle that indicates what is visible in the associated widget.

trough2 The region between the slider and **arrow2**.

arrow2 The bottom or right arrow in the scrollbar.

WIDGET COMMAND

The **scrollbar** command creates a new Tcl command whose name is *pathName*. This command may be used to invoke various operations on the widget. It has the following general form:

pathName option ?*arg arg ...*?

Option and the *arg*s determine the exact behavior of the command. The following commands are possible for scrollbar widgets:

pathName **activate** ?*element*?

Marks the element indicated by *element* as active, which causes it to be displayed as specified by the **activeBackground** and **activeRelief** options. The only element values understood by this command are **arrow1**, **slider**, or **arrow2**. If any other value is specified then no element of the scrollbar will be active. If *element* is not specified, the command returns the name of the element that is currently active, or an empty string if no element is active.

pathName **cget** *option*

Returns the current value of the configuration option given by *option*. *Option* may have any of the values accepted by the **scrollbar** command.

pathName **configure** ?*option*? ?*value option value ...*?

Query or modify the configuration options of the widget. If no *option* is specified, returns a list describing all of the available options for *pathName* (see **Tk_ConfigureInfo** for information on the format of this list). If *option* is specified with no *value*, then the command returns a list describing the one named option (this list will be identical to the corresponding sublist of the value returned if no *option* is specified). If one or more *option−value* pairs are specified, then the command modifies the given widget option(s) to have the given value(s); in this case the command returns an empty string. *Option* may have any of the values accepted by the **scrollbar** command.

pathName **delta** *deltaX deltaY*

Returns a real number indicating the fractional change in the scrollbar setting that corresponds to a given change in slider position. For example, if the scrollbar is horizontal, the result indicates how much the scrollbar setting must change to move the slider *deltaX* pixels to the right (*deltaY* is ignored in this case). If the scrollbar is vertical, the result indicates how much the scrollbar setting must change to move the slider *deltaY* pixels down. The arguments and the result may be zero or negative.

pathName **fraction** *x y*

Returns a real number between 0 and 1 indicating where the point given by *x* and *y* lies in the trough area of the scrollbar. The value 0 corresponds to the top or left of the trough, the value 1 corresponds to the bottom or right, 0.5 corresponds to the middle, and so on. *X* and *y* must be pixel coordinates relative to the scrollbar widget. If *x* and *y* refer to a point outside the trough, the closest point in the trough is used.

pathName **get**

Returns the scrollbar settings in the form of a list whose elements are the arguments to the most recent **set** widget command.

pathName **identify** *x y*

Returns the name of the element under the point given by *x* and *y* (such as **arrow1**), or an empty string if the point does not lie in any element of the scrollbar. *X* and *y* must be pixel coordinates relative to the scrollbar widget.

pathName **set** *first last*

This command is invoked by the scrollbar's associated widget to tell the scrollbar about the current view in the widget. The command takes two arguments, each of which is a real fraction between 0 and 1. The fractions describe the range of the document that is visible in the associated widget. For example, if *first* is 0.2 and *last* is 0.4, it means that the first part of the document visible in the window is 20% of the way through the document, and the last visible part is 40% of the way through.

SCROLLING COMMANDS

When the user interacts with the scrollbar, for example by dragging the slider, the scrollbar notifies the associated widget that it must change its view. The scrollbar makes the notification by evaluating a Tcl command generated from the scrollbar's **−command** option. The command may take any of the following forms. In each case, *prefix* is the contents of the **−command** option, which usually has a form like **.t yview**

prefix **moveto** *fraction*

Fraction is a real number between 0 and 1. The widget should adjust its view so that the point given by *fraction* appears at the beginning of the widget. If *fraction* is 0 it refers to the beginning of the document. 1.0 refers to the end of the document, 0.333 refers to a point one-third of the way through the document, and so on.

prefix **scroll** *number* **units**

The widget should adjust its view by *number* units. The units are defined in whatever way makes sense for the widget, such as characters or lines in a text widget. *Number* is either 1, which means one unit should scroll off the top or left of the window, or −1, which means that one unit should scroll off the bottom or right of the window.

prefix **scroll** *number* **pages**

The widget should adjust its view by *number* pages. It is up to the widget to define the meaning of a page; typically it is slightly less than what fits in the window, so that there is a slight overlap between the old and new views. *Number* is either 1, which means the next page should become visible, or −1, which means that the previous page should become visible.

OLD COMMAND SYNTAX

In versions of Tk before 4.0, the **set** and **get** widget commands used a different form. This form is still supported for backward compatibility, but it is deprecated. In the old command syntax, the **set** widget command has the following form:

pathName **set** *totalUnits windowUnits firstUnit lastUnit*

In this form the arguments are all integers. *TotalUnits* gives the total size of the object being

displayed in the associated widget. The meaning of one unit depends on the associated widget; for example, in a text editor widget units might correspond to lines of text. *WindowUnits* indicates the total number of units that can fit in the associated window at one time. *FirstUnit* and *lastUnit* give the indices of the first and last units currently visible in the associated window (zero corresponds to the first unit of the object).

Under the old syntax the **get** widget command returns a list of four integers, consisting of the *totalUnits*, *windowUnits*, *firstUnit*, and *lastUnit* values from the last **set** widget command.

The commands generated by scrollbars also have a different form when the old syntax is being used:

prefix unit

Unit is an integer that indicates what should appear at the top or left of the associated widget's window. It has the same meaning as the *firstUnit* and *lastUnit* arguments to the **set** widget command.

The most recent **set** widget command determines whether or not to use the old syntax. If it is given two real arguments then the new syntax will be used in the future, and if it is given four integer arguments then the old syntax will be used.

BINDINGS

Tk automatically creates class bindings for scrollbars that give them the following default behavior. If the behavior is different for vertical and horizontal scrollbars, the horizontal behavior is described in parentheses.

[1] Pressing button 1 over **arrow1** causes the view in the associated widget to shift up (left) by one unit so that the document appears to move down (right) one unit. If the button is held down, the action auto-repeats.

[2] Pressing button 1 over **trough1** causes the view in the associated widget to shift up (left) by one screenful so that the document appears to move down (right) one screenful. If the button is held down, the action auto-repeats.

[3] Pressing button 1 over the slider and dragging causes the view to drag with the slider. If the **jump** option is true, then the view doesn't drag along with the slider; it changes only when the mouse button is released.

[4] Pressing button 1 over **trough2** causes the view in the associated widget to shift down (right) by one screenful so that the document appears to move up (left) one screenful. If the button is held down, the action auto-repeats.

[5] Pressing button 1 over **arrow2** causes the view in the associated widget to shift down (right) by one unit so that the document appears to move up (left) one unit. If the button is held down, the action auto-repeats.

[6] If button 2 is pressed over the trough or the slider, it sets the view to correspond to the mouse position; dragging the mouse with button 2 down causes the view to drag with the mouse. If button 2 is pressed over one of the arrows, it causes the same behavior as pressing button 1.

[7] If button 1 is pressed with the Control key down, then if the mouse is over **arrow1** or **trough1** the view changes to the very top (left) of the document; if the mouse is over **arrow2** or **trough2** the view changes to the very bottom (right) of the document; if the mouse is anywhere else then the button press has no effect.

[8] In vertical scrollbars the Up and Down keys have the same behavior as mouse clicks over **arrow1** and **arrow2**, respectively. In horizontal scrollbars these keys have no effect.

[9] In vertical scrollbars Control-Up and Control-Down have the same behavior as mouse clicks over

trough1 and **trough2**, respectively. In horizontal scrollbars these keys have no effect.

[10] In horizontal scrollbars the Up and Down keys have the same behavior as mouse clicks over **arrow1** and **arrow2**, respectively. In vertical scrollbars these keys have no effect.

[11] In horizontal scrollbars Control-Up and Control-Down have the same behavior as mouse clicks over **trough1** and **trough2**, respectively. In vertical scrollbars these keys have no effect.

[12] The Prior and Next keys have the same behavior as mouse clicks over **trough1** and **trough2**, respectively.

[13] The Home key adjusts the view to the top (left edge) of the document.

[14] The End key adjusts the view to the bottom (right edge) of the document.

KEYWORDS

scrollbar, widget

NAME

selection − Manipulate the X selection

SYNOPSIS

selection *option* ?*arg arg ...*?

DESCRIPTION

This command provides a Tcl interface to the X selection mechanism and implements the full selection functionality described in the X Inter-Client Communication Conventions Manual (ICCCM).

The first argument to **selection** determines the format of the rest of the arguments and the behavior of the command. The following forms are currently supported:

selection clear ?**−displayof** *window*? ?**−selection** *selection*?

If *selection* exists anywhere on *window*'s display, clear it so that no window owns the selection anymore. *Selection* specifies the X selection that should be cleared, and should be an atom name such as PRIMARY or CLIPBOARD; see the Inter-Client Communication Conventions Manual for complete details. *Selection* defaults to PRIMARY and *window* defaults to ''.''. Returns an empty string.

selection get ?**−displayof** *window*? ?**−selection** *selection*? ?**−type** *type*?

Retrieves the value of *selection* from *window*'s display and returns it as a result. *Selection* defaults to PRIMARY and *window* defaults to ''.''. *Type* specifies the form in which the selection is to be returned (the desired ''target'' for conversion, in ICCCM terminology), and should be an atom name such as STRING or FILE_NAME; see the Inter-Client Communication Conventions Manual for complete details. *Type* defaults to STRING. The selection owner may choose to return the selection in any of several different representation formats, such as STRING, ATOM, INTEGER, etc. (this format is different than the selection type; see the ICCCM for all the confusing details). If the selection is returned in a non-string format, such as INTEGER or ATOM, the **selection** command converts it to string format as a collection of fields separated by spaces: atoms are converted to their textual names, and anything else is converted to hexadecimal integers.

selection handle ?**−selection** *selection*? ?**−type** *type*? ?**−format** *format*? *window command*

Creates a handler for selection requests, such that *command* will be executed whenever *selection* is owned by *window* and someone attempts to retrieve it in the form given by *type* (e.g. *type* is specified in the **selection get** command). *Selection* defaults to PRIMARY, *type* defaults to STRING, and *format* defaults to STRING. If *command* is an empty string then any existing handler for *window*, *type*, and *selection* is removed.

When *selection* is requested, *window* is the selection owner, and *type* is the requested type, *command* will be executed as a Tcl command with two additional numbers appended to it (with space separators). The two additional numbers are *offset* and *maxBytes*: *offset* specifies a starting character position in the selection and *maxBytes* gives the maximum number of bytes to retrieve. The command should return a value consisting of at most *maxBytes* of the selection, starting at position *offset*. For very large selections (larger than *maxBytes*) the selection will be retrieved using several invocations of *command* with increasing *offset* values. If *command* returns a string whose length is less than *maxBytes*, the return value is assumed to include all of the remainder of the selection; if the length of *command*'s result is equal to *maxBytes* then *command* will be invoked again, until it eventually returns a result shorter than *maxBytes*. The value of *maxBytes* will always be relatively large (thousands of bytes).

If *command* returns an error then the selection retrieval is rejected just as if the selection didn't exist at all.

The *format* argument specifies the representation that should be used to transmit the selection to the requester (the second column of Table 2 of the ICCCM), and defaults to STRING. If *format* is STRING, the selection is transmitted as 8-bit ASCII characters (i.e. just in the form returned by *command*). If *format* is ATOM, then the return value from *command* is divided into fields separated by white space; each field is converted to its atom value, and the 32-bit atom value is transmitted instead of the atom name. For any other *format*, the return value from *command* is divided into fields separated by white space and each field is converted to a 32-bit integer; an array of integers is transmitted to the selection requester.

The *format* argument is needed only for compatibility with selection requesters that don't use Tk. If Tk is being used to retrieve the selection then the value is converted back to a string at the requesting end, so *format* is irrelevant.

selection own ?**−displayof** *window*? ?**−selection** *selection*?

selection own ?**−command** *command*? ?**−selection** *selection*? *window*

The first form of **selection own** returns the path name of the window in this application that owns *selection* on the display containing *window*, or an empty string if no window in this application owns the selection. *Selection* defaults to PRIMARY and *window* defaults to ''.''.

The second form of **selection own** causes *window* to become the new owner of *selection* on *window*'s display, returning an empty string as result. The existing owner, if any, is notified that it has lost the selection. If *command* is specified, it is a Tcl script to execute when some other window claims ownership of the selection away from *window*. *Selection* defaults to PRIMARY.

KEYWORDS

clear, format, handler, ICCCM, own, selection, target, type

NAME

send − Execute a command in a different application

SYNOPSIS

send ?*options*? *app cmd* ?*arg arg ...*?

DESCRIPTION

This command arranges for *cmd* (and *arg*s) to be executed in the application named by *app*. It returns the result or error from that command execution. *App* may be the name of any application whose main window is on the display containing the sender's main window; it need not be within the same process. If no *arg* arguments are present, then the command to be executed is contained entirely within the *cmd* argument. If one or more *arg*s are present, they are concatenated to form the command to be executed, just as for the **eval** command.

If the initial arguments of the command begin with ''−'' they are treated as options. The following options are currently defined:

−async Requests asynchronous invocation. In this case the **send** command will complete immediately without waiting for *cmd* to complete in the target application; no result will be available and errors in the sent command will be ignored. If the target application is in the same process as the sending application then the **−async** option is ignored.

−displayof *pathName*

Specifies that the target application's main window is on the display of the window given by *pathName*, instead of the display containing the application's main window.

− − Serves no purpose except to terminate the list of options. This option is needed only if *app* could contain a leading ''−'' character.

APPLICATION NAMES

The name of an application is set initially from the name of the program or script that created the application. You can query and change the name of an application with the **tk appname** command.

DISABLING SENDS

If the **send** command is removed from an application (e.g. with the command **rename send {}**) then the application will not respond to incoming send requests anymore, nor will it be able to issue outgoing requests. Communication can be reenabled by invoking the **tk appname** command.

SECURITY

The **send** command is potentially a serious security loophole, since any application that can connect to your X server can send scripts to your applications. These incoming scripts can use Tcl to read and write your files and invoke subprocesses under your name. Host-based access control such as that provided by **xhost** is particularly insecure, since it allows anyone with an account on particular hosts to connect to your server, and if disabled it allows anyone anywhere to connect to your server. In order to provide at least a small amount of security, Tk checks the access control being used by the server and rejects incoming sends unless (a) **xhost**-style access control is enabled (i.e. only certain hosts can establish connections) and (b) the list of enabled hosts is empty. This means that applications cannot connect to your server unless they use some other form of authorization such as that provide by **xauth**.

KEYWORDS

application, name, remote execution, security, send

NAME

text − Create and manipulate text widgets

SYNOPSIS

text *pathName* ?*options*?

STANDARD OPTIONS

−background	**−highlightbackground**	**−insertontime**	**−selectborderwidth**
−borderwidth	**−highlightcolor**	**−insertwidth**	**−selectforeground**
−cursor	**−highlightthickness**	**−padx**	**−setgrid**
−exportselection	**−insertbackground**	**−pady**	**−takefocus**
−font	**−insertborderwidth**	**−relief**	**−xscrollcommand**
−foreground	**−insertofftime**	**−selectbackground**	**−yscrollcommand**

See the **options** manual entry for details on the standard options.

WIDGET-SPECIFIC OPTIONS

Command-Line Name: **−height**
Database Name: **height**
Database Class: **Height**

Specifies the desired height for the window, in units of characters in the font given by the **−font** option. Must be at least one.

Command-Line Name: **−spacing1**
Database Name: **spacing1**
Database Class: **Spacing1**

Requests additional space above each text line in the widget, using any of the standard forms for screen distances. If a line wraps, this option only applies to the first line on the display. This option may be overriden with **−spacing1** options in tags.

Command-Line Name: **−spacing2**
Database Name: **spacing2**
Database Class: **Spacing2**

For lines that wrap (so that they cover more than one line on the display) this option specifies additional space to provide between the display lines that represent a single line of text. The value may have any of the standard forms for screen distances. This option may be overriden with **−spacing2** options in tags.

Command-Line Name: **−spacing3**
Database Name: **spacing3**
Database Class: **Spacing3**

Requests additional space below each text line in the widget, using any of the standard forms for screen distances. If a line wraps, this option only applies to the last line on the display. This option may be overriden with **−spacing3** options in tags.

Command-Line Name: **−state**
Database Name: **state**
Database Class: **State**

Specifies one of two states for the text: **normal** or **disabled**. If the text is disabled then characters may not be inserted or deleted and no insertion cursor will be displayed, even if the input focus is in the widget.

Command-Line Name: **−tabs**
Database Name: **tabs**
Database Class: **Tabs**

Specifies a set of tab stops for the window. The option's value consists of a list of screen distances giving the positions of the tab stops. Each position may optionally be followed in the next list element by one of the keywords **left**, **right**, **center**, or **numeric**, which specifies how to justify text relative to the tab stop. **Left** is the default; it causes the text following the tab character to be positioned with its left edge at the tab position. **Right** means that the right edge of the text following the tab character is positioned at the tab position, and **center** means that the text is centered at the tab position. **Numeric** means that the decimal point in the text is positioned at the tab position; if there is no decimal point then the least significant digit of the number is positioned just to the left of the tab position; if there is no number in the text then the text is right-justified at the tab position. For example, **−tabs {2c left 4c 6c center}** creates three tab stops at two-centimeter intervals; the first two use left justification and the third uses center justification. If the list of tab stops does not have enough elements to cover all of the tabs in a text line, then Tk extrapolates new tab stops using the spacing and alignment from the last tab stop in the list. The value of the **tabs** option may be overridden by **−tabs** options in tags. If no **−tabs** option is specified, or if it is specified as an empty list, then Tk uses default tabs spaced every eight (average size) characters.

Command-Line Name: **−width**
Database Name: **width**
Database Class: **Width**

Specifies the desired width for the window in units of characters in the font given by the **−font** option. If the font doesn't have a uniform width then the width of the character "0" is used in translating from character units to screen units.

Command-Line Name: **−wrap**
Database Name: **wrap**
Database Class: **Wrap**

Specifies how to handle lines in the text that are too long to be displayed in a single line of the text's window. The value must be **none** or **char** or **word**. A wrap mode of **none** means that each line of text appears as exactly one line on the screen; extra characters that don't fit on the screen are not displayed. In the other modes each line of text will be broken up into several screen lines if necessary to keep all the characters visible. In **char** mode a screen line break may occur after any character; in **word** mode a line break will only be made at word boundaries.

DESCRIPTION

The **text** command creates a new window (given by the *pathName* argument) and makes it into a text widget. Additional options, described above, may be specified on the command line or in the option database to configure aspects of the text such as its default background color and relief. The **text** command returns the path name of the new window.

A text widget displays one or more lines of text and allows that text to be edited. Text widgets support four different kinds of annotations on the text, called tags, marks, embedded windows or embedded images. Tags allow different portions of the text to be displayed with different fonts and colors. In addition, Tcl commands can be associated with tags so that scripts are invoked when particular actions such as keystrokes and mouse button presses occur in particular ranges of the text. See TAGS below for more details.

The second form of annotation consists of marks, which are floating markers in the text. Marks are used to keep track of various interesting positions in the text as it is edited. See MARKS below for more details.

The third form of annotation allows arbitrary windows to be embedded in a text widget. See EMBEDDED WINDOWS below for more details.

The fourth form of annotation allows Tk images to be embedded in a text widget. See EMBEDDED IMAGES below for more details.

INDICES

Many of the widget commands for texts take one or more indices as arguments. An index is a string used to indicate a particular place within a text, such as a place to insert characters or one endpoint of a range of characters to delete. Indices have the syntax

base modifier modifier modifier ...

Where *base* gives a starting point and the *modifier*s adjust the index from the starting point (e.g. move forward or backward one character). Every index must contain a *base*, but the *modifier*s are optional.

The *base* for an index must have one of the following forms:

*line***.***char* Indicates *char*'th character on line *line*. Lines are numbered from 1 for consistency with other UNIX programs that use this numbering scheme. Within a line, characters are numbered from 0. If *char* is **end** then it refers to the newline character that ends the line.

@*x***,***y* Indicates the character that covers the pixel whose x and y coordinates within the text's window are *x* and *y*.

end Indicates the end of the text (the character just after the last newline).

mark Indicates the character just after the mark whose name is *mark*.

*tag***.first** Indicates the first character in the text that has been tagged with *tag*. This form generates an error if no characters are currently tagged with *tag*.

*tag***.last** Indicates the character just after the last one in the text that has been tagged with *tag*. This form generates an error if no characters are currently tagged with *tag*.

pathName Indicates the position of the embedded window whose name is *pathName*. This form generates an error if there is no embedded window by the given name.

imageName Indicates the position of the embedded image whose name is *imageName*. This form generates an error if there is no embedded image by the given name.

If the *base* could match more than one of the above forms, such as a *mark* and *imageName* both having the same value, then the form earlier in the above list takes precedence. If modifiers follow the base index, each one of them must have one of the forms listed below. Keywords such as **chars** and **wordend** may be abbreviated as long as the abbreviation is unambiguous.

+ *count* **chars**

Adjust the index forward by *count* characters, moving to later lines in the text if necessary. If there are fewer than *count* characters in the text after the current index, then set the index to the last character in the text. Spaces on either side of *count* are optional.

− *count* **chars**

Adjust the index backward by *count* characters, moving to earlier lines in the text if necessary. If there are fewer than *count* characters in the text before the current index, then set the index to the first character in the text. Spaces on either side of *count* are optional.

+ *count* **lines**

Adjust the index forward by *count* lines, retaining the same character position within the line. If there are fewer than *count* lines after the line containing the current index, then set the index to refer to the same character position on the last line of the text. Then, if the line is not long enough to contain a character at the indicated character position, adjust the character position to refer to the last character of the line (the newline). Spaces on either side of *count* are optional.

− *count* **lines**

Adjust the index backward by *count* lines, retaining the same character position within the line. If

there are fewer than *count* lines before the line containing the current index, then set the index to refer to the same character position on the first line of the text. Then, if the line is not long enough to contain a character at the indicated character position, adjust the character position to refer to the last character of the line (the newline). Spaces on either side of *count* are optional.

linestart
Adjust the index to refer to the first character on the line.

lineend Adjust the index to refer to the last character on the line (the newline).

wordstart
Adjust the index to refer to the first character of the word containing the current index. A word consists of any number of adjacent characters that are letters, digits, or underscores, or a single character that is not one of these.

wordend
Adjust the index to refer to the character just after the last one of the word containing the current index. If the current index refers to the last character of the text then it is not modified.

If more than one modifier is present then they are applied in left-to-right order. For example, the index "**end − 1 chars**" refers to the next-to-last character in the text and "**insert wordstart − 1 c**" refers to the character just before the first one in the word containing the insertion cursor.

TAGS

The first form of annotation in text widgets is a tag. A tag is a textual string that is associated with some of the characters in a text. Tags may contain arbitrary characters, but it is probably best to avoid using the the characters " " (space), +, or −: these characters have special meaning in indices, so tags containing them can't be used as indices. There may be any number of tags associated with characters in a text. Each tag may refer to a single character, a range of characters, or several ranges of characters. An individual character may have any number of tags associated with it.

A priority order is defined among tags, and this order is used in implementing some of the tag-related functions described below. When a tag is defined (by associating it with characters or setting its display options or binding commands to it), it is given a priority higher than any existing tag. The priority order of tags may be redefined using the "*pathName* **tag raise**" and "*pathName* **tag lower**" widget commands.

Tags serve three purposes in text widgets. First, they control the way information is displayed on the screen. By default, characters are displayed as determined by the **background**, **font**, and **foreground** options for the text widget. However, display options may be associated with individual tags using the "*pathName* **tag configure**" widget command. If a character has been tagged, then the display options associated with the tag override the default display style. The following options are currently supported for tags:

−background *color*
Color specifies the background color to use for characters associated with the tag. It may have any of the forms accepted by **Tk_GetColor**.

−bgstipple *bitmap*
Bitmap specifies a bitmap that is used as a stipple pattern for the background. It may have any of the forms accepted by **Tk_GetBitmap**. If *bitmap* hasn't been specified, or if it is specified as an empty string, then a solid fill will be used for the background.

−borderwidth *pixels*
Pixels specifies the width of a 3-D border to draw around the background. It may have any of the forms accepted by **Tk_GetPixels**. This option is used in conjunction with the **−relief** option to give a 3-D appearance to the background for characters; it is ignored unless the **−background** option has been set for the tag.

−fgstipple *bitmap*
Bitmap specifies a bitmap that is used as a stipple pattern when drawing text and other foreground information such as underlines. It may have any of the forms accepted by **Tk_GetBitmap**. If *bitmap* hasn't been specified, or if it is specified as an empty string, then a solid fill will be used.

−font *fontName*
FontName is the name of a font to use for drawing characters. It may have any of the forms accepted by **Tk_GetFontStruct**.

−foreground *color*
Color specifies the color to use when drawing text and other foreground information such as underlines. It may have any of the forms accepted by **Tk_GetColor**.

−justify *justify*
If the first character of a display line has a tag for which this option has been specified, then *justify* determines how to justify the line. It must be one of **left**, **right**, or **center**. If a line wraps, then the justification for each line on the display is determined by the first character of that display line.

−lmargin1 *pixels*
If the first character of a text line has a tag for which this option has been specified, then *pixels* specifies how much the line should be indented from the left edge of the window. *Pixels* may have any of the standard forms for screen distances. If a line of text wraps, this option only applies to the first line on the display; the **−lmargin2** option controls the indentation for subsequent lines.

−lmargin2 *pixels*
If the first character of a display line has a tag for which this option has been specified, and if the display line is not the first for its text line (i.e., the text line has wrapped), then *pixels* specifies how much the line should be indented from the left edge of the window. *Pixels* may have any of the standard forms for screen distances. This option is only used when wrapping is enabled, and it only applies to the second and later display lines for a text line.

−offset *pixels*
Pixels specifies an amount by which the text's baseline should be offset vertically from the baseline of the overall line, in pixels. For example, a positive offset can be used for superscripts and a negative offset can be used for subscripts. *Pixels* may have any of the standard forms for screen distances.

−overstrike *boolean*
Specifies whether or not to draw a horizontal rule through the middle of characters. *Boolean* may have any of the forms accepted by **Tk_GetBoolean**.

−relief *relief*
Relief specifies the 3-D relief to use for drawing backgrounds, in any of the forms accepted by **Tk_GetRelief**. This option is used in conjunction with the **−borderwidth** option to give a 3-D appearance to the background for characters; it is ignored unless the **−background** option has been set for the tag.

−rmargin *pixels*
If the first character of a display line has a tag for which this option has been specified, then *pixels* specifies how wide a margin to leave between the end of the line and the right edge of the window. *Pixels* may have any of the standard forms for screen distances. This option is only used when wrapping is enabled. If a text line wraps, the right margin for each line on the display is determined by the first character of that display line.

−spacing1 *pixels*
Pixels specifies how much additional space should be left above each text line, using any of the standard forms for screen distances. If a line wraps, this option only applies to the first line on the display.

−spacing2 *pixels*
For lines that wrap, this option specifies how much additional space to leave between the display lines for a single text line. *Pixels* may have any of the standard forms for screen distances.

−spacing3 *pixels*
Pixels specifies how much additional space should be left below each text line, using any of the standard forms for screen distances. If a line wraps, this option only applies to the last line on the display.

−tabs *tabList*
TabList specifies a set of tab stops in the same form as for the **−tabs** option for the text widget. This option only applies to a display line if it applies to the first character on that display line. If this option is specified as an empty string, it cancels the option, leaving it unspecified for the tag (the default). If the option is specified as a non-empty string that is an empty list, such as **−tags { }**, then it requests default 8-character tabs as described for the **tags** widget option.

−underline *boolean*
Boolean specifies whether or not to draw an underline underneath characters. It may have any of the forms accepted by **Tk_GetBoolean**.

−wrap *mode*
Mode specifies how to handle lines that are wider than the text's window. It has the same legal values as the **−wrap** option for the text widget: **none**, **char**, or **word**. If this tag option is specified, it overrides the **−wrap** option for the text widget.

If a character has several tags associated with it, and if their display options conflict, then the options of the highest priority tag are used. If a particular display option hasn't been specified for a particular tag, or if it is specified as an empty string, then that option will never be used; the next-highest-priority tag's option will used instead. If no tag specifies a particular display option, then the default style for the widget will be used.

The second purpose for tags is event bindings. You can associate bindings with a tag in much the same way you can associate bindings with a widget class: whenever particular X events occur on characters with the given tag, a given Tcl command will be executed. Tag bindings can be used to give behaviors to ranges of characters; among other things, this allows hypertext-like features to be implemented. For details, see the description of the **tag bind** widget command below.

The third use for tags is in managing the selection. See THE SELECTION below.

MARKS

The second form of annotation in text widgets is a mark. Marks are used for remembering particular places in a text. They are something like tags, in that they have names and they refer to places in the file, but a mark isn't associated with particular characters. Instead, a mark is associated with the gap between two characters. Only a single position may be associated with a mark at any given time. If the characters around a mark are deleted the mark will still remain; it will just have new neighbor characters. In contrast, if the characters containing a tag are deleted then the tag will no longer have an association with characters in the file. Marks may be manipulated with the ''*pathName* **mark**'' widget command, and their current locations may be determined by using the mark name as an index in widget commands.

Each mark also has a *gravity*, which is either **left** or **right**. The gravity for a mark specifies what happens to the mark when text is inserted at the point of the mark. If a mark has left gravity, then the mark is treated as if it were attached to the character on its left, so the mark will remain to the left of any text inserted at the mark position. If the mark has right gravity, new text inserted at the mark position will appear to the right of the mark. The gravity for a mark defaults to **right**.

The name space for marks is different from that for tags: the same name may be used for both a mark and a tag, but they will refer to different things.

Two marks have special significance. First, the mark **insert** is associated with the insertion cursor, as described under THE INSERTION CURSOR below. Second, the mark **current** is associated with the character closest to the mouse and is adjusted automatically to track the mouse position and any changes to the text in the widget (one exception: **current** is not updated in response to mouse motions if a mouse button is down; the update will be deferred until all mouse buttons have been released). Neither of these special marks may be deleted.

EMBEDDED WINDOWS

The third form of annotation in text widgets is an embedded window. Each embedded window annotation causes a window to be displayed at a particular point in the text. There may be any number of embedded windows in a text widget, and any widget may be used as an embedded window (subject to the usual rules for geometry management, which require the text window to be the parent of the embedded window or a descendant of its parent). The embedded window's position on the screen will be updated as the text is modified or scrolled, and it will be mapped and unmapped as it moves into and out of the visible area of the text widget. Each embedded window occupies one character's worth of index space in the text widget, and it may be referred to either by the name of its embedded window or by its position in the widget's index space. If the range of text containing the embedded window is deleted then the window is destroyed.

When an embedded window is added to a text widget with the **window create** widget command, several configuration options may be associated with it. These options may be modified later with the **window configure** widget command. The following options are currently supported:

−align *where*
: If the window is not as tall as the line in which it is displayed, this option determines where the window is displayed in the line. *Where* must have one of the values **top** (align the top of the window with the top of the line), **center** (center the window within the range of the line), **bottom** (align the bottom of the window with the bottom of the line's area), or **baseline** (align the bottom of the window with the baseline of the line).

−create *script*
: Specifies a Tcl script that may be evaluated to create the window for the annotation. If no **−window** option has been specified for the annotation this script will be evaluated when the annotation is about to be displayed on the screen. *Script* must create a window for the annotation and return the name of that window as its result. If the annotation's window should ever be deleted, *script* will be evaluated again the next time the annotation is displayed.

−padx *pixels*
: *Pixels* specifies the amount of extra space to leave on each side of the embedded window. It may have any of the usual forms defined for a screen distance.

−pady *pixels*
: *Pixels* specifies the amount of extra space to leave on the top and on the bottom of the embedded window. It may have any of the usual forms defined for a screen distance.

−stretch *boolean*
: If the requested height of the embedded window is less than the height of the line in which it is displayed, this option can be used to specify whether the window should be stretched vertically to fill its line. If the **−pady** option has been specified as well, then the requested padding will be retained even if the window is stretched.

−window *pathName*
: Specifies the name of a window to display in the annotation.

EMBEDDED IMAGES

The final form of annotation in text widgets is an embedded image. Each embedded image annotation causes an image to be displayed at a particular point in the text. There may be any number of embedded images in a text widget, and a particular image may be embedded in multiple places in the same text widget. The embedded image's position on the screen will be updated as the text is modified or scrolled. Each embedded image occupies one character's worth of index space in the text widget, and it may be referred to either by its position in the widget's index space, or the name it is assigned when the image is inserted into the text widget widh **image create**. If the range of text containing the embedded image is deleted then that copy of the image is removed from the screen.

When an embedded image is added to a text widget with the **image create** widget command, a name unique to this instance of the image is returned. This name may then be used to refer to this image instance. The name is taken to be the value of the **-name** option (described below). If the **-name** option is not provided, the **-image** name is used instead. If the *imageName* is already in use in the text widget, then *#nn* is added to the end of the *imageName*, where *nn* is an arbitrary integer. This insures the *imageName* is unique. Once this name is assigned to this instance of the image, it does not change, even though the **-image** or **-name** values can be changed with **image configure**.

When an embedded image is added to a text widget with the **image create** widget command, several configuration options may be associated with it. These options may be modified later with the **image configure** widget command. The following options are currently supported:

−align *where*
: If the image is not as tall as the line in which it is displayed, this option determines where the image is displayed in the line. *Where* must have one of the values **top** (align the top of the image with the top of the line), **center** (center the image within the range of the line), **bottom** (align the bottom of the image with the bottom of the line's area), or **baseline** (align the bottom of the image with the baseline of the line).

−image *image*
: Specifies the name of the Tk image to display in the annotation. If *image* is not a valid Tk image, then an error is returned.

−name *ImageName*
: Specifies the name by which this image instance may be referenced in the text widget. If *ImageName* is not supplied, then the name of the Tk image is used instead. If the *imageName* is already in use, *#nn* is appended to the end of the name as described above.

−padx *pixels*
: *Pixels* specifies the amount of extra space to leave on each side of the embedded image. It may have any of the usual forms defined for a screen distance.

−pady *pixels*
: *Pixels* specifies the amount of extra space to leave on the top and on the bottom of the embedded image. It may have any of the usual forms defined for a screen distance.

THE SELECTION

Selection support is implemented via tags. If the **exportSelection** option for the text widget is true then the **sel** tag will be associated with the selection:

[1] Whenever characters are tagged with **sel** the text widget will claim ownership of the selection.

[2] Attempts to retrieve the selection will be serviced by the text widget, returning all the characters with the **sel** tag.

[3] If the selection is claimed away by another application or by another window within this application, then the **sel** tag will be removed from all characters in the text.

The **sel** tag is automatically defined when a text widget is created, and it may not be deleted with the ''*pathName* **tag delete**'' widget command. Furthermore, the **selectBackground**, **selectBorderWidth**, and **selectForeground** options for the text widget are tied to the **−background**, **−borderwidth**, and **−foreground** options for the **sel** tag: changes in either will automatically be reflected in the other.

THE INSERTION CURSOR

The mark named **insert** has special significance in text widgets. It is defined automatically when a text widget is created and it may not be unset with the ''*pathName* **mark unset**'' widget command. The **insert** mark represents the position of the insertion cursor, and the insertion cursor will automatically be drawn at this point whenever the text widget has the input focus.

WIDGET COMMAND

The **text** command creates a new Tcl command whose name is the same as the path name of the text's window. This command may be used to invoke various operations on the widget. It has the following general form:

pathName option ?*arg arg ...*?

PathName is the name of the command, which is the same as the text widget's path name. *Option* and the *arg*s determine the exact behavior of the command. The following commands are possible for text widgets:

pathName **bbox** *index*

Returns a list of four elements describing the screen area of the character given by *index*. The first two elements of the list give the x and y coordinates of the upper-left corner of the area occupied by the character, and the last two elements give the width and height of the area. If the character is only partially visible on the screen, then the return value reflects just the visible part. If the character is not visible on the screen then the return value is an empty list.

pathName **cget** *option*

Returns the current value of the configuration option given by *option*. *Option* may have any of the values accepted by the **text** command.

pathName **compare** *index1 op index2*

Compares the indices given by *index1* and *index2* according to the relational operator given by *op*, and returns 1 if the relationship is satisfied and 0 if it isn't. *Op* must be one of the operators <, <=, ==, >=, >, or !=. If *op* is == then 1 is returned if the two indices refer to the same character, if *op* is < then 1 is returned if *index1* refers to an earlier character in the text than *index2*, and so on.

pathName **configure** ?*option*? *?value option value ...*?

Query or modify the configuration options of the widget. If no *option* is specified, returns a list describing all of the available options for *pathName* (see **Tk_ConfigureInfo** for information on the format of this list). If *option* is specified with no *value*, then the command returns a list describing the one named option (this list will be identical to the corresponding sublist of the value returned if no *option* is specified). If one or more *option−value* pairs are specified, then the command modifies the given widget option(s) to have the given value(s); in this case the command returns an empty string. *Option* may have any of the values accepted by the **text** command.

pathName **debug** ?*boolean*?

If *boolean* is specified, then it must have one of the true or false values accepted by Tcl_GetBoolean. If the value is a true one then internal consistency checks will be turned on in the B-tree code associated with text widgets. If *boolean* has a false value then the debugging checks will be turned off. In either case the command returns an empty string. If *boolean* is not specified then the command returns **on** or **off** to indicate whether or not debugging is turned on. There is a single debugging switch shared by all text widgets: turning debugging on or off in any widget turns it on or off for all widgets. For widgets with large amounts of text, the consistency checks may cause a noticeable slow-down.

pathName **delete** *index1* ?*index2*?

Delete a range of characters from the text. If both *index1* and *index2* are specified, then delete all the characters starting with the one given by *index1* and stopping just before *index2* (i.e. the character at *index2* is not deleted). If *index2* doesn't specify a position later in the text than *index1* then no characters are deleted. If *index2* isn't specified then the single character at *index1* is deleted. It is not allowable to delete characters in a way that would leave the text without a newline as the last character. The command returns an empty string.

pathName **dlineinfo** *index*

Returns a list with five elements describing the area occupied by the display line containing *index*. The first two elements of the list give the x and y coordinates of the upper-left corner of the area occupied by the line, the third and fourth elements give the width and height of the area, and the fifth element gives the position of the baseline for the line, measured down from the top of the area. All of this information is measured in pixels. If the current wrap mode is **none** and the line extends beyond the boundaries of the window, the area returned reflects the entire area of the line, including the portions that are out of the window. If the line is shorter than the full width of the window then the area returned reflects just the portion of the line that is occupied by characters and embedded windows. If the display line containing *index* is not visible on the screen then the return value is an empty list.

pathName **dump** ?*switches*? *index1* ?*index2*?

Return the contents of the text widget from *index1* up to, but not including *index2*, including the text and information about marks, tags, and embedded windows. If *index2* is not specified, then it defaults to one character past *index1*. The information is returned in the following format:

key1 value1 index1 key2 value2 index2 ...

The possible *key* values are **text**, **mark**, **tagon**, **tagoff**, and **window**. The corresponding *value* is the text, mark name, tag name, or window name. The *index* information is the index of the start of the text, the mark, the tag transition, or the window. One or more of the following switches (or abbreviations thereof) may be specified to control the dump:

−all Return information about all elements: text, marks, tags, and windows. This is the default.

−command *command*

Instead of returning the information as the result of the dump operation, invoke the *command* on each element of the text widget within the range. The command has three arguments appended to it before it is evaluated: the *key*, *value*, and *index*.

−mark Include information about marks in the dump results.

−tag Include information about tag transitions in the dump results. Tag information is returned as **tagon** and **tagoff** elements that indicate the begin and end of each range of each tag, respectively.

−text Include information about text in the dump results. The value is the text up to the next element or the end of range indicated by *index2*. A text element does not span newlines. A multi-line block of text that contains no marks or tag transitions will still be dumped as a set of text seqments that each end with a newline. The newline is part of the value.

−window

Include information about embedded windows in the dump results. The value of a window is its Tk pathname, unless the window has not been created yet. (It must have a create script.) In this case an empty string is returned, and you must query the window by its index position to get more information.

pathName **get** *index1* ?*index2*?

Return a range of characters from the text. The return value will be all the characters in the text starting with the one whose index is *index1* and ending just before the one whose index is *index2* (the character at *index2* will not be returned). If *index2* is omitted then the single character at *index1* is returned. If there are no characters in the specified range (e.g. *index1* is past the end of the file or *index2* is less than or equal to *index1*) then an empty string is returned. If the specified range contains embedded windows, no information about them is included in the returned string.

pathName **image** *option* ?*arg arg ...*?

This command is used to manipulate embedded images. The behavior of the command depends on the *option* argument that follows the **tag** argument. The following forms of the command are currently supported:

pathName **image cget** *index option*

Returns the value of a configuration option for an embedded image. *Index* identifies the embedded image, and *option* specifies a particular configuration option, which must be one of the ones listed in the section EMBEDDED IMAGES.

pathName **image configure** *index* ?*option value ...*?

Query or modify the configuration options for an embedded image. If no *option* is specified, returns a list describing all of the available options for the embedded image at *index* (see **Tk_ConfigureInfo** for information on the format of this list). If *option* is specified with no *value*, then the command returns a list describing the one named option (this list will be identical to the corresponding sublist of the value returned if no *option* is specified). If one or more *option−value* pairs are specified, then the command modifies the given option(s) to have the given value(s); in this case the command returns an empty string. See EMBEDDED IMAGES for information on the options that are supported.

pathName **image create** *index* ?*option value ...*?

This command creates a new image annotation, which will appear in the text at the position given by *index*. Any number of *option−value* pairs may be specified to configure the annotation. Returns a unique identifier that may be used as an index to refer to this image. See EMBEDDED IMAGES for information on the options that are supported, and a description of the identifier returned.

pathName **image names**

Returns a list whose elements are the names of all image instances currently embedded in *window*.

pathName **index** *index*

Returns the position corresponding to *index* in the form *line.char* where *line* is the line number and *char* is the character number. *Index* may have any of the forms described under INDICES above.

pathName **insert** *index chars* ?*tagList chars tagList ...*?

Inserts all of the *chars* arguments just before the character at *index*. If *index* refers to the end of the text (the character after the last newline) then the new text is inserted just before the last newline instead. If there is a single *chars* argument and no *tagList*, then the new text will receive any tags that are present on both the character before and the character after the insertion point; if a tag is present on only one of these characters then it will not be applied to the new text. If *tagList* is specified then it consists of a list of tag names; the new characters will receive all of the tags in this list and no others, regardless of the tags present around the insertion point. If multiple *chars−tagList* argument pairs are present, they produce the same effect as if a separate **insert** widget command had been issued for each pair, in order. The last *tagList* argument may be omitted.

pathName **mark** *option* ?*arg arg ...*?

This command is used to manipulate marks. The exact behavior of the command depends on the *option* argument that follows the **mark** argument. The following forms of the command are

currently supported:

pathName **mark gravity** *markName* ?*direction*?
If *direction* is not specified, returns **left** or **right** to indicate which of its adjacent characters *markName* is attached to. If *direction* is specified, it must be **left** or **right**; the gravity of *markName* is set to the given value.

pathName **mark names**
Returns a list whose elements are the names of all the marks that are currently set.

pathName **mark next** *index*
Returns the name of the next mark at or after *index*. If *index* is specified in numerical form, then the search for the next mark begins at that index. If *index* is the name of a mark, then the search for the next mark begins immediately after that mark. This can still return a mark at the same position if there are multiple marks at the same index. These semantics mean that the **mark next** operation can be used to step through all the marks in a text widget in the same order as the mark information returned by the **dump** operation. If a mark has been set to the special **end** index, then it appears to be *after* **end** with respect to the **mark next** operation. An empty string is returned if there are no marks after *index*.

pathName **mark previous** *index*
Returns the name of the mark at or before *index*. If *index* is specified in numerical form, then the search for the previous mark begins with the character just before that index. If *index* is the name of a mark, then the search for the next mark begins immediately before that mark. This can still return a mark at the same position if there are multiple marks at the same index. These semantics mean that the **mark previous** operation can be used to step through all the marks in a text widget in the reverse order as the mark information returned by the **dump** operation. An empty string is returned if there are no marks before *index*.

pathName **mark set** *markName index*
Sets the mark named *markName* to a position just before the character at *index*. If *markName* already exists, it is moved from its old position; if it doesn't exist, a new mark is created. This command returns an empty string.

pathName **mark unset** *markName* ?*markName markName ...*?
Remove the mark corresponding to each of the *markName* arguments. The removed marks will not be usable in indices and will not be returned by future calls to "*pathName* **mark names**". This command returns an empty string.

pathName **scan** *option args*
This command is used to implement scanning on texts. It has two forms, depending on *option*:

pathName **scan mark** *x y*
Records *x* and *y* and the current view in the text window, for use in conjunction with later **scan dragto** commands. Typically this command is associated with a mouse button press in the widget. It returns an empty string.

pathName **scan dragto** *x y*
This command computes the difference between its *x* and *y* arguments and the *x* and *y* arguments to the last **scan mark** command for the widget. It then adjusts the view by 10 times the difference in coordinates. This command is typically associated with mouse motion events in the widget, to produce the effect of dragging the text at high speed through the window. The return value is an empty string.

pathName **search** ?*switches*? *pattern index* ?*stopIndex*?
Searches the text in *pathName* starting at *index* for a range of characters that matches *pattern*. If a

match is found, the index of the first character in the match is returned as result; otherwise an empty string is returned. One or more of the following switches (or abbreviations thereof) may be specified to control the search:

−forwards
The search will proceed forward through the text, finding the first matching range starting at or after the position given by *index*. This is the default.

−backwards
The search will proceed backward through the text, finding the matching range closest to *index* whose first character is before *index*.

−exact Use exact matching: the characters in the matching range must be identical to those in *pattern*. This is the default.

−regexp
Treat *pattern* as a regular expression and match it against the text using the rules for regular expressions (see the **regexp** command for details).

−nocase
Ignore case differences between the pattern and the text.

−count *varName*
The argument following **−count** gives the name of a variable; if a match is found, the number of characters in the matching range will be stored in the variable.

− − This switch has no effect except to terminate the list of switches: the next argument will be treated as *pattern* even if it starts with **−**.

The matching range must be entirely within a single line of text. For regular expression matching the newlines are removed from the ends of the lines before matching: use the **$** feature in regular expressions to match the end of a line. For exact matching the newlines are retained. If *stopIndex* is specified, the search stops at that index: for forward searches, no match at or after *stopIndex* will be considered; for backward searches, no match earlier in the text than *stopIndex* will be considered. If *stopIndex* is omitted, the entire text will be searched: when the beginning or end of the text is reached, the search continues at the other end until the starting location is reached again; if *stopIndex* is specified, no wrap-around will occur.

pathName **see** *index*
Adjusts the view in the window so that the character given by *index* is completely visible. If *index* is already visible then the command does nothing. If *index* is a short distance out of view, the command adjusts the view just enough to make *index* visible at the edge of the window. If *index* is far out of view, then the command centers *index* in the window.

pathName **tag** *option* ?*arg arg ...*?
This command is used to manipulate tags. The exact behavior of the command depends on the *option* argument that follows the **tag** argument. The following forms of the command are currently supported:

pathName **tag add** *tagName index1* ?*index2 index1 index2 ...*?
Associate the tag *tagName* with all of the characters starting with *index1* and ending just before *index2* (the character at *index2* isn't tagged). A single command may contain any number of *index1*−*index2* pairs. If the last *index2* is omitted then the single character at *index1* is tagged. If there are no characters in the specified range (e.g. *index1* is past the end of the file or *index2* is less than or equal to *index1*) then the command has no effect.

pathName **tag bind** *tagName* ?*sequence*? ?*script*?
This command associates *script* with the tag given by *tagName*. Whenever the event sequence given by *sequence* occurs for a character that has been tagged with *tagName*,

the script will be invoked. This widget command is similar to the **bind** command except that it operates on characters in a text rather than entire widgets. See the **bind** manual entry for complete details on the syntax of *sequence* and the substitutions performed on *script* before invoking it. If all arguments are specified then a new binding is created, replacing any existing binding for the same *sequence* and *tagName* (if the first character of *script* is "+" then *script* augments an existing binding rather than replacing it). In this case the return value is an empty string. If *script* is omitted then the command returns the *script* associated with *tagName* and *sequence* (an error occurs if there is no such binding). If both *script* and *sequence* are omitted then the command returns a list of all the sequences for which bindings have been defined for *tagName*.

The only events for which bindings may be specified are those related to the mouse and keyboard (such as **Enter**, **Leave**, **ButtonPress**, **Motion**, and **KeyPress**) or virtual events. Event bindings for a text widget use the **current** mark described under MARKS above. An **Enter** event triggers for a tag when the tag first becomes present on the current character, and a **Leave** event triggers for a tag when it ceases to be present on the current character. **Enter** and **Leave** events can happen either because the **current** mark moved or because the character at that position changed. Note that these events are different than **Enter** and **Leave** events for windows. Mouse and keyboard events are directed to the current character. If a virtual event is used in a binding, that binding can trigger only if the virtual event is defined by an underlying mouse-related or keyboard-related event.

It is possible for the current character to have multiple tags, and for each of them to have a binding for a particular event sequence. When this occurs, one binding is invoked for each tag, in order from lowest-priority to highest priority. If there are multiple matching bindings for a single tag, then the most specific binding is chosen (see the manual entry for the **bind** command for details). **continue** and **break** commands within binding scripts are processed in the same way as for bindings created with the **bind** command.

If bindings are created for the widget as a whole using the **bind** command, then those bindings will supplement the tag bindings. The tag bindings will be invoked first, followed by bindings for the window as a whole.

pathName **tag cget** *tagName option*

This command returns the current value of the option named *option* associated with the tag given by *tagName*. *Option* may have any of the values accepted by the **tag configure** widget command.

pathName **tag configure** *tagName* ?*option*? ?*value*? ?*option value ...*?

This command is similar to the **configure** widget command except that it modifies options associated with the tag given by *tagName* instead of modifying options for the overall text widget. If no *option* is specified, the command returns a list describing all of the available options for *tagName* (see **Tk_ConfigureInfo** for information on the format of this list). If *option* is specified with no *value*, then the command returns a list describing the one named option (this list will be identical to the corresponding sublist of the value returned if no *option* is specified). If one or more *option−value* pairs are specified, then the command modifies the given option(s) to have the given value(s) in *tagName*; in this case the command returns an empty string. See TAGS above for details on the options available for tags.

pathName **tag delete** *tagName* ?*tagName ...*?

Deletes all tag information for each of the *tagName* arguments. The command removes the tags from all characters in the file and also deletes any other information associated with the tags, such as bindings and display information. The command returns an empty string.

pathName **tag lower** *tagName* ?*belowThis*?
Changes the priority of tag *tagName* so that it is just lower in priority than the tag whose name is *belowThis*. If *belowThis* is omitted, then *tagName*'s priority is changed to make it lowest priority of all tags.

pathName **tag names** ?*index*?
Returns a list whose elements are the names of all the tags that are active at the character position given by *index*. If *index* is omitted, then the return value will describe all of the tags that exist for the text (this includes all tags that have been named in a ''*pathName* **tag**'' widget command but haven't been deleted by a ''*pathName* **tag delete**'' widget command, even if no characters are currently marked with the tag). The list will be sorted in order from lowest priority to highest priority.

pathName **tag nextrange** *tagName index1* ?*index2*?
This command searches the text for a range of characters tagged with *tagName* where the first character of the range is no earlier than the character at *index1* and no later than the character just before *index2* (a range starting at *index2* will not be considered). If several matching ranges exist, the first one is chosen. The command's return value is a list containing two elements, which are the index of the first character of the range and the index of the character just after the last one in the range. If no matching range is found then the return value is an empty string. If *index2* is not given then it defaults to the end of the text.

pathName **tag prevrange** *tagName index1* ?*index2*?
This command searches the text for a range of characters tagged with *tagName* where the first character of the range is before the character at *index1* and no earlier than the character at *index2* (a range starting at *index2* will be considered). If several matching ranges exist, the one closest to *index1* is chosen. The command's return value is a list containing two elements, which are the index of the first character of the range and the index of the character just after the last one in the range. If no matching range is found then the return value is an empty string. If *index2* is not given then it defaults to the beginning of the text.

pathName **tag raise** *tagName* ?*aboveThis*?
Changes the priority of tag *tagName* so that it is just higher in priority than the tag whose name is *aboveThis*. If *aboveThis* is omitted, then *tagName*'s priority is changed to make it highest priority of all tags.

pathName **tag ranges** *tagName*
Returns a list describing all of the ranges of text that have been tagged with *tagName*. The first two elements of the list describe the first tagged range in the text, the next two elements describe the second range, and so on. The first element of each pair contains the index of the first character of the range, and the second element of the pair contains the index of the character just after the last one in the range. If there are no characters tagged with *tag* then an empty string is returned.

pathName **tag remove** *tagName index1* ?*index2 index1 index2 ...*?
Remove the tag *tagName* from all of the characters starting at *index1* and ending just before *index2* (the character at *index2* isn't affected). A single command may contain any number of *index1*−*index2* pairs. If the last *index2* is omitted then the single character at *index1* is tagged. If there are no characters in the specified range (e.g. *index1* is past the end of the file or *index2* is less than or equal to *index1*) then the command has no effect. This command returns an empty string.

pathName **window** *option* ?*arg arg ...*?
This command is used to manipulate embedded windows. The behavior of the command depends

on the *option* argument that follows the **tag** argument. The following forms of the command are currently supported:

pathName **window cget** *index option*
Returns the value of a configuration option for an embedded window. *Index* identifies the embedded window, and *option* specifies a particular configuration option, which must be one of the ones listed in the section EMBEDDED WINDOWS.

pathName **window configure** *index* ?*option value ...*?
Query or modify the configuration options for an embedded window. If no *option* is specified, returns a list describing all of the available options for the embedded window at *index* (see **Tk_ConfigureInfo** for information on the format of this list). If *option* is specified with no *value*, then the command returns a list describing the one named option (this list will be identical to the corresponding sublist of the value returned if no *option* is specified). If one or more *option−value* pairs are specified, then the command modifies the given option(s) to have the given value(s); in this case the command returns an empty string. See EMBEDDED WINDOWS for information on the options that are supported.

pathName **window create** *index* ?*option value ...*?
This command creates a new window annotation, which will appear in the text at the position given by *index*. Any number of *option−value* pairs may be specified to configure the annotation. See EMBEDDED WINDOWS for information on the options that are supported. Returns an empty string.

pathName **window names**
Returns a list whose elements are the names of all windows currently embedded in *window*.

pathName **xview** *option args*
This command is used to query and change the horizontal position of the text in the widget's window. It can take any of the following forms:

pathName **xview**
Returns a list containing two elements. Each element is a real fraction between 0 and 1; together they describe the portion of the document's horizontal span that is visible in the window. For example, if the first element is .2 and the second element is .6, 20% of the text is off-screen to the left, the middle 40% is visible in the window, and 40% of the text is off-screen to the right. The fractions refer only to the lines that are actually visible in the window: if the lines in the window are all very short, so that they are entirely visible, the returned fractions will be 0 and 1, even if there are other lines in the text that are much wider than the window. These are the same values passed to scrollbars via the **−xscrollcommand** option.

pathName **xview moveto** *fraction*
Adjusts the view in the window so that *fraction* of the horizontal span of the text is off-screen to the left. *Fraction* is a fraction between 0 and 1.

pathName **xview scroll** *number what*
This command shifts the view in the window left or right according to *number* and *what*. *Number* must be an integer. *What* must be either **units** or **pages** or an abbreviation of one of these. If *what* is **units**, the view adjusts left or right by *number* average-width characters on the display; if it is **pages** then the view adjusts by *number* screenfuls. If *number* is negative then characters farther to the left become visible; if it is positive then characters farther to the right become visible.

pathName **yview** *?args*?
This command is used to query and change the vertical position of the text in the widget's window.

It can take any of the following forms:

pathName **yview**
Returns a list containing two elements, both of which are real fractions between 0 and 1. The first element gives the position of the first character in the top line in the window, relative to the text as a whole (0.5 means it is halfway through the text, for example). The second element gives the position of the character just after the last one in the bottom line of the window, relative to the text as a whole. These are the same values passed to scrollbars via the **−yscrollcommand** option.

pathName **yview moveto** *fraction*
Adjusts the view in the window so that the character given by *fraction* appears on the top line of the window. *Fraction* is a fraction between 0 and 1; 0 indicates the first character in the text, 0.33 indicates the character one-third the way through the text, and so on.

pathName **yview scroll** *number what*
This command adjust the view in the window up or down according to *number* and *what*. *Number* must be an integer. *What* must be either **units** or **pages**. If *what* is **units**, the view adjusts up or down by *number* lines on the display; if it is **pages** then the view adjusts by *number* screenfuls. If *number* is negative then earlier positions in the text become visible; if it is positive then later positions in the text become visible.

pathName **yview** ?**−pickplace**? *index*
Changes the view in the widget's window to make *index* visible. If the **−pickplace** option isn't specified then *index* will appear at the top of the window. If **−pickplace** is specified then the widget chooses where *index* appears in the window:

[1] If *index* is already visible somewhere in the window then the command does nothing.

[2] If *index* is only a few lines off-screen above the window then it will be positioned at the top of the window.

[3] If *index* is only a few lines off-screen below the window then it will be positioned at the bottom of the window.

[4] Otherwise, *index* will be centered in the window.

The **−pickplace** option has been obsoleted by the **see** widget command (**see** handles both x- and y-motion to make a location visible, whereas **−pickplace** only handles motion in y).

pathName **yview** *number*
This command makes the first character on the line after the one given by *number* visible at the top of the window. *Number* must be an integer. This command used to be used for scrolling, but now it is obsolete.

BINDINGS

Tk automatically creates class bindings for texts that give them the following default behavior. In the descriptions below, ''word'' refers to a contiguous group of letters, digits, or ''_'' characters, or any single character other than these.

[1] Clicking mouse button 1 positions the insertion cursor just before the character underneath the mouse cursor, sets the input focus to this widget, and clears any selection in the widget. Dragging with mouse button 1 strokes out a selection between the insertion cursor and the character under the mouse.

[2] Double-clicking with mouse button 1 selects the word under the mouse and positions the insertion

cursor at the beginning of the word. Dragging after a double click will stroke out a selection consisting of whole words.

[3] Triple-clicking with mouse button 1 selects the line under the mouse and positions the insertion cursor at the beginning of the line. Dragging after a triple click will stroke out a selection consisting of whole lines.

[4] The ends of the selection can be adjusted by dragging with mouse button 1 while the Shift key is down; this will adjust the end of the selection that was nearest to the mouse cursor when button 1 was pressed. If the button is double-clicked before dragging then the selection will be adjusted in units of whole words; if it is triple-clicked then the selection will be adjusted in units of whole lines.

[5] Clicking mouse button 1 with the Control key down will reposition the insertion cursor without affecting the selection.

[6] If any normal printing characters are typed, they are inserted at the point of the insertion cursor.

[7] The view in the widget can be adjusted by dragging with mouse button 2. If mouse button 2 is clicked without moving the mouse, the selection is copied into the text at the position of the mouse cursor. The Insert key also inserts the selection, but at the position of the insertion cursor.

[8] If the mouse is dragged out of the widget while button 1 is pressed, the entry will automatically scroll to make more text visible (if there is more text off-screen on the side where the mouse left the window).

[9] The Left and Right keys move the insertion cursor one character to the left or right; they also clear any selection in the text. If Left or Right is typed with the Shift key down, then the insertion cursor moves and the selection is extended to include the new character. Control-Left and Control-Right move the insertion cursor by words, and Control-Shift-Left and Control-Shift-Right move the insertion cursor by words and also extend the selection. Control-b and Control-f behave the same as Left and Right, respectively. Meta-b and Meta-f behave the same as Control-Left and Control-Right, respectively.

[10] The Up and Down keys move the insertion cursor one line up or down and clear any selection in the text. If Up or Right is typed with the Shift key down, then the insertion cursor moves and the selection is extended to include the new character. Control-Up and Control-Down move the insertion cursor by paragraphs (groups of lines separated by blank lines), and Control-Shift-Up and Control-Shift-Down move the insertion cursor by paragraphs and also extend the selection. Control-p and Control-n behave the same as Up and Down, respectively.

[11] The Next and Prior keys move the insertion cursor forward or backwards by one screenful and clear any selection in the text. If the Shift key is held down while Next or Prior is typed, then the selection is extended to include the new character. Control-v moves the view down one screenful without moving the insertion cursor or adjusting the selection.

[12] Control-Next and Control-Prior scroll the view right or left by one page without moving the insertion cursor or affecting the selection.

[13] Home and Control-a move the insertion cursor to the beginning of its line and clear any selection in the widget. Shift-Home moves the insertion cursor to the beginning of the line and also extends the selection to that point.

[14] End and Control-e move the insertion cursor to the end of the line and clear any selection in the widget. Shift-End moves the cursor to the end of the line and extends the selection to that point.

[15] Control-Home and Meta-< move the insertion cursor to the beginning of the text and clear any selection in the widget. Control-Shift-Home moves the insertion cursor to the beginning of the text and also extends the selection to that point.

[16] Control-End and Meta-> move the insertion cursor to the end of the text and clear any selection in the widget. Control-Shift-End moves the cursor to the end of the text and extends the selection to that point.

[17] The Select key and Control-Space set the selection anchor to the position of the insertion cursor. They don't affect the current selection. Shift-Select and Control-Shift-Space adjust the selection to the current position of the insertion cursor, selecting from the anchor to the insertion cursor if there was not any selection previously.

[18] Control-/ selects the entire contents of the widget.

[19] Control-\ clears any selection in the widget.

[20] The F16 key (labelled Copy on many Sun workstations) or Meta-w copies the selection in the widget to the clipboard, if there is a selection.

[21] The F20 key (labelled Cut on many Sun workstations) or Control-w copies the selection in the widget to the clipboard and deletes the selection. If there is no selection in the widget then these keys have no effect.

[22] The F18 key (labelled Paste on many Sun workstations) or Control-y inserts the contents of the clipboard at the position of the insertion cursor.

[23] The Delete key deletes the selection, if there is one in the widget. If there is no selection, it deletes the character to the right of the insertion cursor.

[24] Backspace and Control-h delete the selection, if there is one in the widget. If there is no selection, they delete the character to the left of the insertion cursor.

[25] Control-d deletes the character to the right of the insertion cursor.

[26] Meta-d deletes the word to the right of the insertion cursor.

[27] Control-k deletes from the insertion cursor to the end of its line; if the insertion cursor is already at the end of a line, then Control-k deletes the newline character.

[28] Control-o opens a new line by inserting a newline character in front of the insertion cursor without moving the insertion cursor.

[29] Meta-backspace and Meta-Delete delete the word to the left of the insertion cursor.

[30] Control-x deletes whatever is selected in the text widget.

[31] Control-t reverses the order of the two characters to the right of the insertion cursor.

If the widget is disabled using the **−state** option, then its view can still be adjusted and text can still be selected, but no insertion cursor will be displayed and no text modifications will take place.

The behavior of texts can be changed by defining new bindings for individual widgets or by redefining the class bindings.

PERFORMANCE ISSUES

Text widgets should run efficiently under a variety of conditions. The text widget uses about 2-3 bytes of main memory for each byte of text, so texts containing a megabyte or more should be practical on most workstations. Text is represented internally with a modified B-tree structure that makes operations relatively efficient even with large texts. Tags are included in the B-tree structure in a way that allows tags to span large ranges or have many disjoint smaller ranges without loss of efficiency. Marks are also implemented in a way that allows large numbers of marks. In most cases it is fine to have large numbers of unique tags, or a tag that has many distinct ranges.

One performance problem can arise if you have hundreds or thousands of different tags that all have the following characteristics: the first and last ranges of each tag are near the beginning and end of the text, respectively, or a single tag range covers most of the text widget. The cost of adding and deleting tags like this is proportional to the number of other tags with the same properties. In contrast, there is no problem with having thousands of distinct tags if their overall ranges are localized and spread uniformly throughout the text.

Very long text lines can be expensive, especially if they have many marks and tags within them.

The display line with the insert cursor is redrawn each time the cursor blinks, which causes a steady stream of graphics traffic. Set the **insertOffTime** attribute to 0 avoid this.

KEYWORDS

text, widget

NAME

tk − Manipulate Tk internal state

SYNOPSIS

tk *option* ?*arg arg ...*?

DESCRIPTION

The **tk** command provides access to miscellaneous elements of Tk's internal state. Most of the information manipulated by this command pertains to the application as a whole, or to a screen or display, rather than to a particular window. The command can take any of a number of different forms depending on the *option* argument. The legal forms are:

tk appname ?*newName*?

If *newName* isn't specified, this command returns the name of the application (the name that may be used in **send** commands to communicate with the application). If *newName* is specified, then the name of the application is changed to *newName*. If the given name is already in use, then a suffix of the form " **#2**" or " **#3**" is appended in order to make the name unique. The command's result is the name actually chosen. *newName* should not start with a capital letter. This will interfere with option processing, since names starting with capitals are assumed to be classes; as a result, Tk may not be able to find some options for the application. If sends have been disabled by deleting the **send** command, this command will reenable them and recreate the **send** command.

tk scaling ?**−displayof** *window*? ?*number*?

Sets and queries the current scaling factor used by Tk to convert between physical units (for example, points, inches, or millimeters) and pixels. The *number* argument is a floating point number that specifies the number of pixels per point on *window*'s display. If the *window* argument is omitted, it defaults to the main window. If the *number* argument is omitted, the current value of the scaling factor is returned.

A "point" is a unit of measurement equal to 1/72 inch. A scaling factor of 1.0 corresponds to 1 pixel per point, which is equivalent to a standard 72 dpi monitor. A scaling factor of 1.25 would mean 1.25 pixels per point, which is the setting for a 90 dpi monitor; setting the scaling factor to 1.25 on a 72 dpi monitor would cause everything in the application to be displayed 1.25 times as large as normal. The initial value for the scaling factor is set when the application starts, based on properties of the installed monitor, but it can be changed at any time. Measurements made after the scaling factor is changed will use the new scaling factor, but it is undefined whether existing widgets will resize themselves dynamically to accomodate the new scaling factor.

KEYWORDS

application name, send

NAME

tkerror − Command invoked to process background errors

SYNOPSIS

tkerror *message*

DESCRIPTION

Note: as of Tk 4.1 the **tkerror** command has been renamed to **bgerror** because the event loop (which is what usually invokes it) is now part of Tcl. For backward compatibility the **bgerror** provided by the current Tk version still tries to call **tkerror** if there is one (or an auto loadable one), so old script defining that error handler should still work, but you should anyhow modify your scripts to use **bgerror** instead of **tkerror** because that support for the old name might vanish in the near future. If that call fails, **bgerror** posts a dialog showing the error and offering to see the stack trace to the user. If you want your own error management you should directly override **bgerror** instead of **tkerror**. Documentation for **bgerror** is available as part of Tcl's documentation.

KEYWORDS

background error, reporting

NAME

tkvars − Variables used or set by Tk

DESCRIPTION

The following Tcl variables are either set or used by Tk at various times in its execution:

tk_library This variable holds the file name for a directory containing a library of Tcl scripts related to Tk. These scripts include an initialization file that is normally processed whenever a Tk application starts up, plus other files containing procedures that implement default behaviors for widgets. The initial value of **tcl_library** is set when Tk is added to an interpreter; this is done by searching several different directories until one is found that contains an appropriate Tk startup script. If the **TK_LIBRARY** environment variable exists, then the directory it names is checked first. If **TK_LIBRARY** isn't set or doesn't refer to an appropriate directory, then Tk checks several other directories based on a compiled-in default location, the location of the Tcl library directory, the location of the binary containing the application, and the current working directory. The variable can be modified by an application to switch to a different library.

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

tkPriv This variable is an array containing several pieces of information that are private to Tk. The elements of **tkPriv** are used by Tk library procedures and default bindings. They should not be accessed by any code outside Tk.

tk_strictMotif This variable is set to zero by default. If an application sets it to one, then Tk attempts to adhere as closely as possible to Motif look-and-feel standards. For example, active elements such as buttons and scrollbar sliders will not change color when the pointer passes over them.

tk_version Tk sets this variable in the interpreter for each application. The variable holds the current version number of the Tk library in the form *major*.*minor*. *Major* and *minor* are integers. The major version number increases in any Tk release that includes changes that are not backward compatible (i.e. whenever existing Tk applications and scripts may have to change to work with the new release). The minor version number increases with each new release of Tk, except that it resets to zero whenever the major version number changes.

KEYWORDS

variables, version

NAME

tkwait − Wait for variable to change or window to be destroyed

SYNOPSIS

tkwait variable *name*

tkwait visibility *name*

tkwait window *name*

DESCRIPTION

The **tkwait** command waits for one of several things to happen, then it returns without taking any other actions. The return value is always an empty string. If the first argument is **variable** (or any abbreviation of it) then the second argument is the name of a global variable and the command waits for that variable to be modified. If the first argument is **visibility** (or any abbreviation of it) then the second argument is the name of a window and the **tkwait** command waits for a change in its visibility state (as indicated by the arrival of a VisibilityNotify event). This form is typically used to wait for a newly-created window to appear on the screen before taking some action. If the first argument is **window** (or any abbreviation of it) then the second argument is the name of a window and the **tkwait** command waits for that window to be destroyed. This form is typically used to wait for a user to finish interacting with a dialog box before using the result of that interaction.

While the **tkwait** command is waiting it processes events in the normal fashion, so the application will continue to respond to user interactions. If an event handler invokes **tkwait** again, the nested call to **tkwait** must complete before the outer call can complete.

KEYWORDS

variable, visibility, wait, window

NAME

toplevel − Create and manipulate toplevel widgets

SYNOPSIS

toplevel *pathName* ?*options*?

STANDARD OPTIONS

−borderwidth	**−highlightbackground**	**−highlightthickness**	**−takefocus**
−cursor	**−highlightcolor**	**−relief**	

See the **options** manual entry for details on the standard options.

WIDGET-SPECIFIC OPTIONS

Command-Line Name: **−background**
Database Name: **background**
Database Class: **Background**

This option is the same as the standard **background** option except that its value may also be specified as an empty string. In this case, the widget will display no background or border, and no colors will be consumed from its colormap for its background and border.

Command-Line Name: **−class**
Database Name: **class**
Database Class: **Class**

Specifies a class for the window. This class will be used when querying the option database for the window's other options, and it will also be used later for other purposes such as bindings. The **class** option may not be changed with the **configure** widget command.

Command-Line Name: **−colormap**
Database Name: **colormap**
Database Class: **Colormap**

Specifies a colormap to use for the window. The value may be either **new**, in which case a new colormap is created for the window and its children, or the name of another window (which must be on the same screen and have the same visual as *pathName*), in which case the new window will use the colormap from the specified window. If the **colormap** option is not specified, the new window uses the default colormap of its screen. This option may not be changed with the **configure** widget command.

Command-Line Name: **−container**
Database Name: **container**
Database Class: **Container**

The value must be a boolean. If true, it means that this window will be used as a container in which some other application will be embedded (for example, a Tk toplevel can be embedded using the **−use** option). The window will support the appropriate window manager protocols for things like geometry requests. The window should not have any children of its own in this application. This option may not be changed with the **configure** widget command.

Command-Line Name: **−height**
Database Name: **height**
Database Class: **Height**

Specifies the desired height for the window in any of the forms acceptable to **Tk_GetPixels**. If this option is less than or equal to zero then the window will not request any size at all.

Command-Line Name: **−menu**
Database Name: **menu**
Database Class: **Menu**

Specifies a menu widget to be used as a menubar. On the Macintosh, the menubar will be displayed accross the top of the main monitor. On Microsoft Windows and all UNIX platforms, the menu will appear accross the toplevel window as part of the window dressing maintained by the window manager.

Command-Line Name: **−screen**
Database Name:
Database Class:

Specifies the screen on which to place the new window. Any valid screen name may be used, even one associated with a different display. Defaults to the same screen as its parent. This option is special in that it may not be specified via the option database, and it may not be modified with the **configure** widget command.

Command-Line Name: **−use**
Database Name: **use**
Database Class: **Use**

This option is used for embedding. If the value isn't an empty string, it must be the the window identifier of a container window, specified as a hexadecimal string like the ones returned by the **winfo id** command. The toplevel widget will be created as a child of the given container instead of the root window for the screen. If the container window is in a Tk application, it must be a frame or toplevel widget for which the **−container** option was specified. This option may not be changed with the **configure** widget command.

Command-Line Name: **−visual**
Database Name: **visual**
Database Class: **Visual**

Specifies visual information for the new window in any of the forms accepted by **Tk_GetVisual**. If this option is not specified, the new window will use the default visual for its screen. The **visual** option may not be modified with the **configure** widget command.

Command-Line Name: **−width**
Database Name: **width**
Database Class: **Width**

Specifies the desired width for the window in any of the forms acceptable to **Tk_GetPixels**. If this option is less than or equal to zero then the window will not request any size at all.

DESCRIPTION

The **toplevel** command creates a new toplevel widget (given by the *pathName* argument). Additional options, described above, may be specified on the command line or in the option database to configure aspects of the toplevel such as its background color and relief. The **toplevel** command returns the path name of the new window.

A toplevel is similar to a frame except that it is created as a top-level window: its X parent is the root window of a screen rather than the logical parent from its path name. The primary purpose of a toplevel is to serve as a container for dialog boxes and other collections of widgets. The only visible features of a toplevel are its background color and an optional 3-D border to make the toplevel appear raised or sunken.

WIDGET COMMAND

The **toplevel** command creates a new Tcl command whose name is the same as the path name of the toplevel's window. This command may be used to invoke various operations on the widget. It has the following general form:

pathName option ?*arg arg ...*?

PathName is the name of the command, which is the same as the toplevel widget's path name. *Option* and the *arg*s determine the exact behavior of the command. The following commands are possible for toplevel widgets:

pathName **cget** *option*

Returns the current value of the configuration option given by *option*. *Option* may have any of the values accepted by the **toplevel** command.

pathName **configure** ?*option*? ?*value option value ...*?

Query or modify the configuration options of the widget. If no *option* is specified, returns a list describing all of the available options for *pathName* (see **Tk_ConfigureInfo** for information on the format of this list). If *option* is specified with no *value*, then the command returns a list describing the one named option (this list will be identical to the corresponding sublist of the value returned if no *option* is specified). If one or more *option−value* pairs are specified, then the command modifies the given widget option(s) to have the given value(s); in this case the command returns an empty string. *Option* may have any of the values accepted by the **toplevel** command.

BINDINGS

When a new toplevel is created, it has no default event bindings: toplevels are not intended to be interactive.

KEYWORDS

toplevel, widget

NAME

winfo − Return window-related information

SYNOPSIS

winfo *option* ?*arg arg ...*?

DESCRIPTION

The **winfo** command is used to retrieve information about windows managed by Tk. It can take any of a number of different forms, depending on the *option* argument. The legal forms are:

winfo atom ?−**displayof** *window*? *name*

Returns a decimal string giving the integer identifier for the atom whose name is *name*. If no atom exists with the name *name* then a new one is created. If the −**displayof** option is given then the atom is looked up on the display of *window*; otherwise it is looked up on the display of the application's main window.

winfo atomname ?−**displayof** *window*? *id*

Returns the textual name for the atom whose integer identifier is *id*. If the −**displayof** option is given then the identifier is looked up on the display of *window*; otherwise it is looked up on the display of the application's main window. This command is the inverse of the **winfo atom** command. It generates an error if no such atom exists.

winfo cells *window*

Returns a decimal string giving the number of cells in the color map for *window*.

winfo children *window*

Returns a list containing the path names of all the children of *window*. The list is in stacking order, with the lowest window first. Top-level windows are returned as children of their logical parents.

winfo class *window*

Returns the class name for *window*.

winfo colormapfull *window*

Returns 1 if the colormap for *window* is known to be full, 0 otherwise. The colormap for a window is ''known'' to be full if the last attempt to allocate a new color on that window failed and this application hasn't freed any colors in the colormap since the failed allocation.

winfo containing ?−**displayof** *window*? *rootX rootY*

Returns the path name for the window containing the point given by *rootX* and *rootY*. *RootX* and *rootY* are specified in screen units (i.e. any form acceptable to **Tk_GetPixels**) in the coordinate system of the root window (if a virtual-root window manager is in use then the coordinate system of the virtual root window is used). If the −**displayof** option is given then the coordinates refer to the screen containing *window*; otherwise they refer to the screen of the application's main window. If no window in this application contains the point then an empty string is returned. In selecting the containing window, children are given higher priority than parents and among siblings the highest one in the stacking order is chosen.

winfo depth *window*

Returns a decimal string giving the depth of *window* (number of bits per pixel).

winfo exists *window*

Returns 1 if there exists a window named *window*, 0 if no such window exists.

winfo fpixels *window number*

Returns a floating-point value giving the number of pixels in *window* corresponding to the distance given by *number*. *Number* may be specified in any of the forms acceptable to **Tk_GetScreenMM**,

such as ‘‘2.0c’’ or ‘‘1i’’. The return value may be fractional; for an integer value, use **winfo pixels**.

winfo geometry *window*
Returns the geometry for *window*, in the form *width***x***height*+*x*+*y*. All dimensions are in pixels.

winfo height *window*
Returns a decimal string giving *window*’s height in pixels. When a window is first created its height will be 1 pixel; the height will eventually be changed by a geometry manager to fulfill the window’s needs. If you need the true height immediately after creating a widget, invoke **update** to force the geometry manager to arrange it, or use **winfo reqheight** to get the window’s requested height instead of its actual height.

winfo id *window*
Returns a hexadecimal string giving a low-level platform-specific identifier for *window*. On Unix platforms, this is the X window identifier. Under Windows, this is the Windows HWND. On the Macintosh the value has no meaning outside Tk.

winfo interps ?**−displayof** *window*?
Returns a list whose members are the names of all Tcl interpreters (e.g. all Tk-based applications) currently registered for a particular display. If the **−displayof** option is given then the return value refers to the display of *window*; otherwise it refers to the display of the application’s main window.

winfo ismapped *window*
Returns **1** if *window* is currently mapped, **0** otherwise.

winfo manager *window*
Returns the name of the geometry manager currently responsible for *window*, or an empty string if *window* isn’t managed by any geometry manager. The name is usually the name of the Tcl command for the geometry manager, such as **pack** or **place**. If the geometry manager is a widget, such as canvases or text, the name is the widget’s class command, such as **canvas**.

winfo name *window*
Returns *window*’s name (i.e. its name within its parent, as opposed to its full path name). The command **winfo name .** will return the name of the application.

winfo parent *window*
Returns the path name of *window*’s parent, or an empty string if *window* is the main window of the application.

winfo pathname ?**−displayof** *window*? *id*
Returns the path name of the window whose X identifier is *id*. *Id* must be a decimal, hexadecimal, or octal integer and must correspond to a window in the invoking application. If the **−displayof** option is given then the identifier is looked up on the display of *window*; otherwise it is looked up on the display of the application’s main window.

winfo pixels *window number*
Returns the number of pixels in *window* corresponding to the distance given by *number*. *Number* may be specified in any of the forms acceptable to **Tk_GetPixels**, such as ‘‘2.0c’’ or ‘‘1i’’. The result is rounded to the nearest integer value; for a fractional result, use **winfo fpixels**.

winfo pointerx *window*
If the mouse pointer is on the same screen as *window*, returns the pointer’s x coordinate, measured in pixels in the screen’s root window. If a virtual root window is in use on the screen, the position is measured in the virtual root. If the mouse pointer isn’t on the same screen as *window* then -1 is returned.

winfo pointerxy *window*

If the mouse pointer is on the same screen as *window*, returns a list with two elements, which are the pointer's x and y coordinates measured in pixels in the screen's root window. If a virtual root window is in use on the screen, the position is computed in the virtual root. If the mouse pointer isn't on the same screen as *window* then both of the returned coordinates are -1.

winfo pointery *window*
If the mouse pointer is on the same screen as *window*, returns the pointer's y coordinate, measured in pixels in the screen's root window. If a virtual root window is in use on the screen, the position is computed in the virtual root. If the mouse pointer isn't on the same screen as *window* then -1 is returned.

winfo reqheight *window*
Returns a decimal string giving *window*'s requested height, in pixels. This is the value used by *window*'s geometry manager to compute its geometry.

winfo reqwidth *window*
Returns a decimal string giving *window*'s requested width, in pixels. This is the value used by *window*'s geometry manager to compute its geometry.

winfo rgb *window color*
Returns a list containing three decimal values, which are the red, green, and blue intensities that correspond to *color* in the window given by *window*. *Color* may be specified in any of the forms acceptable for a color option.

winfo rootx *window*
Returns a decimal string giving the x-coordinate, in the root window of the screen, of the upper-left corner of *window*'s border (or *window* if it has no border).

winfo rooty *window*
Returns a decimal string giving the y-coordinate, in the root window of the screen, of the upper-left corner of *window*'s border (or *window* if it has no border).

winfo screen *window*
Returns the name of the screen associated with *window*, in the form *displayName.screenIndex*.

winfo screencells *window*
Returns a decimal string giving the number of cells in the default color map for *window*'s screen.

winfo screendepth *window*
Returns a decimal string giving the depth of the root window of *window*'s screen (number of bits per pixel).

winfo screenheight *window*
Returns a decimal string giving the height of *window*'s screen, in pixels.

winfo screenmmheight *window*
Returns a decimal string giving the height of *window*'s screen, in millimeters.

winfo screenmmwidth *window*
Returns a decimal string giving the width of *window*'s screen, in millimeters.

winfo screenvisual *window*
Returns one of the following strings to indicate the default visual class for *window*'s screen: **directcolor**, **grayscale**, **pseudocolor**, **staticcolor**, **staticgray**, or **truecolor**.

winfo screenwidth *window*
Returns a decimal string giving the width of *window*'s screen, in pixels.

winfo server *window*
Returns a string containing information about the server for *window*'s display. The exact format of this string may vary from platform to platform. For X servers the string has the form

"**X***major***R***minor vendor vendorVersion*" where *major* and *minor* are the version and revision numbers provided by the server (e.g., **X11R5**), *vendor* is the name of the vendor for the server, and *vendorRelease* is an integer release number provided by the server.

winfo toplevel *window*
Returns the path name of the top-level window containing *window*.

winfo viewable *window*
Returns 1 if *window* and all of its ancestors up through the nearest toplevel window are mapped. Returns 0 if any of these windows are not mapped.

winfo visual *window*
Returns one of the following strings to indicate the visual class for *window*: **directcolor**, **grayscale**, **pseudocolor**, **staticcolor**, **staticgray**, or **truecolor**.

winfo visualid *window*
Returns the X identifier for the visual for *window*.

winfo visualsavailable *window* ?**includeids**?
Returns a list whose elements describe the visuals available for *window*'s screen. Each element consists of a visual class followed by an integer depth. The class has the same form as returned by **winfo visual**. The depth gives the number of bits per pixel in the visual. In addition, if the **includeids** argument is provided, then the depth is followed by the X identifier for the visual.

winfo vrootheight *window*
Returns the height of the virtual root window associated with *window* if there is one; otherwise returns the height of *window*'s screen.

winfo vrootwidth *window*
Returns the width of the virtual root window associated with *window* if there is one; otherwise returns the width of *window*'s screen.

winfo vrootx *window*
Returns the x-offset of the virtual root window associated with *window*, relative to the root window of its screen. This is normally either zero or negative. Returns 0 if there is no virtual root window for *window*.

winfo vrooty *window*
Returns the y-offset of the virtual root window associated with *window*, relative to the root window of its screen. This is normally either zero or negative. Returns 0 if there is no virtual root window for *window*.

winfo width *window*
Returns a decimal string giving *window*'s width in pixels. When a window is first created its width will be 1 pixel; the width will eventually be changed by a geometry manager to fulfill the window's needs. If you need the true width immediately after creating a widget, invoke **update** to force the geometry manager to arrange it, or use **winfo reqwidth** to get the window's requested width instead of its actual width.

winfo x *window*
Returns a decimal string giving the x-coordinate, in *window*'s parent, of the upper-left corner of *window*'s border (or *window* if it has no border).

winfo y *window*
Returns a decimal string giving the y-coordinate, in *window*'s parent, of the upper-left corner of *window*'s border (or *window* if it has no border).

KEYWORDS

atom, children, class, geometry, height, identifier, information, interpreters, mapped, parent, path name, screen, virtual root, width, window

NAME

wm − Communicate with window manager

SYNOPSIS

wm *option window* ?*args*?

DESCRIPTION

The **wm** command is used to interact with window managers in order to control such things as the title for a window, its geometry, or the increments in terms of which it may be resized. The **wm** command can take any of a number of different forms, depending on the *option* argument. All of the forms expect at least one additional argument, *window*, which must be the path name of a top-level window.

The legal forms for the **wm** command are:

wm aspect *window* ?*minNumer minDenom maxNumer maxDenom*?

If *minNumer*, *minDenom*, *maxNumer*, and *maxDenom* are all specified, then they will be passed to the window manager and the window manager should use them to enforce a range of acceptable aspect ratios for *window*. The aspect ratio of *window* (width/length) will be constrained to lie between *minNumer*/*minDenom* and *maxNumer*/*maxDenom*. If *minNumer* etc. are all specified as empty strings, then any existing aspect ratio restrictions are removed. If *minNumer* etc. are specified, then the command returns an empty string. Otherwise, it returns a Tcl list containing four elements, which are the current values of *minNumer*, *minDenom*, *maxNumer*, and *maxDenom* (if no aspect restrictions are in effect, then an empty string is returned).

wm client *window* ?*name*?

If *name* is specified, this command stores *name* (which should be the name of the host on which the application is executing) in *window*'s **WM_CLIENT_MACHINE** property for use by the window manager or session manager. The command returns an empty string in this case. If *name* isn't specified, the command returns the last name set in a **wm client** command for *window*. If *name* is specified as an empty string, the command deletes the **WM_CLIENT_MACHINE** property from *window*.

wm colormapwindows *window* ?*windowList*?

This command is used to manipulate the **WM_COLORMAP_WINDOWS** property, which provides information to the window managers about windows that have private colormaps. If *windowList* isn't specified, the command returns a list whose elements are the names of the windows in the **WM_COLORMAP_WINDOWS** property. If *windowList* is specified, it consists of a list of window path names; the command overwrites the **WM_COLORMAP_WINDOWS** property with the given windows and returns an empty string. The **WM_COLORMAP_WINDOWS** property should normally contain a list of the internal windows within *window* whose colormaps differ from their parents. The order of the windows in the property indicates a priority order: the window manager will attempt to install as many colormaps as possible from the head of this list when *window* gets the colormap focus. If *window* is not included among the windows in *windowList*, Tk implicitly adds it at the end of the **WM_COLORMAP_WINDOWS** property, so that its colormap is lowest in priority. If **wm colormapwindows** is not invoked, Tk will automatically set the property for each top-level window to all the internal windows whose colormaps differ from their parents, followed by the top-level itself; the order of the internal windows is undefined. See the ICCCM documentation for more information on the **WM_COLORMAP_WINDOWS** property.

wm command *window* ?*value*?

If *value* is specified, this command stores *value* in *window*'s **WM_COMMAND** property for use by the window manager or session manager and returns an empty string. *Value* must have proper list structure; the elements should contain the words of the command used to invoke the

application. If *value* isn't specified then the command returns the last value set in a **wm command** command for *window*. If *value* is specified as an empty string, the command deletes the **WM_COMMAND** property from *window*.

wm deiconify *window*

Arrange for *window* to be displayed in normal (non-iconified) form. This is done by mapping the window. If the window has never been mapped then this command will not map the window, but it will ensure that when the window is first mapped it will be displayed in de-iconified form. Returns an empty string.

wm focusmodel *window* ?**active**|**passive**?

If **active** or **passive** is supplied as an optional argument to the command, then it specifies the focus model for *window*. In this case the command returns an empty string. If no additional argument is supplied, then the command returns the current focus model for *window*. An **active** focus model means that *window* will claim the input focus for itself or its descendants, even at times when the focus is currently in some other application. **Passive** means that *window* will never claim the focus for itself: the window manager should give the focus to *window* at appropriate times. However, once the focus has been given to *window* or one of its descendants, the application may reassign the focus among *window*'s descendants. The focus model defaults to **passive**, and Tk's **focus** command assumes a passive model of focusing.

wm frame *window*

If *window* has been reparented by the window manager into a decorative frame, the command returns the platform specific window identifier for the outermost frame that contains *window* (the window whose parent is the root or virtual root). If *window* hasn't been reparented by the window manager then the command returns the platform specific window identifier for *window*.

wm geometry *window* ?*newGeometry*?

If *newGeometry* is specified, then the geometry of *window* is changed and an empty string is returned. Otherwise the current geometry for *window* is returned (this is the most recent geometry specified either by manual resizing or in a **wm geometry** command). *NewGeometry* has the form =*width***x***height*±*x*±*y*, where any of =, *width***x***height*, or ±*x*±*y* may be omitted. *Width* and *height* are positive integers specifying the desired dimensions of *window*. If *window* is gridded (see GRIDDED GEOMETRY MANAGEMENT below) then the dimensions are specified in grid units; otherwise they are specified in pixel units. *X* and *y* specify the desired location of *window* on the screen, in pixels. If *x* is preceded by +, it specifies the number of pixels between the left edge of the screen and the left edge of *window*'s border; if preceded by − then *x* specifies the number of pixels between the right edge of the screen and the right edge of *window*'s border. If *y* is preceded by + then it specifies the number of pixels between the top of the screen and the top of *window*'s border; if *y* is preceded by − then it specifies the number of pixels between the bottom of *window*'s border and the bottom of the screen. If *newGeometry* is specified as an empty string then any existing user-specified geometry for *window* is cancelled, and the window will revert to the size requested internally by its widgets.

wm grid *window* ?*baseWidth baseHeight widthInc heightInc*?

This command indicates that *window* is to be managed as a gridded window. It also specifies the relationship between grid units and pixel units. *BaseWidth* and *baseHeight* specify the number of grid units corresponding to the pixel dimensions requested internally by *window* using **Tk_GeometryRequest**. *WidthInc* and *heightInc* specify the number of pixels in each horizontal and vertical grid unit. These four values determine a range of acceptable sizes for *window*, corresponding to grid-based widths and heights that are non-negative integers. Tk will pass this information to the window manager; during manual resizing, the window manager will restrict the window's size to one of these acceptable sizes. Furthermore, during manual resizing the window manager will display the window's current size in terms of grid units rather than pixels. If *baseWidth* etc. are all specified as empty strings, then *window* will no longer be managed as a gridded window. If

baseWidth etc. are specified then the return value is an empty string. Otherwise the return value is a Tcl list containing four elements corresponding to the current *baseWidth*, *baseHeight*, *widthInc*, and *heightInc*; if *window* is not currently gridded, then an empty string is returned. Note: this command should not be needed very often, since the **Tk_SetGrid** library procedure and the **setGrid** option provide easier access to the same functionality.

wm group *window* ?*pathName*?

If *pathName* is specified, it gives the path name for the leader of a group of related windows. The window manager may use this information, for example, to unmap all of the windows in a group when the group's leader is iconified. *PathName* may be specified as an empty string to remove *window* from any group association. If *pathName* is specified then the command returns an empty string; otherwise it returns the path name of *window*'s current group leader, or an empty string if *window* isn't part of any group.

wm iconbitmap *window* ?*bitmap*?

If *bitmap* is specified, then it names a bitmap in the standard forms accepted by Tk (see the **Tk_GetBitmap** manual entry for details). This bitmap is passed to the window manager to be displayed in *window*'s icon, and the command returns an empty string. If an empty string is specified for *bitmap*, then any current icon bitmap is cancelled for *window*. If *bitmap* is specified then the command returns an empty string. Otherwise it returns the name of the current icon bitmap associated with *window*, or an empty string if *window* has no icon bitmap.

wm iconify *window*

Arrange for *window* to be iconified. It *window* hasn't yet been mapped for the first time, this command will arrange for it to appear in the iconified state when it is eventually mapped.

wm iconmask *window* ?*bitmap*?

If *bitmap* is specified, then it names a bitmap in the standard forms accepted by Tk (see the **Tk_GetBitmap** manual entry for details). This bitmap is passed to the window manager to be used as a mask in conjunction with the **iconbitmap** option: where the mask has zeroes no icon will be displayed; where it has ones, the bits from the icon bitmap will be displayed. If an empty string is specified for *bitmap* then any current icon mask is cancelled for *window* (this is equivalent to specifying a bitmap of all ones). If *bitmap* is specified then the command returns an empty string. Otherwise it returns the name of the current icon mask associated with *window*, or an empty string if no mask is in effect.

wm iconname *window* ?*newName*?

If *newName* is specified, then it is passed to the window manager; the window manager should display *newName* inside the icon associated with *window*. In this case an empty string is returned as result. If *newName* isn't specified then the command returns the current icon name for *window*, or an empty string if no icon name has been specified (in this case the window manager will normally display the window's title, as specified with the **wm title** command).

wm iconposition *window* ?*x y*?

If *x* and *y* are specified, they are passed to the window manager as a hint about where to position the icon for *window*. In this case an empty string is returned. If *x* and *y* are specified as empty strings then any existing icon position hint is cancelled. If neither *x* nor *y* is specified, then the command returns a Tcl list containing two values, which are the current icon position hints (if no hints are in effect then an empty string is returned).

wm iconwindow *window* ?*pathName*?

If *pathName* is specified, it is the path name for a window to use as icon for *window*: when *window* is iconified then *pathName* will be mapped to serve as icon, and when *window* is de-iconified then *pathName* will be unmapped again. If *pathName* is specified as an empty string then any existing icon window association for *window* will be cancelled. If the *pathName* argument is specified then an empty string is returned. Otherwise the command returns the path name of the

current icon window for *window*, or an empty string if there is no icon window currently specified for *window*. Button press events are disabled for *window* as long as it is an icon window; this is needed in order to allow window managers to ''own'' those events. Note: not all window managers support the notion of an icon window.

wm maxsize *window* ?*width height*?

If *width* and *height* are specified, they give the maximum permissible dimensions for *window*. For gridded windows the dimensions are specified in grid units; otherwise they are specified in pixel units. The window manager will restrict the window's dimensions to be less than or equal to *width* and *height*. If *width* and *height* are specified, then the command returns an empty string. Otherwise it returns a Tcl list with two elements, which are the maximum width and height currently in effect. The maximum size defaults to the size of the screen. If resizing has been disabled with the **wm resizable** command, then this command has no effect. See the sections on geometry management below for more information.

wm minsize *window* ?*width height*?

If *width* and *height* are specified, they give the minimum permissible dimensions for *window*. For gridded windows the dimensions are specified in grid units; otherwise they are specified in pixel units. The window manager will restrict the window's dimensions to be greater than or equal to *width* and *height*. If *width* and *height* are specified, then the command returns an empty string. Otherwise it returns a Tcl list with two elements, which are the minimum width and height currently in effect. The minimum size defaults to one pixel in each dimension. If resizing has been disabled with the **wm resizable** command, then this command has no effect. See the sections on geometry management below for more information.

wm overrideredirect *window* ?*boolean*?

If *boolean* is specified, it must have a proper boolean form and the override-redirect flag for *window* is set to that value. If *boolean* is not specified then **1** or **0** is returned to indicate whether or not the override-redirect flag is currently set for *window*. Setting the override-redirect flag for a window causes it to be ignored by the window manager; among other things, this means that the window will not be reparented from the root window into a decorative frame and the user will not be able to manipulate the window using the normal window manager mechanisms.

wm positionfrom *window* ?*who*?

If *who* is specified, it must be either **program** or **user**, or an abbreviation of one of these two. It indicates whether *window*'s current position was requested by the program or by the user. Many window managers ignore program-requested initial positions and ask the user to manually position the window; if **user** is specified then the window manager should position the window at the given place without asking the user for assistance. If *who* is specified as an empty string, then the current position source is cancelled. If *who* is specified, then the command returns an empty string. Otherwise it returns **user** or **window** to indicate the source of the window's current position, or an empty string if no source has been specified yet. Most window managers interpret ''no source'' as equivalent to **program**. Tk will automatically set the position source to **user** when a **wm geometry** command is invoked, unless the source has been set explicitly to **program**.

wm protocol *window* ?*name*? ?*command*?

This command is used to manage window manager protocols such as **WM_DELETE_WINDOW**. *Name* is the name of an atom corresponding to a window manager protocol, such as **WM_DELETE_WINDOW** or **WM_SAVE_YOURSELF** or **WM_TAKE_FOCUS**. If both *name* and *command* are specified, then *command* is associated with the protocol specified by *name*. *Name* will be added to *window*'s **WM_PROTOCOLS** property to tell the window manager that the application has a protocol handler for *name*, and *command* will be invoked in the future whenever the window manager sends a message to the client for that protocol. In this case the command returns an empty string. If *name* is specified but *command* isn't, then the current command for *name* is returned, or an empty string if there is no handler defined for *name*. If

command is specified as an empty string then the current handler for *name* is deleted and it is removed from the **WM_PROTOCOLS** property on *window*; an empty string is returned. Lastly, if neither *name* nor *command* is specified, the command returns a list of all the protocols for which handlers are currently defined for *window*.

Tk always defines a protocol handler for **WM_DELETE_WINDOW**, even if you haven't asked for one with **wm protocol**. If a **WM_DELETE_WINDOW** message arrives when you haven't defined a handler, then Tk handles the message by destroying the window for which it was received.

wm resizable *window* ?*width height*?

This command controls whether or not the user may interactively resize a top-level window. If *width* and *height* are specified, they are boolean values that determine whether the width and height of *window* may be modified by the user. In this case the command returns an empty string. If *width* and *height* are omitted then the command returns a list with two 0/1 elements that indicate whether the width and height of *window* are currently resizable. By default, windows are resizable in both dimensions. If resizing is disabled, then the window's size will be the size from the most recent interactive resize or **wm geometry** command. If there has been no such operation then the window's natural size will be used.

wm sizefrom *window* ?*who*?

If *who* is specified, it must be either **program** or **user**, or an abbreviation of one of these two. It indicates whether *window*'s current size was requested by the program or by the user. Some window managers ignore program-requested sizes and ask the user to manually size the window; if **user** is specified then the window manager should give the window its specified size without asking the user for assistance. If *who* is specified as an empty string, then the current size source is cancelled. If *who* is specified, then the command returns an empty string. Otherwise it returns **user** or **window** to indicate the source of the window's current size, or an empty string if no source has been specified yet. Most window managers interpret ''no source'' as equivalent to **program**.

wm state *window*

Returns the current state of *window*: either **normal**, **iconic**, **withdrawn**, or **icon**. The difference between **iconic** and **icon** is that **iconic** refers to a window that has been iconified (e.g., with the **wm iconify** command) while **icon** refers to a window whose only purpose is to serve as the icon for some other window (via the **wm iconwindow** command).

wm title *window* ?*string*?

If *string* is specified, then it will be passed to the window manager for use as the title for *window* (the window manager should display this string in *window*'s title bar). In this case the command returns an empty string. If *string* isn't specified then the command returns the current title for the *window*. The title for a window defaults to its name.

wm transient *window* ?*master*?

If *master* is specified, then the window manager is informed that *window* is a transient window (e.g. pull-down menu) working on behalf of *master* (where *master* is the path name for a top-level window). Some window managers will use this information to manage *window* specially. If *master* is specified as an empty string then *window* is marked as not being a transient window any more. If *master* is specified, then the command returns an empty string. Otherwise the command returns the path name of *window*'s current master, or an empty string if *window* isn't currently a transient window.

wm withdraw *window*

Arranges for *window* to be withdrawn from the screen. This causes the window to be unmapped and forgotten about by the window manager. If the window has never been mapped, then this command causes the window to be mapped in the withdrawn state. Not all window managers appear to know how to handle windows that are mapped in the withdrawn state. Note: it

sometimes seems to be necessary to withdraw a window and then re-map it (e.g. with **wm deiconify**) to get some window managers to pay attention to changes in window attributes such as group.

GEOMETRY MANAGEMENT

By default a top-level window appears on the screen in its *natural size*, which is the one determined internally by its widgets and geometry managers. If the natural size of a top-level window changes, then the window's size changes to match. A top-level window can be given a size other than its natural size in two ways. First, the user can resize the window manually using the facilities of the window manager, such as resize handles. Second, the application can request a particular size for a top-level window using the **wm geometry** command. These two cases are handled identically by Tk; in either case, the requested size overrides the natural size. You can return the window to its natural by invoking **wm geometry** with an empty *geometry* string.

Normally a top-level window can have any size from one pixel in each dimension up to the size of its screen. However, you can use the **wm minsize** and **wm maxsize** commands to limit the range of allowable sizes. The range set by **wm minsize** and **wm maxsize** applies to all forms of resizing, including the window's natural size as well as manual resizes and the **wm geometry** command. You can also use the command **wm resizable** to completely disable interactive resizing in one or both dimensions.

GRIDDED GEOMETRY MANAGEMENT

Gridded geometry management occurs when one of the widgets of an application supports a range of useful sizes. This occurs, for example, in a text editor where the scrollbars, menus, and other adornments are fixed in size but the edit widget can support any number of lines of text or characters per line. In this case, it is usually desirable to let the user specify the number of lines or characters-per-line, either with the **wm geometry** command or by interactively resizing the window. In the case of text, and in other interesting cases also, only discrete sizes of the window make sense, such as integral numbers of lines and characters-per-line; arbitrary pixel sizes are not useful.

Gridded geometry management provides support for this kind of application. Tk (and the window manager) assume that there is a grid of some sort within the application and that the application should be resized in terms of *grid units* rather than pixels. Gridded geometry management is typically invoked by turning on the **setGrid** option for a widget; it can also be invoked with the **wm grid** command or by calling **Tk_SetGrid**. In each of these approaches the particular widget (or sometimes code in the application as a whole) specifies the relationship between integral grid sizes for the window and pixel sizes. To return to non-gridded geometry management, invoke **wm grid** with empty argument strings.

When gridded geometry management is enabled then all the dimensions specified in **wm minsize**, **wm maxsize**, and **wm geometry** commands are treated as grid units rather than pixel units. Interactive resizing is also carried out in even numbers of grid units rather than pixels.

BUGS

Most existing window managers appear to have bugs that affect the operation of the **wm** command. For example, some changes won't take effect if the window is already active: the window will have to be withdrawn and de-iconified in order to make the change happen.

KEYWORDS

aspect ratio, deiconify, focus model, geometry, grid, group, icon, iconify, increments, position, size, title, top-level window, units, window manager

NAME

body − change the body for a class method/proc

SYNOPSIS

body *className***::***function args body*

DESCRIPTION

The **body** command is used outside of an **[incr Tcl]** class definition to define or redefine the body of a class method or proc. This facility allows a class definition to have separate "interface" and "implementation" parts. The "interface" part is a **class** command with declarations for methods, procs, instance variables and common variables. The "implementation" part is a series of **body** and **configbody** commands. If the "implementation" part is kept in a separate file, it can be sourced again and again as bugs are fixed, to support interactive development. When using the "tcl" mode in the **emacs** editor, the "interface" and "implementation" parts can be kept in the same file; as bugs are fixed, individual bodies can be highlighted and sent to the test application.

The name "*className***::***function*" identifies the method/proc being changed.

If an *args* list was specified when the *function* was defined in the class definition, the *args* list for the **body** command must match in meaning. Variable names can change, but the argument lists must have the same required arguments and the same default values for optional arguments. The special **args** argument acts as a wildcard when included in the *args* list in the class definition; it will match zero or more arguments of any type when the body is redefined.

If the *body* string starts with "**@**", it is treated as the symbolic name for a C procedure. The *args* list has little meaning for the C procedure, except to document the expected usage. (The C procedure is not guaranteed to use arguments in this manner.) If *body* does not start with "**@**", it is treated as a Tcl command script. When the function is invoked, command line arguments are matched against the *args* list, and local variables are created to represent each argument. This is the usual behavior for a Tcl-style proc.

Symbolic names for C procedures are established by registering procedures via **Itcl_RegisterC()**. This is usually done in the **Tcl_AppInit()** procedure, which is automatically called when the interpreter starts up. In the following example, the procedure `My_FooCmd()` is registered with the symbolic name "foo". This procedure can be referenced in the **body** command as "`@foo`".

```
int
Tcl_AppInit(interp)
    Tcl_Interp *interp;     /* Interpreter for application. */
{
    if (Itcl_Init(interp) == TCL_ERROR) {
        return TCL_ERROR;
    }

    if (Itcl_RegisterC(interp, "foo", My_FooCmd) != TCL_OK) {
        return TCL_ERROR;
    }
}
```

EXAMPLE

In the following example, a "File" class is defined to represent open files. The method bodies are included below the class definition via the **body** command. Note that the bodies of the constructor/destructor must be included in the class definition, but they can be redefined via the **body** command as well.

```
class File {
    private variable fid ""
```

```
    constructor {name access} {
        set fid [open $name $access]
    }
    destructor {
        close $fid
    }

    method get {}
    method put {line}
    method eof {}
}

body File::get {} {
    return [gets $fid]
}
body File::put {line} {
    puts $fid $line
}
body File::eof {} {
    return [::eof $fid]
}

#
# See the File class in action:
#
File x /etc/passwd "r"
while {![x eof]} {
    puts "=> [x get]"
}
delete object x
```

KEYWORDS

class, object, procedure

NAME

class − create a class of objects

SYNOPSIS

class *className* **{**
inherit *baseClass* ?*baseClass...*?
constructor *args* ?*init*? *body*
destructor *body*
method *name* ?*args*? ?*body*?
proc *name ?args*? ?*body*?
variable *varName* ?*init*? ?*config*?
common *varName* ?*init*?

public *command* ?*arg arg ...*?
protected *command* ?*arg arg ...*?
private *command* ?*arg arg ...*?

set *varName* ?*value*?
array *option* ?*arg arg ...*?
}

className objName ?*arg arg ...*?

objName method ?*arg arg ...*?

className::proc ?arg arg ...?

DESCRIPTION

The fundamental construct in **[incr Tcl]** is the class definition. Each class acts as a template for actual objects that can be created. The class itself is a namespace which contains things common to all objects. Each object has its own unique bundle of data which contains instances of the "variables" defined in the class definition. Each object also has a built-in variable named "this", which contains the name of the object. Classes can also have "common" data members that are shared by all objects in a class.

Two types of functions can be included in the class definition. "Methods" are functions which operate on a specific object, and therefore have access to both "variables" and "common" data members. "Procs" are ordinary procedures in the class namespace, and only have access to "common" data members.

If the body of any method or proc starts with "**@**", it is treated as the symbolic name for a C procedure. Otherwise, it is treated as a Tcl code script. See below for details on registering and using C procedures.

A class can only be defined once, although the bodies of class methods and procs can be defined again and again for interactive debugging. See the **body** and **configbody** commands for details.

Each namespace can have its own collection of objects and classes. The list of classes available in the current context can be queried using the "**itcl::find classes**" command, and the list of objects, with the "**itcl::find objects**" command.

A class can be deleted using the "**delete class**" command. Individual objects can be deleted using the "**delete object**" command.

CLASS DEFINITIONS

class *className definition*

Provides the definition for a class named *className*. If the class *className* already exists, or if a command called *className* exists in the current namespace context, this command returns an error. If the class definition is successfully parsed, *className* becomes a command in the current context, handling the creation of objects for this class.

The class *definition* is evaluated as a series of Tcl statements that define elements within the class. The following class definition commands are recognized:

inherit *baseClass* ?*baseClass...*?

Causes the current class to inherit characteristics from one or more base classes. Classes must have been defined by a previous **class** command, or must be available to the autoloading facility (see "AUTO-LOADING" below). A single class definition can contain no more than one **inherit** command.

The order of *baseClass* names in the **inherit** list affects the name resolution for class members. When the same member name appears in two or more base classes, the base class that appears first in the **inherit** list takes precedence. For example, if classes "Foo" and "Bar" both contain the member "x", and if another class has the "**inherit**" statement:
inherit Foo Bar
then the name "x" means "Foo::x". Other inherited members named "x" must be referenced with their explicit name, like "Bar::x".

constructor *args* ?*init*? *body*

Declares the *args* argument list and *body* used for the constructor, which is automatically invoked whenever an object is created.

Before the *body* is executed, the optional *init* statement is used to invoke any base class constructors that require arguments. Variables in the *args* specification can be accessed in the *init* code fragment, and passed to base class constructors. After evaluating the *init* statement, any base class constructors that have not been executed are invoked automatically without arguments. This ensures that all base classes are fully constructed before the constructor *body* is executed. By default, this scheme causes constructors to be invoked in order from least- to most-specific. This is exactly the opposite of the order that classes are reported by the **info heritage** command.

If construction is successful, the constructor always returns the object name−regardless of how the *body* is defined−and the object name becomes a command in the current namespace context. If construction fails, an error message is returned.

destructor *body*

Declares the *body* used for the destructor, which is automatically invoked when an object is deleted. If the destructor is successful, the object data is destroyed and the object name is removed as a command from the interpreter. If destruction fails, an error message is returned and the object remains.

When an object is destroyed, all destructors in its class hierarchy are invoked in order from most- to least-specific. This is the order that the classes are reported by the "**info heritage**" command, and it is exactly the opposite of the default constructor order.

method *name* ?*args*? ?*body*?

Declares a method called *name*. When the method *body* is executed, it will have automatic access to object-specific variables and common data members.

If the *args* list is specified, it establishes the usage information for this method. The **body** command can be used to redefine the method body, but the *args* list must match this

specification.

Within the body of another class method, a method can be invoked like any other command−simply by using its name. Outside of the class context, the method name must be prefaced an object name, which provides the context for the data that it manipulates. Methods in a base class that are redefined in the current class, or hidden by another base class, can be qualified using the "*className*::*method*" syntax.

proc *name* ?*args*? ?*body*?
Declares a proc called *name*. A proc is an ordinary procedure within the class namespace. Unlike a method, a proc is invoked without referring to a specific object. When the proc *body* is executed, it will have automatic access only to common data members.

If the *args* list is specified, it establishes the usage information for this proc. The **body** command can be used to redefine the proc body, but the *args* list must match this specification.

Within the body of another class method or proc, a proc can be invoked like any other command−simply by using its name. In any other namespace context, the proc is invoked using a qualified name like "*className***::***proc*". Procs in a base class that are redefined in the current class, or hidden by another base class, can also be accessed via their qualified name.

variable *varName* ?*init*? ?*config*?
Defines an object-specific variable named *varName*. All object-specific variables are automatically available in class methods. They need not be declared with anything like the **global** command.

If the optional *init* string is specified, it is used as the initial value of the variable when a new object is created. Initialization forces the variable to be a simple scalar value; uninitialized variables, on the other hand, can be set within the constructor and used as arrays.

The optional *config* script is only allowed for public variables. If specified, this code fragment is executed whenever a public variable is modified by the built-in "configure" method. The *config* script can also be specified outside of the class definition using the **configbody** command.

common *varName* ?*init*?
Declares a common variable named *varName*. Common variables reside in the class namespace and are shared by all objects belonging to the class. They are just like global variables, except that they need not be declared with the usual **global** command. They are automatically visible in all class methods and procs.

If the optional *init* string is specified, it is used as the initial value of the variable. Initialization forces the variable to be a simple scalar value; uninitialized variables, on the other hand, can be set with subsequent **set** and **array** commands and used as arrays.

Once a common data member has been defined, it can be set using **set** and **array** commands within the class definition. This allows common data members to be initialized as arrays. For example:

```
class Foo {
  common boolean
  set boolean(true) 1
```

```
    set boolean(false) 0
}
```

Note that if common data members are initialized within the constructor, they get initialized again and again whenever new objects are created.

public *command* ?*arg arg ...*?

protected *command* ?*arg arg ...*?

private *command* ?*arg arg ...*?

These commands are used to set the protection level for class members that are created when *command* is evaluated. The *command* is usually **method**, **proc**, **variable** or **common**, and the remaining *arg*'s complete the member definition. However, *command* can also be a script containing many different member definitions, and the protection level will apply to all of the members that are created.

CLASS USAGE

Once a class has been defined, the class name can be used as a command to create new objects belonging to the class.

className objName ?*args...*?

Creates a new object in class *className* with the name *objName*. Remaining arguments are passed to the constructor of the most-specific class. This in turn passes arguments to base class constructors before invoking its own body of commands. If construction is successful, a command called *objName* is created in the current namespace context, and *objName* is returned as the result of this operation. If an error is encountered during construction, the destructors are automatically invoked to free any resources that have been allocated, the object is deleted, and an error is returned.

If *objName* contains the string "**#auto**", that string is replaced with an automatically generated name. Names have the form *className*<*number*>, where the *className* part is modified to start with a lowercase letter. In class "Toaster", for example, the "**#auto**" specification would produce names like toaster0, toaster1, etc. Note that "**#auto**" can be also be buried within an object name:
fileselectiondialog .foo.bar.#auto -background red
This would generate an object named ".foo.bar.fileselectiondialog0".

OBJECT USAGE

Once an object has been created, the object name can be used as a command to invoke methods that operate on the object.

objName method ?*args...*?

Invokes a method named *method* on an object named *objName*. Remaining arguments are passed to the argument list for the method. The method name can be "constructor", "destructor", any method name appearing in the class definition, or any of the following built-in methods.

BUILT-IN METHODS

objName **cget option**

Provides access to public variables as configuration options. This mimics the behavior of the usual "cget" operation for Tk widgets. The *option* argument is a string of the form "**-***varName*", and this method returns the current value of the public variable *varName*.

objName **configure** ?*option*? ?*value option value ...*?

Provides access to public variables as configuration options. This mimics the behavior of the usual "configure" operation for Tk widgets. With no arguments, this method returns a list of lists describing all of the public variables. Each list has three elements: the variable name, its initial

value and its current value.

If a single *option* of the form "*-varName*" is specified, then this method returns the information for that one variable.

Otherwise, the arguments are treated as *option*/*value* pairs assigning new values to public variables. Each variable is assigned its new value, and if it has any "config" code associated with it, it is executed in the context of the class where it was defined. If the "config" code generates an error, the variable is set back to its previous value, and the **configure** method returns an error.

objName **isa** *className*

Returns non-zero if the given *className* can be found in the object's heritage, and zero otherwise.

objName **info** *option* ?*args...*?

Returns information related to a particular object named *objName*, or to its class definition. The *option* parameter includes the following things, as well as the options recognized by the usual Tcl "info" command:

objName **info class**

Returns the name of the most-specific class for object *objName*.

objName **info inherit**

Returns the list of base classes as they were defined in the "**inherit**" command, or an empty string if this class has no base classes.

objName **info heritage**

Returns the current class name and the entire list of base classes in the order that they are traversed for member lookup and object destruction.

objName **info function** ?*cmdName*? ?**-protection**? ?**-type**? ?**-name**? ?**-args**? ?**-body**?

With no arguments, this command returns a list of all class methods and procs. If *cmdName* is specified, it returns information for a specific method or proc. If no flags are specified, this command returns a list with the following elements: the protection level, the type (method/proc), the qualified name, the argument list and the body. Flags can be used to request specific elements from this list.

objName **info variable** ?*varName*? ?**-protection**? ?**-type**? ?**-name**? ?**-init**? ?**-value**? ?**-config**?

With no arguments, this command returns a list of all object-specific variables and common data members. If *varName* is specified, it returns information for a specific data member. If no flags are specified, this command returns a list with the following elements: the protection level, the type (variable/common), the qualified name, the initial value, and the current value. If *varName* is a public variable, the "config" code is included on this list. Flags can be used to request specific elements from this list.

CHAINING METHODS/PROCS

Sometimes a base class has a method or proc that is redefined with the same name in a derived class. This is a way of making the derived class handle the same operations as the base class, but with its own specialized behavior. For example, suppose we have a Toaster class that looks like this:

```
class Toaster {
  variable crumbs 0
  method toast {nslices} {
    if {$crumbs > 50} {
      error "== FIRE! FIRE! =="
    }
    set crumbs [expr $crumbs+4*$nslices]
```

```
    }
    method clean {} {
        set crumbs 0
    }
}
```

We might create another class like SmartToaster that redefines the "toast" method. If we want to access the base class method, we can qualify it with the base class name, to avoid ambiguity:

```
class SmartToaster {
    inherit Toaster
    method toast {nslices} {
        if {$crumbs > 40} {
            clean
        }
        return [Toaster::toast $nslices]
    }
}
```

Instead of hard-coding the base class name, we can use the "chain" command like this:

```
class SmartToaster {
    inherit Toaster
    method toast {nslices} {
        if {$crumbs > 40} {
            clean
        }
        return [chain $nslices]
    }
}
```

The chain command searches through the class hierarchy for a slightly more generic (base class) implementation of a method or proc, and invokes it with the specified arguments. It starts at the current class context and searches through base classes in the order that they are reported by the "info heritage" command. If another implementation is not found, this command does nothing and returns the null string.

AUTO-LOADING

Class definitions need not be loaded explicitly; they can be loaded as needed by the usual Tcl auto-loading facility. Each directory containing class definition files should have an accompanying "tclIndex" file. Each line in this file identifies a Tcl procedure or **[incr Tcl]** class definition and the file where the definition can be found.

For example, suppose a directory contains the definitions for classes "Toaster" and "SmartToaster". Then the "tclIndex" file for this directory would look like:

```
# Tcl autoload index file, version 2.0 for [incr Tcl]
# This file is generated by the "auto_mkindex" command
# and sourced to set up indexing information for one or
# more commands.  Typically each line is a command that
# sets an element in the auto_index array, where the
# element name is the name of a command and the value is
# a script that loads the command.

set auto_index(::Toaster) "source $dir/Toaster.itcl"
set auto_index(::SmartToaster) "source $dir/SmartToaster.itcl"
```

The **auto_mkindex** command is used to automatically generate "tclIndex" files.

The auto-loader must be made aware of this directory by appending the directory name to the "auto_path" variable. When this is in place, classes will be auto-loaded as needed when used in an application.

C PROCEDURES

C procedures can be integrated into an **[incr Tcl]** class definition to implement methods, procs, and the "config" code for public variables. Any body that starts with "**@**" is treated as the symbolic name for a C procedure.

Symbolic names are established by registering procedures via **Itcl_RegisterC()**. This is usually done in the **Tcl_AppInit()** procedure, which is automatically called when the interpreter starts up. In the following example, the procedure `My_FooCmd()` is registered with the symbolic name "foo". This procedure can be referenced in the **body** command as "`@foo`".

```
int
Tcl_AppInit(interp)
    Tcl_Interp *interp;     /* Interpreter for application. */
{
    if (Itcl_Init(interp) == TCL_ERROR) {
        return TCL_ERROR;
    }

    if (Itcl_RegisterC(interp, "foo", My_FooCmd) != TCL_OK) {
        return TCL_ERROR;
    }
}
```

C procedures are implemented just like ordinary Tcl commands. See the **CrtCommand** man page for details. Within the procedure, class data members can be accessed like ordinary variables using **Tcl_SetVar()**, **Tcl_GetVar()**, **Tcl_TraceVar()**, etc. Class methods and procs can be executed like ordinary commands using **Tcl_Eval()**. **[incr Tcl]** makes this possible by automatically setting up the context before executing the C procedure.

This scheme provides a natural migration path for code development. Classes can be developed quickly using Tcl code to implement the bodies. An entire application can be built and tested. When necessary, individual bodies can be implemented with C code to improve performance.

KEYWORDS

class, object, object-oriented

NAME

code − capture the namespace context for a code fragment

SYNOPSIS

code ?**-namespace** *name*? *command* ?*arg arg ...*?

DESCRIPTION

Creates a scoped value for the specified *command* and its associated *arg* arguments. A scoped value is a list with three elements: the "`@scope`" keyword, a namespace context, and a value string. For example, the command

```
namespace foo {
  code puts "Hello World!"
}
```

produces the scoped value:

```
@scope ::foo {puts {Hello World!}}
```

Note that the **code** command captures the current namespace context. If the **-namespace** flag is specified, then the current context is ignored, and the *name* string is used as the namespace context.

Extensions like Tk execute ordinary code fragments in the global namespace. A scoped value captures a code fragment together with its namespace context in a way that allows it to be executed properly later. It is needed, for example, to wrap up code fragments when a Tk widget is used within a namespace:

```
namespace foo {
  private proc report {mesg} {
    puts "click: $mesg"
  }

  button .b1 -text "Push Me"       -command [code report "Hello World!"]
  pack .b1
}
```

The code fragment associated with button `.b1` only makes sense in the context of namespace "foo". Furthermore, the "report" procedure is private, and can only be accessed within that namespace. The **code** command wraps up the code fragment in a way that allows it to be executed properly when the button is pressed.

Also, note that the **code** command preserves the integrity of arguments on the command line. This makes it a natural replacement for the **list** command, which is often used to format Tcl code fragments. In other words, instead of using the **list** command like this:

```
after 1000 [list puts "Hello $name!"]
```

use the **code** command like this:

```
after 1000 [code puts "Hello $name!"]
```

This not only formats the command correctly, but also captures its namespace context.

Scoped commands can be invoked like ordinary code fragments, with or without the **eval** command. For example, the following statements work properly:

```
set cmd {@scope ::foo .b1}
$cmd configure -background red

set opts {-bg blue -fg white}
eval $cmd configure $opts
```

Note that scoped commands by-pass the usual protection mechanisms; the command:

```
@scope ::foo {report {Hello World!}}
```

can be used to access the "foo::report" proc from any namespace context, even though it is private.

KEYWORDS

scope, callback, namespace, public, protected, private

NAME

configbody − change the "config" code for a public variable

SYNOPSIS

configbody *className***::***varName body*

DESCRIPTION

The **configbody** command is used outside of an **[incr Tcl]** class definition to define or redefine the configuration code associated with a public variable. Public variables act like configuration options for an object. They can be modified outside the class scope using the built-in **configure** method. Each variable can have a bit of "config" code associate with it that is automatically executed when the variable is configured. The **configbody** command can be used to define or redefine this body of code.

Like the **body** command, this facility allows a class definition to have separate "interface" and "implementation" parts. The "interface" part is a **class** command with declarations for methods, procs, instance variables and common variables. The "implementation" part is a series of **body** and **configbody** commands. If the "implementation" part is kept in a separate file, it can be sourced again and again as bugs are fixed, to support interactive development. When using the "tcl" mode in the **emacs** editor, the "interface" and "implementation" parts can be kept in the same file; as bugs are fixed, individual bodies can be highlighted and sent to the test application.

The name "*className***::***varName*" identifies the public variable being updated. If the *body* string starts with "**@**", it is treated as the symbolic name for a C procedure. Otherwise, it is treated as a Tcl command script.

Symbolic names for C procedures are established by registering procedures via **Itcl_RegisterC()**. This is usually done in the **Tcl_AppInit()** procedure, which is automatically called when the interpreter starts up. In the following example, the procedure `My_FooCmd()` is registered with the symbolic name "foo". This procedure can be referenced in the **configbody** command as "`@foo`".

```
int
Tcl_AppInit(interp)
    Tcl_Interp *interp;     /* Interpreter for application. */
{
    if (Itcl_Init(interp) == TCL_ERROR) {
        return TCL_ERROR;
    }

    if (Itcl_RegisterC(interp, "foo", My_FooCmd) != TCL_OK) {
        return TCL_ERROR;
    }
}
```

EXAMPLE

In the following example, a "File" class is defined to represent open files. Whenever the "-name" option is configured, the existing file is closed, and a new file is opened. Note that the "config" code for a public variable is optional. The "-access" option, for example, does not have it.

```
class File {
    private variable fid ""

    public variable name ""
    public variable access "r"

    constructor {args} {
```

```
        eval configure $args
    }
    destructor {
        if {$fid != ""} {
            close $fid
        }
    }

    method get {}
    method put {line}
    method eof {}
}

body File::get {} {
    return [gets $fid]
}
body File::put {line} {
    puts $fid $line
}
body File::eof {} {
    return [::eof $fid]
}

configbody File::name {
    if {$fid != ""} {
        close $fid
    }
    set fid [open $name $access]
}

#
# See the File class in action:
#
File x

x configure -name /etc/passwd
while {![x eof]} {
    puts "=> [x get]"
}
delete object x
```

KEYWORDS

class, object, variable, configure

NAME

delete − delete things in the interpreter

SYNOPSIS

delete *option* ?*arg arg ...*?

DESCRIPTION

The **delete** command is used to delete things in the interpreter. It is implemented as an ensemble, so extensions can add their own options and extend the behavior of this command. By default, the **delete** command handles the destruction of namespaces.

The *option* argument determines what action is carried out by the command. The legal *options* (which may be abbreviated) are:

delete class *name* ?*name...*?

Deletes one or more **[incr Tcl]** classes called *name*. This deletes all objects in the class, and all derived classes as well.

If an error is encountered while destructing an object, it will prevent the destruction of the class and any remaining objects. To destroy the entire class without regard for errors, use the "**delete namespace**" command.

delete object *name* ?*name...*?

Deletes one or more **[incr Tcl]** objects called *name*. An object is deleted by invoking all destructors in its class hierarchy, in order from most- to least-specific. If all destructors are successful, data associated with the object is deleted and the *name* is removed as a command from the interpreter.

If the access command for an object resides in another namespace, then its qualified name can be used:
delete object foo::bar::x

If an error is encountered while destructing an object, the **delete** command is aborted and the object remains alive. To destroy an object without regard for errors, use the "**rename**" command to destroy the object access command.

delete namespace *name* ?*name...*?

Deletes one or more namespaces called *name*. This deletes all commands and variables in the namespace, and deletes all child namespaces as well. When a namespace is deleted, it is automatically removed from the import lists of all other namespaces.

KEYWORDS

namespace, proc, variable, ensemble

NAME

ensemble − create or modify a composite command

SYNOPSIS

ensemble *ensName* ?*command arg arg...*?
or
ensemble *ensName* {
  part *partName args body*
  ...
  ensemble *partName* {
    part *subPartName args body*
    part *subPartName args body*
  ...
  }
}

DESCRIPTION

The **ensemble** command is used to create or modify a composite command. See the section **WHAT IS AN ENSEMBLE?** below for a brief overview of ensembles.

If the **ensemble** command finds an existing ensemble called *ensName*, it updates that ensemble. Otherwise, it creates an ensemble called *ensName*. If the *ensName* is a simple name like "foo", then an ensemble command named "foo" is added to the current namespace context. If a command named "foo" already exists in that context, then it is deleted. If the *ensName* contains namespace qualifiers like "a::b::foo", then the namespace path is resolved, and the ensemble command is added that namespace context. Parent namespaces like "a" and "b" are created automatically, as needed.

If the *ensName* contains spaces like "a::b::foo bar baz", then additional words like "bar" and "baz" are treated as sub-ensembles. Sub-ensembles are merely parts within an ensemble; they do not have a Tcl command associated with them. An ensemble like "foo" can have a sub-ensemble called "foo bar", which in turn can have a sub-ensemble called "foo bar baz". In this case, the sub-ensemble "foo bar" must be created before the sub-ensemble "foo bar baz" that resides within it.

If there are any arguments following *ensName*, then they are treated as commands, and they are executed to update the ensemble. The following commands are recognized in this context: **part** and **ensemble**.

The **part** command defines a new part for the ensemble. Its syntax is identical to the usual **proc** command, but it defines a part within an ensemble, instead of a Tcl command. If a part called *partName* already exists within the ensemble, then the **part** command returns an error.

The **ensemble** command can be nested inside another **ensemble** command to define a sub-ensemble.

WHAT IS AN ENSEMBLE?

The usual "info" command is a composite command--the command name **info** must be followed by a subcommand like **body** or **globals**. We will refer to a command like **info** as an *ensemble*, and to sub-commands like **body** or **globals** as its *parts*.

Ensembles can be nested. For example, the **info** command has an ensemble **info namespace** within it. This ensemble has parts like **info namespace all** and **info namespace children**.

With ensembles, composite commands can be created and extended in an automatic way. Any package can find an existing ensemble and add new parts to it. So extension writers can add their own parts, for example, to the **info** command.

The ensemble facility manages all of the part names and keeps track of unique abbreviations. Normally, you can abbreviate **info complete** to **info comp**. But if an extension adds the part **info complexity**, the minimum abbreviation for **info complete** becomes **info complet**.

The ensemble facility not only automates the construction of composite commands, but it automates the error handling as well. If you invoke an ensemble command without specifying a part name, you get an automatically generated error message that summarizes the usage information. For example, when the **info** command is invoked without any arguments, it produces the following error message:

```
wrong # args: should be one of...
  info args procname
  info body procname
  info cmdcount
  info commands ?pattern?
  info complete command
  info context
  info default procname arg varname
  info exists varName
  info globals ?pattern?
  info level ?number?
  info library
  info locals ?pattern?
  info namespace option ?arg arg ...?
  info patchlevel
  info procs ?pattern?
  info protection ?-command? ?-variable? name
  info script
  info tclversion
  info vars ?pattern?
  info which ?-command? ?-variable? ?-namespace? name
```

You can also customize the way an ensemble responds to errors. When an ensemble encounters an unspecified or ambiguous part name, it looks for a part called **@error**. If it exists, then it is used to handle the error. This part will receive all of the arguments on the command line starting with the offending part name. It can find another way of resolving the command, or generate its own error message.

EXAMPLE

We could use an ensemble to clean up the syntax of the various "wait" commands in Tcl/Tk. Instead of using a series of strange commands like this:

```
vwait x
tkwait visibility .top
tkwait window .
```

we could use commands with a uniform syntax, like this:

```
wait variable x
wait visibility .top
wait window .
```

The Tcl package could define the following ensemble:

```
ensemble wait part variable {name} {
   uplevel vwait $name
}
```

The Tk package could add some options to this ensemble, with a command like this:

```
ensemble wait {
   part visibility {name} {
      tkwait visibility $name
```

```
    }
    part window {name} {
        tkwait window $name
    }
}
```

Other extensions could add their own parts to the **wait** command too.

KEYWORDS

ensemble, part, info

NAME

find − search for classes and objects

SYNOPSIS

find *option* ?*arg arg ...*?

DESCRIPTION

The **find** command is used to find classes and objects that are available in the current context. A class or object is "available" if its access command can be found in the current namespace context or in the global namespace. Therefore, classes and objects created in the global namespace are available to all other namespaces in a program. Classes and objects created in one namespace can also be imported into another using the **namespace import** command.

The *option* argument determines what action is carried out by the command. The legal *options* (which may be abbreviated) are:

find classes ?*pattern*?

Returns a list of classes found in the current namespace context. If the optional *pattern* is specified, then the reported names are compared using the rules of the "**string match**" command, and only matching names are reported.

If a class resides in the current namespace context, this command reports its simple name--without any qualifiers. However, if the *pattern* contains **::** qualifiers, or if the class resides in another context, this command reports its fully-qualified name.

find objects ?*pattern*? ?**-class** *className*? ?**-isa** *className*?

Returns a list of objects found in the current namespace context. If the optional *pattern* is specified, then the reported names are compared using the rules of the "**string match**" command, and only matching names are reported. If the optional "**-class**" parameter is specified, this list is restricted to objects whose most-specific class is *className*. If the optional "**-isa**" parameter is specified, this list is further restricted to objects having the given *className* anywhere in their heritage.

If an object resides in the current namespace context, this command reports its simple name--without any qualifiers. However, if the *pattern* contains **::** qualifiers, or if the object resides in another context, this command reports its fully-qualified name.

KEYWORDS

class, object, search, import

NAME

itcl − object-oriented extensions to Tcl

DESCRIPTION

[incr Tcl] provides object-oriented extensions to Tcl, much as C++ provides object-oriented extensions to C. The emphasis of this work, however, is not to create a whiz-bang object-oriented programming environment. Rather, it is to support more structured programming practices in Tcl without changing the flavor of the language. More than anything else, **[incr Tcl]** provides a means of encapsulating related procedures together with their shared data in a namespace that is hidden from the outside world. It encourages better programming by promoting the object-oriented "library" mindset. It also allows for code re-use through inheritance.

CLASSES

The fundamental construct in **[incr Tcl]** is the class definition. Each class acts as a template for actual objects that can be created. Each object has its own unique bundle of data, which contains instances of the "variables" defined in the class. Special procedures called "methods" are used to manipulate individual objects. Methods are just like the operations that are used to manipulate Tk widgets. The "**button**" widget, for example, has methods such as "flash" and "invoke" that cause a particular button to blink and invoke its command.

Within the body of a method, the "variables" defined in the class are automatically available. They need not be declared with anything like the **global** command. Within another class method, a method can be invoked like any other command−simply by using its name. From any other context, the method name must be prefaced by an object name, which provides a context for the data that the method can access.

Each class has its own namespace containing things that are common to all objects which belong to the class. For example, "common" data members are shared by all objects in the class. They are global variables that exist in the class namespace, but since they are included in the class definition, they need not be declared using the **global** command; they are automatically available to any code executing in the class context. A class can also create ordinary global variables, but these must be declared using the **global** command each time they are used.

Classes can also have ordinary procedures declared as "procs". Within another class method or proc, a proc can be invoked like any other command−simply by using its name. From any other context, the procedure name should be qualified with the class namespace like "*className*::*proc*". Class procs execute in the class context, and therefore have automatic access to all "common" data members. However, they cannot access object-specific "variables", since they are invoked without reference to any specific object. They are usually used to perform generic operations which affect all objects belonging to the class.

Each of the elements in a class can be declared "public", "protected" or "private". Public elements can be accessed by the class, by derived classes (other classes that inherit this class), and by external clients that use the class. Protected elements can be accessed by the class, and by derived classes. Private elements are only accessible in the class where they are defined.

The "public" elements within a class define its interface to the external world. Public methods define the operations that can be used to manipulate an object. Public variables are recognized as configuration options by the "configure" and "cget" methods that are built into each class. The public interface says *what* an object will do but not *how* it will do it. Protected and private members, along with the bodies of class methods and procs, provide the implementation details. Insulating the application developer from these details leaves the class designer free to change them at any time, without warning, and without affecting programs that rely on the class. It is precisely this encapsulation that makes object-oriented programs easier to understand and maintain.

The fact that **[incr Tcl]** objects look like Tk widgets is no accident. **[incr Tcl]** was designed this way, to blend naturally into a Tcl/Tk application. But **[incr Tcl]** extends the Tk paradigm from being merely object-based to being fully object-oriented. An object-oriented system supports inheritance, allowing classes to share common behaviors by inheriting them from an ancestor or base class. Having a base class as a common abstraction allows a programmer to treat related classes in a similar manner. For example, a toaster and a blender perform different (specialized) functions, but both share the abstraction of being appliances. By abstracting common behaviors into a base class, code can be *shared* rather than *copied*. The resulting application is easier to understand and maintain, and derived classes (e.g., specialized appliances) can be added or removed more easily.

This description was merely a brief overview of object-oriented programming and **[incr Tcl]**. A more tutorial introduction is presented in the paper included with this distribution. See the **class** command for more details on creating and using classes.

NAMESPACES

[incr Tcl] now includes a complete namespace facility. A namespace is a collection of commands and global variables that is kept apart from the usual global scope. This allows Tcl code libraries to be packaged in a well-defined manner, and prevents unwanted interactions with other libraries. A namespace can also have child namespaces within it, so one library can contain its own private copy of many other libraries. A namespace can also be used to wrap up a group of related classes. The global scope (named "**::**") is the root namespace for an interpreter; all other namespaces are contained within it.

See the **namespace** command for details on creating and using namespaces.

MEGA-WIDGETS

Mega-widgets are high-level widgets that are constructed using Tk widgets as component parts, usually without any C code. A fileselectionbox, for example, may have a few listboxes, some entry widgets and some control buttons. These individual widgets are put together in a way that makes them act like one big widget.

[incr Tk] is a framework for building mega-widgets. It uses **[incr Tcl]** to support the object paradigm, and adds base classes which provide default widget behaviors. See the **itk** man page for more details.

[incr Widgets] is a library of mega-widgets built using **[incr Tk]**. It contains more than 30 different widget classes that can be used right out of the box to build Tcl/Tk applications. Each widget class has its own man page describing the features available.

KEYWORDS

class, object, object-oriented, namespace, mega-widget

NAME

itcl_class − create a class of objects (obsolete)

SYNOPSIS

itcl_class *className* **{**
inherit *baseClass* ?*baseClass...*?
constructor *args* ?*init*? *body*
destructor *body*
method *name args body*
proc *name args body*
public *varName* ?*init*? ?*config*?
protected *varName* ?*init*?
common *varName* ?*init*?
}

className objName ?*args...*?
className **#auto** ?*args...*?
className **::** *proc* ?*args...*?

objName method ?*args...*?

Commands available within class methods/procs:
global *varName* ?*varName...*?
previous *command* ?*args...*?
virtual *command* ?*args...*?

DESCRIPTION

This command is considered obsolete, but is retained for backward-compatibility with earlier versions of **[incr Tcl]**. It has been replaced by the **class** command, which should be used for any new development.

itcl_class *className definition*

Provides the definition for a class named *className*. If *className* is already defined, then this command returns an error. If the class definition is successfully parsed, *className* becomes a command in the current namespace context, handling the creation of objects and providing access to class scope. The class *definition* is evaluated as a series of Tcl statements that define elements within the class. In addition to the usual commands, the following class definition commands are recognized:

inherit *baseClass* ?*baseClass...*?

Declares one or more base classes, causing the current class to inherit their characteristics. Classes must have been defined by a previous **itcl_class** command, or must be available to the auto-loading facility (see "AUTO-LOADING" below). A single class definition can contain no more than one **inherit** command.

When the same member name appears in two or more base classes, the base class that appears first in the **inherit** list takes precedence. For example, if classes "Foo" and "Bar" both contain the member "x", then the "**inherit**" statement:

inherit Foo Bar

allows "Foo::x" to be accessed simply as "x" but forces "Bar::x" (and all other inherited members named "x") to be referenced with their explicit "*class*::*member*" name.

constructor *args* ?*init*? *body*

Declares the argument list and body used for the constructor, which is automatically invoked whenever an object is created. Before the *body* is executed, the optional *init* statement is used to invoke any base class constructors that require arguments. Variables in the *args* specification can be accessed in the *init* code fragment, and passed to base class constructors. After evaluating the *init* statement, any base class constructors that have not been executed are invoked without arguments. This ensures that all base classes are fully constructed before the constructor *body* is executed. If construction is successful, the constructor always returns the object name−regardless of how the *body* is defined−and the object name becomes a command in the current namespace context. If construction fails, an error message is returned.

destructor *body*

Declares the body used for the destructor, which is automatically invoked whenever an object is deleted. If the destructor is successful, the object data is destroyed and the object name is removed as a command from the interpreter. If destruction fails, an error message is returned and the object remains.

When an object is destroyed, all destructors in a class hierarchy are invoked in order from most- to least-specific. This is the order that the classes are reported by the "**info heritage**" command, and it is exactly the opposite of the default constructor order.

method *name args body*

Declares a method called *name* with an argument list *args* and a *body* of Tcl statements. A method is just like the usual Tcl "proc" except that it has transparent access to object-specific variables, as well as common variables. Within the class scope, a method can be invoked like any other command−simply by using its name. Outside of the class scope, the method name must be prefaced by an object name. Methods in a base class that are redefined in the current class or hidden by another base class can be explicitly scoped using the "*class*::*method*" syntax.

proc *name args body*

Declares a proc called *name* with an argument list *args* and a *body* of Tcl statements. A proc is similar to a method, except that it can be invoked without referring to a specific object, and therefore has access only to common variables−not to object-specific variables declared with the **public** and **protected** commands. Within the class scope, a proc can be invoked like any other command−simply by using its name. In any other namespace context, the proc is invoked using a qualified name like "*className*::*proc*". Procs in a base class that are redefined in the current class, or hidden by another base class, can also be accessed via their qualified name.

public *varName* ?*init*? ?*config*?

Declares a public variable named *varName*. Public variables are visible in methods within the scope of their class and any derived class. In addition, they can be modified outside of the class scope using the special "config" formal argument (see "ARGUMENT LISTS" above). If the optional *init* is specified, it is used as the initial value of the variable when a new object is created. If the optional *config* command is specified, it is invoked whenever a public variable is modified via the "config" formal argument; if the *config* command returns an error, the public variable is reset to its value before configuration, and the method handling the configuration returns an error.

protected *varName* ?*init*?

Declares a protected variable named *varName*. Protected variables are visible in methods within the scope of their class and any derived class, but cannot be modified outside of the class scope. If the optional *init* is specified, it is used as the initial value of the variable when a new object is created. Initialization forces the variable to be a simple scalar

value; uninitialized variables, on the other hand, can be used as arrays. All objects have a built-in protected variable named "this" which is initialized to the instance name for the object.

common *varName* ?*init*?

Declares a common variable named *varName*. Common variables are shared among all objects in a class. They are visible in methods and procs in the scope of their class and any derived class, but cannot be modified outside of the class scope. If the optional *init* is specified, it is used as the initial value of the variable. Initialization forces the variable to be a simple scalar value; uninitialized variables, on the other hand, can be used as arrays.

Once a common variable has been declared, it can be configured using ordinary Tcl code within the class definition. This facility is particularly useful when the initialization of the variable is non-trivial−when the variable contains an array of values, for example:

```
itcl_class Foo {
    .
    .
    common boolean
    set boolean(true) 1
    set boolean(false) 0
}
```

CLASS USAGE

When a class definition has been loaded (or made available to the auto-loader), the class name can be used as a command.

className objName ?*args...*?

Creates a new object in class *className* with the name *objName*. Remaining arguments are passed to the constructor. If construction is successful, the object name is returned and this name becomes a command in the current namespace context. Otherwise, an error is returned.

className #auto ?*args...*?

Creates a new object in class *className* with an automatically generated name. Names are of the form *className*<*number*>, where the *className* part is modified to start with a lowercase letter. In class "Toaster", for example, the "**#auto**" specification would produce names toaster0, toaster1, etc. Remaining arguments are passed to the constructor. If construction is successful, the object name is returned and this name becomes a command in the current namespace context. Otherwise, an error is returned.

className :: *proc* ?*args...*?

Used outside of the class scope to invoke a class proc named *proc*. Class procs are like ordinary Tcl procs, except that they are executed in the scope of the class and therefore have transparent access to common data members.

Notice that, unlike any other scope qualifier in **[incr Tcl]**, the "::" shown above is surrounded by spaces. This is unnecessary with the new namespace facility, and is considered obsolete. The capability is still supported, however, to provide backward-compatibility with earlier versions.

OBJECT USAGE

objName method ?*args...*?

Invokes a method named *method* to operate on the specified object. Remaining arguments are passed to the method. The method name can be "constructor", "destructor", any method name appearing in the class definition, or any of the following built-in methods.

BUILT-IN METHODS

objName **isa** *className*

Returns non-zero if the given *className* can be found in the object's heritage, and zero otherwise.

objName **delete**

Invokes the destructor associated with an object. If the destructor is successful, data associated with the object is deleted and *objName* is removed as a command from the interpreter. Returns the empty string, regardless of the destructor body.

The built-in **delete** method has been replaced by the "**delete object**" command in the global namespace, and is considered obsolete. The capability is still supported, however, to provide backward-compatibility with earlier versions.

objName **info** *option* ?*args...*?

Returns information related to the class definition or to a particular object named *objName*. The *option* parameter includes the following things, as well as the options recognized by the usual Tcl "info" command:

objName **info class**

Returns the name of the most-specific class for object *objName*.

objName **info inherit**

Returns the list of base classes as they were defined in the "**inherit**" command, or an empty string if this class has no base classes.

objName **info heritage**

Returns the current class name and the entire list of base classes in the order that they are traversed for member lookup and object destruction.

objName **info method** ?*methodName*? ?**-args**? ?**-body**?

With no arguments, this command returns a list of all class methods. If *methodName* is specified, it returns information for a specific method. If neither of the optional **-args** or **-body** flags is specified, a complete method definition is returned as a list of three elements including the method name, argument list and body. Otherwise, the requested information is returned without the method name. If the *methodName* is not recognized, an empty string is returned.

objName **info proc** ?*procName*? ?**-args**? ?**-body**?

With no arguments, this command returns a list of all class procs. If *procName* is specified, it returns information for a specific proc. If neither of the optional **-args** or **-body** flags is specified, a complete proc definition is returned as a list of three elements including the proc name, argument list and body. Otherwise, the requested information is returned without the proc name. If the *procName* is not recognized, an empty string is returned.

objName **info public** ?*varName*? ?**-init**? ?**-value**? ?**-config**?

With no arguments, this command returns a list of all public variables. If *varName* is specified, it returns information for a specific public variable. If none of the optional **-init**, **-value** or **-config** flags are specified, all available information is returned as a list of four elements including the variable name, initial value, current value, and configuration commands. Otherwise, the requested information is returned without the variable name. If the *varName* is not recognized, an empty string is returned.

objName **info protected** ?*varName*? ?**-init**? ?**-value**?

With no arguments, this command returns a list of all protected variables. If *varName* is specified, it returns information for a specific protected variable. If neither of the optional **-init** or **-value** flags is specified, all available information is returned as a list of three elements including the variable name, initial value and current value. Otherwise, the

requested information is returned without the variable name. If the *varName* is not recognized, an empty string is returned.

objName **info common** ?*varName*? ?**-init**? ?**-value**?
With no arguments, this command returns a list of all common variables. If *varName* is specified, it returns information for a specific common variable. If neither of the optional **-init** or **-value** flags is specified, all available information is returned as a list of three elements including the variable name, initial value and current value. Otherwise, the requested information is returned without the variable name. If the *varName* is not recognized, an empty string is returned.

OTHER BUILT-IN COMMANDS

The following commands are also available within the scope of each class. They cannot be accessed from outside of the class as proper methods or procs; rather, they are useful inside the class when implementing its functionality.

global *varName* ?*varName...*?
Creates a link to one or more global variables in the current namespace context. Global variables can also be accessed in other namespaces by including namespace qualifiers in *varName*. This is useful when communicating with Tk widgets that rely on global variables.

previous *command* ?*args...*?
Invokes *command* in the scope of the most immediate base class (i.e., the "previous" class) for the object. For classes using single inheritance, this facility can be used to avoid hard-wired base class references of the form "*class*::*command*", making code easier to maintain. For classes using multiple inheritance, the utility of this function is dubious. If the class at the relevant scope has no base class, an error is returned.

virtual *command* ?*args...*?
Invokes *command* in the scope of the most-specific class for the object. The methods within a class are automatically virtual; whenever an unqualified method name is used, it always refers to the most-specific implementation for that method. This function provides a way of evaluating code fragments in a base class that have access to the most-specific object information. It is useful, for example, for creating base classes that can capture and save an object's state. It inverts the usual notions of object-oriented programming, however, and should therefore be used sparingly.

AUTO-LOADING

Class definitions need not be loaded explicitly; they can be loaded as needed by the usual Tcl auto-loading facility. Each directory containing class definition files should have an accompanying "tclIndex" file. Each line in this file identifies a Tcl procedure or **[incr Tcl]** class definition and the file where the definition can be found.

For example, suppose a directory contains the definitions for classes "Toaster" and "SmartToaster". Then the "tclIndex" file for this directory would look like:

```
# Tcl autoload index file, version 2.0 for [incr Tcl]
# This file is generated by the "auto_mkindex" command
# and sourced to set up indexing information for one or
# more commands.  Typically each line is a command that
# sets an element in the auto_index array, where the
# element name is the name of a command and the value is
# a script that loads the command.

set auto_index(::Toaster) "source $dir/Toaster.itcl"
set auto_index(::SmartToaster) "source $dir/SmartToaster.itcl"
```

The **auto_mkindex** command is used to automatically
generate "tclIndex" files.

The auto-loader must be made aware of this directory by appending the directory name to the "auto_path" variable. When this is in place, classes will be auto-loaded as needed when used in an application.

KEYWORDS

class, object, object-oriented

NAME

itcl_info − query info regarding classes and objects (obsolete)

SYNOPSIS

itcl_info classes ?*pattern*?
itcl_info objects ?*pattern*? ?**-class** *className*? ?**-isa** *className*?

DESCRIPTION

This command is considered obsolete, but is retained for backward-compatibility with earlier versions of **[incr Tcl]**. It has been replaced by the "**info classes**" and "**info objects**" commands, which should be used for any new development.

The following commands are available in the global namespace to query information about classes and objects that have been created.

itcl_info classes ?*pattern*?

Returns a list of classes available in the current namespace context. If a class belongs to the current namespace context, its simple name is reported; otherwise, if a class is imported from another namespace, its fully-qualified name is reported.

If the optional *pattern* is specified, then the reported names are compared using the rules of the "**string match**" command, and only matching names are reported.

itcl_info objects ?*pattern*? ?**-class** *className*? ?**-isa** *className*?

Returns a list of objects available in the current namespace context. If an object belongs to the current namespace context, its simple name is reported; otherwise, if an object is imported from another namespace, its fully-qualified access command is reported.

If the optional *pattern* is specified, then the reported names are compared using the rules of the "**string match**" command, and only matching names are reported. If the optional "**-class**" parameter is specified, this list is restricted to objects whose most-specific class is *className*. If the optional "**-isa**" parameter is specified, this list is further restricted to objects having the given *className* anywhere in their heritage.

KEYWORDS

class, object, object-oriented

NAME

itclvars − variables used by [incr Tcl]

DESCRIPTION

The following global variables are created and managed automatically by the **[incr Tcl]** library. Except where noted below, these variables should normally be treated as read-only by application-specific code and by users.

itcl::library

When an interpreter is created, **[incr Tcl]** initializes this variable to hold the name of a directory containing the system library of **[incr Tcl]** scripts. The initial value of **itcl::library** is set from the ITCL_LIBRARY environment variable if it exists, or from a compiled-in value otherwise.

itcl::patchLevel

When an interpreter is created, **[incr Tcl]** initializes this variable to hold the current patch level for **[incr Tcl]**. For example, the value "**2.0p1**" indicates **[incr Tcl]** version 2.0 with the first set of patches applied.

itcl::purist

When an interpreter is created containing Tcl/Tk and the **[incr Tcl]** namespace facility, this variable controls a "backward-compatibility" mode for widget access.

In vanilla Tcl/Tk, there is a single pool of commands, so the access command for a widget is the same as the window name. When a widget is created within a namespace, however, its access command is installed in that namespace, and should be accessed outside of the namespace using a qualified name. For example,

```
namespace foo {
  namespace bar {
    button .b -text "Testing"
  }
}
foo::bar::.b configure -background red
pack .b
```

Note that the window name "`.b`" is still used in conjunction with commands like **pack** and **destroy**. However, the access command for the widget (i.e., name that appears as the *first* argument on a command line) must be more specific.

The "**winfo command**" command can be used to query the fully-qualified access command for any widget, so one can write:

```
[winfo command .b] configure -background red
```

and this is good practice when writing library procedures. Also, in conjunction with the **bind** command, the "%q" field can be used in place of "%W" as the access command:

```
bind Button <Key-Return> {%q flash; %q invoke}
```

While this behavior makes sense from the standpoint of encapsulation, it causes problems with existing Tcl/Tk applications. Many existing applications are written with bindings that use "%W". Many library procedures assume that the window name is the access command.

The **itcl::purist** variable controls a backward-compatibility mode. By default, this variable is "0", and the window name can be used as an access command in any context. Whenever the **unknown** procedure stumbles across a widget name, it simply uses "**winfo command**" to determine the appropriate command name. If this variable is set to "1", this backward-compatibility mode is disabled. This gives better encapsulation, but using the window name as the access command may lead to "invalid command" errors.

itcl::version

When an interpreter is created, **[incr Tcl]** initializes this variable to hold the version number of the form *x.y*. Changes to *x* represent major changes with probable incompatibilities and changes to *y* represent small enhancements and bug fixes that retain backward compatibility.

KEYWORDS

itcl, variables

NAME

local − create an object local to a procedure

SYNOPSIS

local *className objName* ?*arg arg ...*?

DESCRIPTION

The **local** command creates an **[incr Tcl]** object that is local to the current call frame. When the call frame goes away, the object is automatically deleted. This command is useful for creating objects that are local to a procedure.

As a side effect, this command creates a variable named "`itcl-local-`*xxx*", where *xxx* is the name of the object that is created. This variable detects when the call frame is destroyed and automatically deletes the associated object.

EXAMPLE

In the following example, a simple "counter" object is used within the procedure "test". The counter is created as a local object, so it is automatically deleted each time the procedure exits. The **puts** statements included in the constructor/destructor show the object coming and going as the procedure is called.

```
class counter {
  private variable count 0
  constructor {} {
    puts "created: $this"
  }
  destructor {
    puts "deleted: $this"
  }

  method bump {{by 1}} {
    incr count $by
  }
  method get {} {
    return $count
  }
}

proc test {val} {
  local counter x
  for {set i 0} {$i < $val} {incr i} {
    x bump
  }
  return [x get]
}

set result [test 5]
puts "test: $result"

set result [test 10]
puts "test: $result"

puts "objects: [info objects]"
```

KEYWORDS

class, object, procedure

NAME

scope − capture the namespace context for a variable

SYNOPSIS

scope *name*

DESCRIPTION

Creates a scoped value for the specified *name*, which must be a variable name. If the *name* is an instance variable, then the scope command returns a string of the following form:

@itcl *object varName*

This is recognized in any context as an instance variable belonging to *object*. So with itcl3.0 and beyond, it is possible to use instance variables in conjunction with widgets. For example, if you have an object with a private variable `x`, and you can use `x` in conjunction with the `-textvariable` option of an entry widget. Before itcl3.0, only common variables could be used in this manner.

If the *name* is not an instance variable, then it must be a common variable or a global variable. In that case, the scope command returns the fully qualified name of the variable, e.g., `::foo::bar::x`.

If the *name* is not recognized as a variable, the scope command returns an error.

Ordinary variable names refer to variables in the global namespace. A scoped value captures a variable name together with its namespace context in a way that allows it to be referenced properly later. It is needed, for example, to wrap up variable names when a Tk widget is used within a namespace:

```
namespace foo {
  private variable mode 1

  radiobutton .rb1 -text "Mode #1"      -variable [scope mode] -value 1
  pack .rb1

  radiobutton .rb2 -text "Mode #2"      -variable [scope mode] -value 2
  pack .rb2
}
```

Radiobuttons `.rb1` and `.rb2` interact via the variable "mode" contained in the namespace "foo". The **scope** command guarantees this by returning the fully qualified variable name `::foo::mode`.

You should never use the `@itcl` syntax directly. For example, it is a bad idea to write code like this:

```
set {@itcl ::fred x} 3
puts "value = ${@itcl ::fred x}"
```

Instead, you should always use the scope command to generate the variable name dynamically. Then, you can pass that name to a widget or to any other bit of code in your program.

KEYWORDS

code, namespace, variable

NAME

Archetype − base class for all [incr Tk] mega-widgets

INHERITANCE

none

WIDGET-SPECIFIC OPTIONS

Name:	**clientData**
Class:	**ClientData**
Command-Line Switch:	**-clientdata**

This does not affect the widget operation in any way. It is simply a hook that clients can use to store a bit of data with each widget. This can come in handy when using widgets to build applications.

DESCRIPTION

The **Archetype** class is the basis for all **[incr Tk]** mega-widgets. It keeps track of component widgets and provides methods like "configure" and "cget" that are used to access the composite configuration options. Each component widget must be registered with the **Archetype** base class using the "**itk_component add**" method. When the component is registered, its configuration options are integrated into the composite option list. Configuring a composite option like "-background" causes all of the internal components to change their background color.

It is not used as a widget by itself, but is used as a base class for more specialized widgets. The **Widget** base class inherits from **Archetype**, and adds a Tk frame which acts as the "hull" for the mega-widget. The **Toplevel** base class inherits from **Archetype**, but adds a Tk toplevel which acts as the "hull".

Each derived class must invoke the **itk_initialize** *method within its constructor*, so that all options are properly integrated and initialized in the composite list.

PUBLIC METHODS

The following methods are provided to support the public interface of the mega-widget.

pathName **cget** *option*

Returns the current value of the configuration option given by *option*.

In this case, *option* refers to a composite configuration option for the mega-widget. Individual components integrate their own configuration options onto the composite list when they are registered by the "**itk_component add**" method.

pathName **component** ?*name*? ?*command arg arg* ...?

Used to query or access component widgets within a mega-widget. With no arguments, this returns a list of symbolic names for component widgets that are accessible in the current scope. The symbolic name for a component is established when it is registered by the "**itk_component add**" method. Note that component widgets obey any public/protected/private access restriction that is in force when the component is created.

If a symbolic *name* is specified, this method returns the window path name for that component.

Otherwise, the *command* and any remaining *arg* arguments are invoked as a method on the component with the symbolic name *name*. This provides a well-defined way of accessing internal components without relying on specific window path names, which are really details of the implementation.

pathName **configure** ?*option*? ?*value option value ...*?

Query or modify the configuration options of the widget. If no *option* is specified, returns a list describing all of the available options for *pathName* (see **Tk_ConfigureInfo** for information on the format of this list). If *option* is specified with no *value*, then the command returns a list describing the one named option (this list will be identical to the corresponding sublist of the value returned if no *option* is specified). If one or more *option−value* pairs are specified, then the command modifies the given widget option(s) to have the given value(s); in this case the command returns an empty string.

In this case, the *options* refer to composite configuration options for the mega-widget. Individual components integrate their own configuration options onto the composite list when they are registered by the "**itk_component add**" method.

PROTECTED METHODS

The following methods are used in derived classes as part of the implementation for a mega-widget.

itk_component add ?**-protected**? ?**-private**? ?**--**? *name createCmds* ?*optionCmds*?

Creates a component widget by executing the *createCmds* argument and registers the new component with the symbolic name *name*. The **-protected** and **-private** options can be used to keep the component hidden from the outside world. These options have a similar effect on component visibility as they have on class members.

The *createCmds* code can contain any number of commands, but it must return the window path name for the new component widget.

The *optionCmds* script contains commands that describe how the configuration options for the new component should be integrated into the composite list for the mega-widget. It can contain any of the following commands:

ignore *option* ?*option option ...*?

Removes one or more configuration *options* from the composite list. All options are ignored by default, so the **ignore** command is only used to negate the effect of a previous **keep** or **rename** command. This is useful, for example, when the some of the options added by the **usual** command should not apply to a particular component, and need to be ignored.

keep *option* ?*option option ...*?

Integrates one or more configuration *options* into the composite list, keeping the name the same. Whenever the mega-widget option is configured, the new value is also applied to the current component. Options like "-background" and "-cursor" are commonly found on the **keep** list.

rename *option switchName resourceName resourceClass*

Integrates the configuration *option* into the composite list with a different name. The option will be called *switchName* on the composite list. It will also be modified by setting values for *resourceName* and *resourceClass* in the X11 resource database. The "-highlightbackground" option is commonly renamed to "-background", so that when the mega-widget background changes, the background of the focus ring will change as well.

usual ?*tag*?

Finds the usual option-handling commands for the specified *tag* name and executes them. If the *tag* is not specified, then the widget class name is used as the *tag* name. The "usual" option-handling commands are registered via the **usual** command.

If the *optionCmds* script is not specified, the usual option-handling commands associated with the class of the component widget are used by default.

itk_component delete *name* ?*name name* ...?
Removes the component widget with the symbolic name *name* from the mega-widget. The component widget will still exist, but it will no longer be accessible as a component of the mega-widget. Also, any options associated with the component are removed from the composite option list.

Note that you can destroy a component using the **destroy** command, just as you would destroy any Tk widget. Components automatically detach themselves from their mega-widget parent when destroyed, so "**itk_component delete**" is rarely used.

itk_initialize ?*option value option value*...?
This method must be invoked within the constructor for each class in a mega-widget hierarchy. It makes sure that all options are properly integrated into the composite option list, and synchronizes all components to the initial option values. It is usually invoked near the bottom of the constructor, after all component widgets have been added.

If any *option*/*value* pairs are specified, they override settings determined from the X11 resource database. The arguments to the constructor are usually passed along to this method as follows:

```
class MyWidget {
  inherit Widget

  constructor {args} {
    .
    .
    .
    eval itk_initialize $args
  }
}
```

itk_option add *optName* ?*optName optName* ...?
Adds one or more options to the composite option list for a mega-widget. Here, *optName* can have one of the following forms:

component.*option*
Accesses an *option* belonging to a component with the symbolic name *component*. The *option* name is specified without a leading "-" sign.

className::*option*
Accesses an *option* defined by the "**itk_option define**" command in class *className*. The *option* name is specified without a leading "-" sign.

Options are normally integrated into the composite option list when a component widget is first created. This method can be used to add options at a later time. For example, the **Widget** and **Toplevel** base classes keep only the bare minimum options for their "hull" component: -background and -cursor. A derived class can override this decision, and add options that control the border of the "hull" component as well:

```
class MyWidget {
  inherit Widget

  constructor {args} {
    itk_option add hull.borderwidth hull.relief
```

```
        itk_component add label {
            label $itk_interior.l1 -text "Hello World!"
        }
        pack $itk_component(label)

        eval itk_initialize $args
    }
}
```

itk_option define *switchName resourceName resourceClass init* ?*config*?

This command is used at the level of the class definition to define a synthetic mega-widget option. Within the **configure** and **cget** methods, this option is referenced by *switchName*, which must start with a "-" sign. It can also be modified by setting values for *resourceName* and *resourceClass* in the X11 resource database. The *init* value string is used as a last resort to initialize the option if no other value can be used from an existing option, or queried from the X11 resource database. If any *config* code is specified, it is executed whenever the option is modified via the **configure** method. The *config* code can also be specified outside of the class definition via the **configbody** command.

In the following example, a synthetic "-background" option is added to the class, so that whenever the background changes, the new value is reported to standard output. Note that this synthetic option is integrated with the rest of the "-background" options that have been kept from component widgets:

```
class MyWidget {
    inherit Widget
    constructor {args} {
        itk_component add label {
            label $itk_interior.l1 -text "Hello World!"
        }
        pack $itk_component(label)

        eval itk_initialize $args
    }
    itk_option define -background background Background #d9d9d9 {
        puts "new background: $itk_option(-background)"
    }
}
```

itk_option remove *optName* ?*optName optName* ...?

Removes one or more options from the composite option list for a mega-widget. Here, *optName* can have one of the forms described above for the "**itk_option add**" command.

Options are normally integrated into the composite option list when a component widget is first created. This method can be used to remove options at a later time. For example, a derived class can override an option defined in a base class by removing and redefining the option:

```
class Base {
    inherit Widget
    constructor {args} {
        eval itk_initialize $args
    }
```

```
    itk_option define -foo foo Foo "" {
      puts "Base: $itk_option(-foo)"
    }
}

class Derived {
    inherit Base

    constructor {args} {
      itk_option remove Base::foo
      eval itk_initialize $args
    }
    itk_option define -foo foo Foo "" {
      puts "Derived: $itk_option(-foo)"
    }
}
```

Without the "**itk_option remove**" command, the code fragments for both of the "-foo" options would be executed each time the composite "-foo" option is configured. In the example above, the `Base::foo` option is suppressed in all Derived class widgets, so only the `Derived::foo` option remains.

PROTECTED VARIABLES

Derived classes can find useful information in the following protected variables.

itk_component(*name*)

The "itk_component" array returns the real window path name for a component widget with the symbolic name *name*. The same information can be queried using the **component** method, but accessing this array is faster and more convenient.

itk_interior

This variable contains the name of the window that acts as a parent for internal components. It is initialized to the name of the "hull" component provided by the **Widget** and **Toplevel** classes. Derived classes can override the initial setting to point to another interior window to be used for further-derived classes.

itk_option(*option*)

The "itk_option" array returns the current option value for the composite widget option named *option*. Here, the *option* name should include a leading "-" sign. The same information can be queried using the **cget** method, but accessing this array is faster and more convenient.

KEYWORDS

itk, Widget, Toplevel, mega-widget

NAME

Toplevel − base class for mega-widgets in a top-level window

INHERITANCE

itk::Archetype <- itk::Toplevel

STANDARD OPTIONS

background **cursor**

See the "options" manual entry for details on the standard options.

WIDGET-SPECIFIC OPTIONS

Name: **title**
Class: **Title**
Command-Line Switch: **-title**

Sets the title that the window manager displays in the title bar above the window. The default title is the null string.

DESCRIPTION

The **Toplevel** class inherits everything from the **Archetype** class, and adds a Tk toplevel called the "hull" component to represent the body of the mega-widget. The window class name for the hull is set to the most-specific class name for the mega-widget. The protected variable **itk_interior** contains the window path name for the "hull" component. Derived classes specialize this widget by packing other widget components into the hull.

Since the hull for the **Toplevel** class is implemented with a Tk toplevel, mega-widgets in the **Toplevel** class have their own toplevel window. This class is used to create dialog boxes and other pop-up windows.

COMPONENTS

Name: **hull**
Class: **Toplevel**

The "hull" component acts as the body for the entire mega-widget. Other components are packed into the hull to further specialize the widget.

EXAMPLE

The following example implements a **MessageInfo** mega-widget. It creates a pop-up message that the user can dismiss by pushing the "Dismiss" button.

```
option add *MessageInfo.title "Notice" widgetDefault

class MessageInfo {
  inherit itk::Toplevel

  constructor {args} {
    itk_component add dismiss {
      button $itk_interior.dismiss -text "Dismiss"          -command "destroy $itk_component(hull)"
    }
    pack $itk_component(dismiss) -side bottom -pady 4

    itk_component add separator {
      frame $itk_interior.sep -height 2 -borderwidth 1 -relief sunken
    }
    pack $itk_component(separator) -side bottom -fill x -padx 4
```

```
        itk_component add icon {
            label $itk_interior.icon -bitmap info
        }
        pack $itk_component(icon) -side left -padx 8 -pady 8

        itk_component add infoFrame {
            frame $itk_interior.info
        }
        pack $itk_component(infoFrame) -side left -expand yes          -fill both -padx 4 -pady 4

        itk_component add message {
            label $itk_interior.mesg -width 20
        } {
            usual
            rename -text -message message Text
        }
        pack $itk_component(message) -expand yes -fill both

        eval itk_initialize $args

        after idle [code $this centerOnScreen]
    }

    protected method centerOnScreen {} {
        update idletasks
        set wd [winfo reqwidth $itk_component(hull)]
        set ht [winfo reqheight $itk_component(hull)]
        set x [expr ([winfo screenwidth $itk_component(hull)]-$wd)/2]
        set y [expr ([winfo screenheight $itk_component(hull)]-$ht)/2]
        wm geometry $itk_component(hull) +$x+$y
    }
}

usual MessageInfo {
    keep -background -cursor -foreground -font
    keep -activebackground -activeforeground -disabledforeground
    keep -highlightcolor -highlightthickness
}

#
# EXAMPLE:  Create a notice window:
#
MessageInfo .m -message "File not found:\n/usr/local/bin/foo"
```

KEYWORDS

itk, Archetype, Widget, mega-widget

NAME

Widget − base class for mega-widgets within a frame

INHERITANCE

itk::Archetype <- itk::Widget

STANDARD OPTIONS

background **cursor**

See the "options" manual entry for details on the standard options.

DESCRIPTION

The **Widget** class inherits everything from the **Archetype** class, and adds a Tk frame called the "hull" component to represent the body of the mega-widget. The window class name for the hull is set to the most-specific class name for the mega-widget. The protected variable **itk_interior** contains the window path name for the "hull" component. Derived classes specialize this widget by packing other widget components into the hull.

Since the hull for the **Widget** class is implemented with a Tk frame, mega-widgets in the **Widget** class can be packed into other frames and toplevels.

COMPONENTS

Name: **hull**
Class: **Frame**

The "hull" component acts as the body for the entire mega-widget. Other components are packed into the hull to further specialize the widget.

EXAMPLE

The following example implements a simple **TextDisplay** mega-widget. It creates a read-only display of text with a text widget and a scrollbar.

```
option add *TextDisplay.wrap none widgetDefault
option add *TextDisplay.textBackground ivory widgetDefault
option add *TextDisplay.width 40 widgetDefault
option add *TextDisplay.height 10 widgetDefault

class TextDisplay {
  inherit itk::Widget

  constructor {args} {
    itk_component add text {
      text $itk_interior.info -state disabled          -yscrollcommand [code $itk_interior.sbar set]
    } {
      usual
      keep -tabs -wrap -width -height
      rename -background -textbackground textBackground Background
    }
    pack $itk_component(text) -side left -expand yes -fill both

    itk_component add scrollbar {
      scrollbar $itk_interior.sbar          -command [code $itk_interior.info yview]
    }
```

```
        pack $itk_component(scrollbar) -side right -fill y

        eval itk_initialize $args
    }

    public method display {info}
    public method append {info}
}

body TextDisplay::display {info} {
    $itk_component(text) configure -state normal
    $itk_component(text) delete 1.0 end
    $itk_component(text) insert 1.0 $info
    $itk_component(text) configure -state disabled
}

body TextDisplay::append {info} {
    $itk_component(text) configure -state normal
    $itk_component(text) insert end $info
    $itk_component(text) configure -state disabled
}

usual TextDisplay {
    keep -background -cursor -foreground -font
    keep -activebackground -activerelief
    keep -highlightcolor -highlightthickness
    keep -insertbackground -insertborderwidth -insertwidth
    keep -insertontime -insertofftime
    keep -selectbackground -selectborderwidth -selectforeground
    keep -textbackground -troughcolor
}

#
# EXAMPLE:  Display the /etc/passwd file
#
TextDisplay .file -background red
pack .file

.file display [exec cat /etc/passwd]
```

KEYWORDS

itk, Archetype, Widget, mega-widget

NAME

itk − framework for building mega-widgets in Tcl/Tk

DESCRIPTION

Mega-widgets are high-level widgets that are constructed using Tk widgets as component parts, usually without any C code. A fileselectionbox, for example, may have a few listboxes, some entry widgets and some control buttons. These individual widgets are put together in a way that makes them act like one big widget. A fileselectionbox mega-widget can be created with a command like:

```
fileselectionbox .fsb -background blue -foreground white
```

Once it has been created, it can be reconfigured with a command like:

```
.fsb configure -background green -foreground black
```

and all of its internal components will change color. Each mega-widget has a set of methods that can be used to manipulate it. For example, the current selection can be queried from a fileselectionbox like this:

```
set fileName [.fsb get]
```

In effect, a mega-widget looks and acts exactly like a Tk widget, but is considerably easier to implement.

[incr Tk] is a framework for building mega-widgets. It uses **[incr Tcl]** to support the object paradigm, and adds base classes which provide default widget behaviors.

All **[incr Tk]** widgets are derived from the **Archetype** base class. This class manages internal component widgets, and provides methods like "configure" and "cget" to access configuration options.

The **Widget** base class inherits everything from **Archetype**, and adds a Tk frame which acts as a container for the mega-widget. It is used to build mega-widgets that sit inside of other frames and toplevels. Derived classes create other internal components and pack them into the "hull" frame created by the **Widget** base class.

The **Toplevel** base class inherits everything from **Archetype**, but adds a Tk toplevel which acts as a container for the mega-widget. It is used to build mega-widgets, such as dialog boxes, that have their own toplevel window. Derived classes create other internal components and pack them into the "hull" toplevel created by the **Toplevel** base class.

[incr Widgets] LIBRARY

[incr Widgets] is a mega-widget library built using **[incr Tk]**. It can be used right out of the box, and contains more than 30 different widget classes, including:

- fileselectiondialog
- tabnotebook
- panedwindow
- combobox
- optionmenu
- scrolledlistbox
- scrolledframe
- messagedialog
- and many others...

The **catalog** demo in the "iwidgets/demos" directory shows all of the available widgets in action. Each widget class has its own man page describing the features available.

KEYWORDS

class, object, object-oriented, mega-widget

NAME

itkvars − variables used by [incr Tk]

DESCRIPTION

The following global variables are created and managed automatically by the **[incr Tk]** library. Except where noted below, these variables should normally be treated as read-only by application-specific code and by users.

itk::library

When an interpreter is created, **[incr Tk]** initializes this variable to hold the name of a directory containing the system library of **[incr Tk]** scripts. The initial value of **itk::library** is set from the ITK_LIBRARY environment variable if it exists, or from a compiled-in value otherwise.

When **[incr Tk]** is added to an interpreter, it executes the script "`init.itk`" in this directory. This script, in turn, looks for other script files with the name "`init.`*xxx*". Mega-widget libraries will be automatically initialized if they install a script named "`init.`*xxx*" in this directory, where "*xxx*" is the name of the mega-widget library. For example, the **[incr Widgets]** library installs the script "`init.iwidgets`" in this directory. This script establishes the "iwidgets" namespace, and sets up autoloading for all **[incr Widgets]** commands.

KEYWORDS

itcl, itk, variables

NAME

usual − access default option-handling commands for a mega-widget component

SYNOPSIS

usual ?*tag*? ?*commands*?

DESCRIPTION

The **usual** command is used outside of an **[incr Tcl]** class definition to define the usual set of option-handling commands for a component widget. Option-handling commands are used when a component is registered with the **Archetype** base class via the "**itk_component add**" method. They specify how the component's configuration options should be integrated into the composite option list for the mega-widget. Options can be kept, renamed, or ignored, as described in the **Archetype** man page.

It is tedious to include the same declarations again and again whenever components are added. The **usual** command allows a standard code fragment to be registered for each widget class, which is used by default to handle the options. All of the standard Tk widgets have **usual** declarations defined in the **[incr Tk]** library. Similar **usual** declarations should be created whenever a new mega-widget class is conceived. Only the most-generic options should be included in the **usual** declaration.

The *tag* name is usually the name of a widget class, which starts with a capital letter; however, any string registered here can be used later with the **usual** command described on the **Archetype** man page.

If the *commands* argument is specified, it is associated with the *tag* string, and can be accessed later via **itk_component add**.

If only the *tag* argument is specified, this command looks for an existing *tag* name and returns the commands associated with it. If there are no commands associated with *tag*, this command returns the null string.

If no arguments are specified, this command returns a list of all *tag* names previously registered.

EXAMPLE

Following is the **usual** declaration for the standard Tk button widget:

```
usual Button {
  keep -background -cursor -foreground -font
  keep -activebackground -activeforeground -disabledforeground
  keep -highlightcolor -highlightthickness
  rename -highlightbackground -background background Background
}
```

Only the options that would be common to all buttons in a single mega-widget are kept or renamed. Options like "-text" that would be unique to a particular button are ignored.

KEYWORDS

itk, Archetype, component, mega-widget

NAME

buttonbox − Create and manipulate a manager widget for buttons

SYNOPSIS

buttonbox *pathName* ?*options*?

INHERITANCE

itk::Widget <- buttonbox

STANDARD OPTIONS

background **cursor**

See the "options" manual entry for details on the standard options.

WIDGET-SPECIFIC OPTIONS

Name: **orient**
Class: **Orient**
Command-Line Switch: **-orient**

Orientation of the button box: **horizontal** or **vertical**. The default is horizontal.

Name: **padX**
Class: **PadX**
Command-Line Switch: **-padx**

Specifies a non-negative padding distance to leave between the button group and the outer edge of the button box in the x direction. The value may be given in any of the forms acceptable to **Tk_GetPixels**. The default is 5 pixels.

Name: **padY**
Class: **PadY**
Command-Line Switch: **-pady**

Specifies a non-negative padding distance to leave between the button group and the outer edge of the button box in the y direction. The value may be given in any of the forms acceptable to **Tk_GetPixels**. The default is 5 pixels.

DESCRIPTION

The **buttonbox** command creates a manager widget for controlling buttons. The button box also supports the display and invocation of a default button. The button box can be configured either horizontally or vertically.

METHODS

The **buttonbox** command creates a new Tcl command whose name is *pathName*. This command may be used to invoke various operations on the widget. It has the following general form:

pathName option ?*arg arg ...*?

Option and the *arg*s determine the exact behavior of the command.

Many of the widget commands for the buttonbox take as one argument an indicator of which button of the button box to operate on. These indicators are called *indexes* and allow reference and manipulation of buttons regardless of their current map state. buttonbox indexes may be specified in any of the following forms:

number Specifies the button numerically, where 0 corresponds to the left/top-most button of the button box.

end Indicates the right/bottom-most button of the button box.

default Indicates the current default button of the button box. This is the button with the default ring displayed.

pattern If the index doesn't satisfy one of the above forms then this form is used. *Pattern* is pattern-matched against the tag of each button in the button box, in order from left/top to right/left, until a matching entry is found. The rules of **Tcl_StringMatch** are used.

WIDGET-SPECIFIC METHODS

pathName **add** *tag args*

Add a button distinguished by *tag* to the end of the button box. If additional arguments are present they specify options to be applied to the button. See **PushButton** for information on the options available.

pathName **buttonconfigure** *index* ?*options*?

This command is similar to the **configure** command, except that it applies to the options for an individual button, whereas **configure** applies to the options for the button box as a whole. *Options* may have any of the values accepted by the **PushButton** command. If *options* are specified, options are modified as indicated in the command and the command returns an empty string. If no *options* are specified, returns a list describing the current options for entry *index* (see **Tk_ConfigureInfo** for information on the format of this list).

pathName **cget** *option*

Returns the current value of the configuration option given by *option*. *Option* may have any of the values accepted by the **buttonbox** command.

pathName **configure** ?*option*? ?*value option value ...*?

Query or modify the configuration options of the widget. If no *option* is specified, returns a list describing all of the available options for *pathName* (see **Tk_ConfigureInfo** for information on the format of this list). If *option* is specified with no *value*, then the command returns a list describing the one named option (this list will be identical to the corresponding sublist of the value returned if no *option* is specified). If one or more *option−value* pairs are specified, then the command modifies the given widget option(s) to have the given value(s); in this case the command returns an empty string. *Option* may have any of the values accepted by the **buttonbox** command.

pathName **default** *index*

Sets the default button to the button given by *index*. This causes the default ring to appear arround the specified button.

pathName **delete** *index*

Deletes the button given by *index* from the button box.

pathName **hide** *index*

Hides the button denoted by *index*. This doesn't remove the button permanently, just inhibits its display.

pathName **index** *index*

Returns the numerical index corresponding to *index*.

pathName **insert** *index tag* ?*option value option value ...*?

Same as the **add** command except that it inserts the new button just before the one given by *index*, instead of appending to the end of the button box. The *option*, and *value* arguments have the same interpretation as for the **add** widget command.

pathName **invoke** *?index?*

Invoke the command associated with a button. If no arguments are given then the current default button is invoked, otherwise the argument is expected to be a button *index*.

pathName **show** *index*
Display a previously hidden button denoted by *index*.

EXAMPLE

```
buttonbox .bb

.bb add Yes -text Yes -command "puts Yes"
.bb add No -text No -command "puts No"
.bb add Maybe -text Maybe -command "puts Maybe"
.bb default Yes

pack .bb -expand yes -fill both
```

AUTHOR

Bret A. Schuhmacher

Mark L. Ulferts

KEYWORDS

buttonbox, pushbutton, button, widget

NAME

calendar − Create and manipulate a monthly calendar

SYNOPSIS

calendar *pathName* ?*options*?

INHERITANCE

itk::Widget <- calendar

STANDARD OPTIONS

background **cursor** **foreground**

See the "options" manual entry for details on the standard options.

WIDGET-SPECIFIC OPTIONS

Name: **backwardImage**
Class: **Image**
Command-Line Switch: **-backwardimage**

Specifies a image to be displayed on the backwards calendar button. This image must have been created previously with the **image create** command. If none is specified, a default is provided.

Name: **buttonForeground**
Class: **Foreground**
Command-Line Switch: **-buttonforeground**

Specifies the foreground color of the forward and backward buttons in any of the forms acceptable to **Tk_GetColor**. The default color is blue.

Name: **command**
Class: **Command**
Command-Line Switch: **-command**

Specifies a Tcl script to executed upon selection of a date in the calendar. If the command script contains any **%** characters, then the script will not be executed directly. Instead, a new script will be generated by replacing each **%**, and the character following it, with information from the calendar. The replacement depends on the character following the **%**, as defined in the list below.

%d Replaced with the date selected in the format mm/dd/yyyy.

Name: **currentDateFont**
Class: **Font**
Command-Line Switch: **-currentdatefont**

Specifies the font used for the current date text in any of the forms acceptable to **Tk_GetFont**.

Name: **dateFont**
Class: **Font**
Command-Line Switch: **-datefont**

Specifies the font used for the days of the month text in any of the forms acceptable to **Tk_GetFont**.

Name: **dayFont**
Class: **Font**
Command-Line Switch: **-dayfont**

Specifies the font used for the days of the week text in any of the forms acceptable to **Tk_GetFont**.

Name: **days**
Class: **days**

Command-Line Switch: **-days**

Specifies a list of values to be used for the days of the week text to displayed above the days of the month. The default value is {Su Mo Tu We Th Fr Sa}.

Name: **forewardImage**
Class: **Image**
Command-Line Switch: **-forewardimage**

Specifies a image to be displayed on the forewards calendar button. This image must have been created previously with the **image create** command. If none is specified, a default is provided.

Name: **height**
Class: **Height**
Command-Line Switch: **-height**

Specifies a desired window height that the calendar widget should request from its geometry manager. The value may be specified in any of the forms acceptable to **Tk_GetPixels**. The default height is 165 pixels.

Name: **outline**
Class: **Outline**
Command-Line Switch: **-outline**

Specifies the outline color used to surround the days of the month text in any of the forms acceptable to **Tk_GetColor**. The default is the same color as the background.

Name: **selectColor**
Class: **Foreground**
Command-Line Switch: **-selectcolor**

Specifies the color of the ring displayed that distinguishes the currently selected date in any of the forms acceptable to **Tk_GetColor**. The default is red.

Name: **selectThickness**
Class: **SelectThickness**
Command-Line Switch: **-selectthickness**

Specifies the thickness of the ring displayed that distinguishes the currently selected date. The default is 3 pixels.

Name: **startday**
Class: **Day**
Command-Line Switch: **-startday**

Specifies the starting day for the week: **sunday**, **monday**, **tuesday**, **wednesday**, **thursday**, **friday**, or **saturday**. The default is sunday.

Name: **titleFont**
Class: **Font**
Command-Line Switch: **-titlefont**

Specifies the font used for the title text which consists of the month and year. The font may be given in any of the forms acceptable to **Tk_GetFont**.

Name: **weekdayBackground**
Class: **Background**
Command-Line Switch: **-weekdaybackground**

Specifies the background color for the weekdays which allows it to be visually distinguished from the weekend. The color may be given in any of the forms acceptable to **Tk_GetColor**. The default is the same as the background.

Name: **weekendBackground**
Class: **Background**
Command-Line Switch: **-weekendbackground**

Specifies the background color for the weekends which allows it to be visually distinguished from the weekdays. The color may be given in any of the forms acceptable to **Tk_GetColor**. The default is the same as the background.

Name: **width**
Class: **Width**
Command-Line Switch: **-width**

Specifies a desired window width that the calendar widget should request from its geometry manager. The value may be specified in any of the forms acceptable to **Tk_GetPixels**. The default width is 200 pixels.

DESCRIPTION

The **calendar** command creates a calendar widget for the selection of a date, displaying a single month at a time. Buttons exist on the top to change the month in effect turning the pages of a calendar. As a page is turned, the dates for the month are modified. Selection of a date visually marks that date. The selected value can be monitored via the -command option or just retrieved using the get command.

METHODS

The **calendar** command creates a new Tcl command whose name is *pathName*. This command may be used to invoke various operations on the widget. It has the following general form:

pathName option ?*arg arg ...*?

Option and the *arg*s determine the exact behavior of the command. The following commands are possible for calendar widgets:

WIDGET-SPECIFIC METHODS

pathName **cget** *option*

Returns the current value of the configuration option given by *option*. *Option* may have any of the values accepted by the **calendar** command.

pathName **configure** ?*option*? ?*value option value ...*?

Query or modify the configuration options of the widget. If no *option* is specified, returns a list describing all of the available options for *pathName* (see **Tk_ConfigureInfo** for information on the format of this list). If *option* is specified with no *value*, then the command returns a list describing the one named option (this list will be identical to the corresponding sublist of the value returned if no *option* is specified). If one or more *option−value* pairs are specified, then the command modifies the given widget option(s) to have the given value(s); in this case the command returns an empty string. *Option* may have any of the values accepted by the **calendar** command.

pathName **get** ?**format**?

Returns the currently selected date in a format of string or as an integer clock value using the **-string** and **-clicks** format options respectively. The default is by string. Reference the clock command for more information on obtaining dates and their formats.

pathName **select** *date*

Changes the currently selected date to the value specified which must be in the form of a date string, an integer clock value or as the keyword "now". Reference the clock command for more

information on obtaining dates and their formats. Note that selecting a date does not change the month being shown to that of the date given. This chore is left to the **showcommand.**

pathName **show** *date*

Changes the currently displayed date to be that of the date argument which must be in the form of a date string, an integer clock value or as the keyword "now". Reference the clock command for more information on obtaining dates and their formats.

COMPONENTS

Name: **forward**
Class: **Button**

The forward component provides the button on the upper right of the calendar that changes the month to be the next. See the "button" widget manual entry for details on the forward component item.

Name: **page**
Class: **Canvas**

The page component provides the canvas on which the title, days of the week, and days of the month appear. See the "canvas" widget manual entry for details on the page component item.

Name: **backward**
Class: **Button**

The backward component provides the button on the upper right of the calendar that changes the month to be the next. See the "button" widget manual entry for details on the backward component item.

EXAMPLE

```
proc selectCmd {date} {
  puts $date
}

calendar .c -command {selectCmd %d} -weekendbackground mistyrose \
                                    -weekdaybackground ghostwhite -outline black \
      -startday wednesday -days {We Th Fr Sa Su Mo Tu}
pack .c
```

AUTHOR

Mark L. Ulferts

Michael J. McLennan

KEYWORDS

calendar, widget

NAME

canvasprintbox − Create and manipulate a canvas print box widget

SYNOPSIS

canvasprintbox *pathName* ?*options*?

INHERITANCE

itk::Widget <- Canvasprintbox

STANDARD OPTIONS

activeBackground	**background**	**borderWidth**	**cursor**
foreground	**highlightBackground**	**highlightColor**	**highlightThickness**
insertBackground	**insertBorderWidth**	**insertOffTime**	**insertOnTime**
insertWidth	**relief**	**repeatDelay**	**repeatInterval**
selectBackground	**selectBorderWidth**	**selectForeground**	

See the "options" manual entry for details on the standard options.

ASSOCIATED OPTIONS

WIDGET-SPECIFIC OPTIONS

Name: **filename**
Class: **FileName**
Command-Line Switch: **-filename**

The file to write the postscript output to (Only when output is set to "file"). If posterizing is turned on and **hpagecnt** and/or **vpagecnt** is more than 1, x.y is appended to the filename where x is the horizontal page number and y the vertical page number.

Name: **hpagecnt**
Class: **PageCnt**
Command-Line Switch: **-hpagecnt**

Is used in combination with **posterize** to determine over how many pages the output should be distributed. This attribute specifies how many pages should be used horizontaly. Any change to this attribute will automatically update the "stamp". Defaults to 1.

Name: **orient**
Class: **Orient**
Command-Line Switch: **-orient**

Determines the orientation of the output to the printer (or file). It can take the value "portrait" or "landscape" (default). Changes to this attribute will be reflected immediately in the "stamp". Defaults to "landscape" but will be changed automaticaly to the value deemed appropiate for the current canvas. Setting this attribute when the canvasprintbox is first constructed (instead of using the "configure" method) will turn off the auto adjustment of this attribute.

Name: **output**
Class: **Output**
Command-Line Switch: **-output**

Specifies where the postscript output should go: to the printer or to a file. Can take on the values "printer" or "file". The corresponding entry-widget will reflect the contents of either the **printcmd** attribute or the **filename** attribute. Defaults to "printer".

Name: **pageSize**
Class: **PageSize**
Command-Line Switch: **-pagesize**

The pagesize the printer supports. Changes to this attribute will be reflected immediately in the

"stamp". Defaults to "a4".

Name: **posterize**
Class: **Posterize**
Command-Line Switch: **-posterize**

Indicates if posterizing is turned on or not. Posterizing the output means that it is possible to distribute the output over more than one page. This way it is possible to print a canvas/region which is larger than the specified pagesize without stretching. If used in combination with stretching it can be used to "blow up" the contents of a canvas to as large as size as you want (See attributes: hpagecnt and vpagecnt). Any change to this attribute will automatically update the "stamp". Defaults to 0.

Name: **printCmd**
Class: **PrintCmd**
Command-Line Switch: **-printcmd**

The command to execute when printing the postscript output. The command will get the postscript directed to its standard input (Only when output is set to "printer"). Defaults to "lpr".

Name: **printRegion**
Class: **PrintRegion**
Command-Line Switch: **-printregion**

A list of four coordinates specifying which part of the canvas to print. An empty list means that the canvas' entire **scrollregion** should be printed. Any change to this attribute will automatically update the "stamp". Defaults to an empty list.

Name: **stretch**
Class: **Stretch**
Command-Line Switch: **-stretch**

Determines if the output should be stretched to fill the page (as defined by the attribute pagesize) as large as possible. The aspect-ratio of the output will be retained and the output will never fall outside of the boundaries of the page. Defaults to 0 but will be changed automaticaly to the value deemed appropiate for the current canvas. Setting this attribute when the canvasprintbox is first constructed (instead of using the "configure" method) will turn off the auto adjustment of this attribute.

Name: **vPageCnt**
Class: **PageCnt**
Command-Line Switch: **-vpagecnt**

Is used in combination with "posterize" to determine over how many pages the output should be distributed. This attribute specifies how many pages should be used verticaly. Any change to this attribute will automatically update the "stamp". Defaults to 1.

DESCRIPTION

Implements a print box for printing the contents of a canvas widget to a printer or a file. It is possible to specify page orientation, the number of pages to print the image on and if the output should be stretched to fit the page. Options exist to control the appearance and actions of the widget.

METHODS

The **canvasprintbox** command creates a new Tcl command whose name is *pathName*. This command may be used to invoke various operations on the widget. It has the following general form:

pathName option ?*arg arg ...*?

Option and the *arg*s determine the exact behavior of the command. The following commands are possible for canvasprintbox widgets:

WIDGET-SPECIFIC METHODS

pathName **cget** *option*

Returns the current value of the configuration option given by *option*. *Option* may have any of the values accepted by the **canvasprintbox** command.

pathName **configure** ?*option*? ?*value option value ...*?

Query or modify the configuration options of the widget. If no *option* is specified, returns a list describing all of the available options for *pathName* (see **Tk_ConfigureInfo** for information on the format of this list). If *option* is specified with no *value*, then the command returns a list describing the one named option (this list will be identical to the corresponding sublist of the value returned if no *option* is specified). If one or more *option−value* pairs are specified, then the command modifies the given widget option(s) to have the given value(s); in this case the command returns an empty string. *Option* may have any of the values accepted by the **canvasprintbox** command.

pathName **getoutput**

Returns the value of the **printercmd** or **filename** option depending on the current setting of **output**.

pathName **print**

Perfrom the actual printing of the canvas using the current settings of all the attributes. Returns a boolean indicating wether the printing was successful or not.

pathName **refresh**

Retrieves the current value for all edit fields and updates the stamp accordingly. Is useful for Apply-buttons.

pathName **setcanvas** *canvas*

This is used to set the *canvas* that has to be printed. A stamp-sized copy will automatically be drawn to show how the output would look with the current settings.

pathName **stop**

Stops the drawing of the "stamp". I'm currently unable to detect when a Canvasprintbox gets destroyed or withdrawn. It's therefore advised that you perform a stop before you do something like that.

COMPONENTS

Name: **prtflentry**
Class: **Entry**

The prtflentry component is the entry field for user input of the **filename** or **printer** command (depending on the value of **output**).

Name: **hpcnt**
Class: **Entry**

The hpcnt component is the entry field for user input of the number of pages to use horizontaly when **posterize** is turned on.

Name: **vpcnt**
Class: **Entry**

The vpcnt component is the entry field for user input of the number of pages to use verticaly when **posterize** is turned on.

EXAMPLE

```
canvasprintbox .fsb -orient landscape -stretch 1
pack .fsb -padx 10 -pady 10 -fill both -expand yes
```

AUTHOR

Tako Schotanus

Tako.Schotanus@bouw.tno.nl

KEYWORDS

canvasprintbox, widget

NAME

canvasprintdialog − Create and manipulate a canvas print dialog widget

SYNOPSIS

canvasprintdialog *pathName* ?*options*?

INHERITANCE

itk::Toplevel <- Dialogshell <- Dialog <- Canvasprintdialog

STANDARD OPTIONS

activeBackground	**background**	**borderWidth**	**cursor**
foreground	**highlightBackground**	**highlightColor**	**highlightThickness**
insertBackground	**insertBorderWidth**	**insertOffTime**	**insertOnTime**
insertWidth	**relief**	**repeatDelay**	**repeatInterval**
selectBackground	**selectBorderWidth**	**selectForeground**	

See the "options" manual entry for details on the standard options.

ASSOCIATED OPTIONS

filename	**hpagecnt**	**orient**	**output**
pagesize	**posterize**	**printcmd**	**printregion**
vpagecnt			

See the "canvasprintbox" widget manual entry for details on the above associated options.

INHERITED OPTIONS

buttonBoxPadX	**buttonBoxPadY**	**buttonBoxPos**	**padX**
padY	**separator**	**thickness**	

See the "dialogshell" widget manual entry for details on the above inherited options.

master **modality**

See the "shell" widget manual entry for details on the above inherited options.

title

See the "Toplevel" widget manual entry for details on the above inherited options.

WIDGET-SPECIFIC OPTIONS

DESCRIPTION

The **canvasprintdialog** command creates a print dialog for printing the contents of a canvas widget to a printer or a file. It is possible to specify page orientation, the number of pages to print the image on and if the output should be stretched to fit the page.

METHODS

The **canvasprintdialog** command creates a new Tcl command whose name is *pathName*. This command may be used to invoke various operations on the widget. It has the following general form:

pathName option ?*arg arg ...*?

Option and the *arg*s determine the exact behavior of the command. The following commands are possible for canvasprintdialog widgets:

ASSOCIATED METHODS

getoutput	**setcanvas**	**refresh**	**print**

See the "canvasprintbox" class manual entry for details on the associated methods.

INHERITED METHODS

add	**buttonconfigure**	**default**	**hide**
insert	**invoke**	**show**	

See the "buttonbox" widget manual entry for details on the above inherited methods.

activate	**deactivate**

See the "dialogshell" widget manual entry for details on the above inherited methods.

WIDGET-SPECIFIC METHODS

pathName **cget** *option*

Returns the current value of the configuration option given by *option*. *Option* may have any of the values accepted by the **canvasprintdialog** command.

pathName **configure** ?*option*? ?*value option value ...*?

Query or modify the configuration options of the widget. If no *option* is specified, returns a list describing all of the available options for *pathName* (see **Tk_ConfigureInfo** for information on the format of this list). If *option* is specified with no *value*, then the command returns a list describing the one named option (this list will be identical to the corresponding sublist of the value returned if no *option* is specified). If one or more *option−value* pairs are specified, then the command modifies the given widget option(s) to have the given value(s); in this case the command returns an empty string. *Option* may have any of the values accepted by the **canvasprintdialog** command.

COMPONENTS

Name:	**cpb**
Class:	**Canvasprintbox**

The cpb component is the canvas print box for the canvas print dialog. See the "canvasprintbox" widget manual entry for details on the cpb component item.

EXAMPLE

```
canvasprintdialog .cpb
.cpb activate
```

AUTHOR

Tako Schotanus

Tako.Schotanus@bouw.tno.nl

KEYWORDS

canvasprintdialog, canvasprintbox, dialog, widget

NAME

checkbox − Create and manipulate a checkbox widget

SYNOPSIS

checkbox *pathName* ?*options*?

INHERITANCE

itk::Widget <- labeledframe <- checkbox

STANDARD OPTIONS

background	**borderWidth**	**cursor**	**disabledForeground**
foreground	**relief**	**selectColor**	

See the "options" manual entry for details on the standard options.

INHERITED OPTIONS

labelBitmap	**labelFont**	**labelImage**	**labelMargin**
labelPos	**labelText**	**labelVariable**	

See the "labeledframe" class manual entry for details on the inherited options.

WIDGET-SPECIFIC OPTIONS

Name: **command**
Class: **Command**
Command-Line Switch: **-command**

Specifies a Tcl command procedure to be evaluated following a change in the current check box selection.

DESCRIPTION

The **checkbox** command creates a check button box widget capable of adding, inserting, deleting, selecting, and configuring checkbuttons as well as obtaining the currently selected button.

METHODS

The **checkbox** command creates a new Tcl command whose name is *pathName*. This command may be used to invoke various operations on the widget. It has the following general form:

pathName option ?*arg arg ...*?

Option and the *arg*s determine the exact behavior of the command.

Many of the widget commands for the **checkbox** take as one argument an indicator of which checkbutton of the checkbox to operate on. These indicators are called *indexes* and allow reference and manipulation of checkbuttons. Checkbox indexes may be specified in any of the following forms:

number Specifies the checkbutton numerically, where 0 corresponds to the top checkbutton of the checkbox.

end Indicates the last checkbutton of the checkbox.

pattern If the index doesn't satisfy one of the above forms then this form is used. *Pattern* is pattern-matched against the tag of each checkbutton in the checkbox, in order from top to bottom, until a matching entry is found. The rules of **Tcl_StringMatch** are used.

WIDGET-SPECIFIC METHODS

pathName **add** *tag* ?*option value option value*?

Adds a new checkbutton to the checkbuttond window on the bottom. The command takes additional options which are passed on to the checkbutton as construction arguments. These include the standard Tk checkbutton options. The tag is returned.

pathName **buttonconfigure** *index* ?*options*?

This command is similar to the **configure** command, except that it applies to the options for an individual checkbutton, whereas **configure**applies to the options for the checkbox as a whole. *Options* may have any of the values accepted by the **add** widget command. If *options* are specified, options are modified as indicated in the command and the command returns an empty string. If no *options* are specified, returns a list describing the current options for entry *index* (see **Tk_ConfigureInfo** for information on the format of this list).

pathName **cget** *option*

Returns the current value of the configuration option given by *option*. *Option* may have any of the values accepted by the **checkbox** command.

pathName **configure** ?*option*? ?*value option value ...*?

Query or modify the configuration options of the widget. If no *option* is specified, returns a list describing all of the available options for *pathName* (see **Tk_ConfigureInfo** for information on the format of this list). If *option* is specified with no *value*, then the command returns a list describing the one named option (this list will be identical to the corresponding sublist of the value returned if no *option* is specified). If one or more *option−value* pairs are specified, then the command modifies the given widget option(s) to have the given value(s); in this case the command returns an empty string. *Option* may have any of the values accepted by the **checkbox** command.

pathName **delete** *index*

Deletes a specified checkbutton given an *index*.

pathName **deselect** *index*

Deselects a specified checkbutton given an *index*.

pathName **flash** *index*

Flashes a specified checkbutton given an *index*.

pathName **get** ?*index*?

Returns the tags of the currently selected checkbuttons or the selection status of specific checkbutton when given an index.

pathName **index** *index*

Returns the numerical index corresponding to index.

pathName **insert** *index tag* ?*option value option value ...*?

Same as the **add** command except that it inserts the new checkbutton just before the one given by *index*, instead of appending to the end of the checkbox. The *option*, and *value* arguments have the same interpretation as for the **add** widget command.

pathName **select** *index*

Selects a specified checkbutton given an *index*.

EXAMPLE

```
checkbox .cb -labeltext Styles
.cb add bold -text Bold
.cb add italic -text Italic
.cb add underline -text Underline
.cb select underline
```

```
pack .cb -padx 10 -pady 10 -fill both -expand yes
```

AUTHOR

John A. Tucker

KEYWORDS

checkbox, widget

NAME

combobox − Create and manipulate combination box widgets

SYNOPSIS

combobox *pathName* ?*options*?

INHERITANCE

itk::Widget <- LabeledWidget <- Entryfield <- Combobox

STANDARD OPTIONS

background	**borderWidth**	**cursor**	**justify**
exportSelection	**foreground**	**highlightColor**	**highlightThickness**
relief	**width**	**insertWidth**	**insertBackground**
insertOffTime	**insertOnTime**	**insertWidth**	**insertBorderWidth**
selectForeground	**selectBackground**		
selectBorderWidth	**textVariable**		

See the "options" manual entry for details on the standard options.

ASSOCIATED OPTIONS

hscrollmode	**textBackground**	**textFont**	**vscrollmode**

See the "scrolledlistbox" manual entry for details on the above inherited options.

show

See the "entry" manual entry for details on the above inherited option.

INHERITED OPTIONS

childSitePos	**command**	**fixed**	**focusCommand**
invalid	**textBackground**	**textFont**	**validate**

See the "entryfield" class manual entry for details on the inherited options.

labelBitmap	**labelFont**	**labelImage**	**labelMargin**
labelPos	**labelText**	**labelVariable**	

See the "labeledwidget" class manual entry for details on the inherited options.

WIDGET-SPECIFIC OPTIONS

Name: **arrowRelief**
Class: **Relief**
Command-Line Switch: **-arrowrelief**

Specifies the relief style to use for a dropdown Combobox's arrow button in a normal (not depressed) state. Acceptable values are **raised**, **sunken**, **flat**, **ridge**, and **groove**. Sunken is discouraged as this is the relief used to indicate a depressed state. This option has no effect on simple Comboboxes. The default is raised.

Name: **completion**
Class: **Completion**
Command-Line Switch: **-completion**

Boolean given in any of the forms acceptable to **Tcl_GetBoolean** which determines whether insertions into the entry field, whether from the keyboard or programmatically via the **insert** method, are automatically completed with the first matching item from the listbox. The default is true.

Name: **dropdown**
Class: **Dropdown**
Command-Line Switch: **-dropdown**

Boolean describing the Combobox layout style given in any of the forms acceptable to **Tcl_GetBoolean**. If true, the Combobox will be a dropdown style widget which displays an entry field and an arrow button which when activated will pop up a scrollable list of items. If false, a simple Combobox style will be used which has an entry field and a scrollable list beneath it which is always visible. Both styles allow an optional label for the entry field area. The default is true.

Name: **editable**
Class: **Editable**
Command-Line Switch: **-editable**

Boolean describing whether or not the text entry area is editable by the user. If true the user can add items to the combobox by entering text into the entry area and then pressing Return. If false, the list of items is non-editable and can only be changed by calling the insert or delete methods. (The value in the entry field can still be modified by selecting from the list.) Given in any of the forms acceptable to **Tcl_GetBoolean**. The default is true.

Name: **grab**
Class: **Grab**
Command-Line Switch: **-grab**

This option sets the grab scope for the appearance of the listbox in drop-down comboboxes. It can be either global or local. The default is local.

Name: **listHeight**
Class: **Height**
Command-Line Switch: **-listheight**

Height of the listbox specified in any of the forms acceptable to **Tk_GetPixels**. The default is 150 pixels.

Name: **margin**
Class: **Margin**
Command-Line Switch: **-margin**

Specifies the width in pixels between the entry component and the arrow button for a dropdown Combobox given in any of the forms acceptable to **Tk_GetPixels**. This option has no effect on a simple Combobox. The default is 1.

Name: **popupCursor**
Class: **Cursor**
Command-Line Switch: **-popupcursor**

Specifies the cursor to be used for dropdown style listboxes. The value may have any of the forms acceptable to **Tk_GetCursor**. The default is arrow.

Name: **selectionCommand**
Class: **SelectionCommand**
Command-Line Switch: **-selectioncommand**

Specifies a Tcl command procedure which is called when an item in the listbox area is selected. The item will be selected in the list, the listbox will be removed if it is a dropdown Combobox, and the selected item's text will be inserted into the entry field before the -selectioncommand proc is called. The default is {}.

Name: **state**
Class: **State**
Command-Line Switch: **-state**

Specifies the overall state of the Combobox megawidget. Can be either normal or disabled. If the Combobox is disabled, no text can be entered into the entry field, no selection can be made in the

listbox, and the arrowBtn component is disabled. The default is normal.

Name: **unique**
Class: **Unique**
Command-Line Switch: **-unique**

Boolean describing whether or not duplicate items are allowed in the combobox list. If true, then duplicates are not allowed to be inserted. If false, a duplicate entry causes selection of the item. Given in any of the forms acceptable to **Tcl_GetBoolean**. The default is true.

DESCRIPTION

The **combobox** command creates an enhanced entry field widget with an optional associated label and a scrollable list. When an item is selected in the list area of a Combobox, its value is then displayed in the entry field text area. Functionally similar to an Optionmenu, the Combobox adds (optional) list scrolling and (optional) item editing and inserting capabilities.

There are two basic styles of Comboboxes (determined by the -dropdown option): dropdown and simple. The dropdown style adds an arrow button to the right of the entry field which when activated will pop up (and down) the scrolled listbox beneath the entry field. The simple (non-dropdown) Combobox permanently displays the listbox beneath the entry field and has no arrow button. Either style allows an optional entry field label.

METHODS

The **combobox** command creates a new Tcl command whose name is *pathName*. This command may be used to invoke various operations on the widget. It has the following general form:

pathName option ?*arg arg ...*?

Option and the *arg*s determine the exact behavior of the command. The following commands are possible for Combobox widgets:

ASSOCIATED METHODS

icursor **scan**

See the "entry" manual entries for details on the above associated methods.

curselection **index** **see** **size**
xview **yview**

See the "listbox" manual entries for details on the above associated methods.

getcurselection **justify** **sort**

See the "scrolledlistbox" manual entries for details on the above associated methods.

WIDGET-SPECIFIC METHODS

pathName **cget** *option*

Returns the current value of the configuration option given by *option*. *Option* may have any of the values accepted by the **combobox** command.

pathName **clear** ?**component**?

Clears the contents from one or both components. Valid component values are **list**, or **entry**. With no component specified, both are cleared.

pathName **configure** ?*option*? ?*value option value ...*?

Query or modify the configuration options of the widget. If no *option* is specified, returns a list describing all of the available options for *pathName* (see **Tk_ConfigureInfo** for information on the format of this list). If *option* is specified with no *value*, then the command returns a list

describing the one named option (this list will be identical to the corresponding sublist of the value returned if no *option* is specified). If one or more *option−value* pairs are specified, then the command modifies the given widget option(s) to have the given value(s); in this case the command returns an empty string. *Option* may have any of the values accepted by the **combobox** command.

pathName **delete** *component first* ?*last*?
Delete one or more elements from a given component, **list** or **entry**. If a list item to be removed is currently selected (displayed in the entry field area), the entry field will be cleared.

pathName **get ?***index*?
With no arguments, returns the contents currently in the entry field area. With a single argument, returns the contents of the listbox item at the indicated index.

pathName **insert** *component index element* ?*element element ...*?
Insert one or more new elements into the given component, **list** or **entry**, just before the element given by *index*.

pathName **selection** *option first* ?*last*?
Adjust the selection within the listbox component and updates the contents of the entry field component to the value of the selected item. See the "listbox" manual entry for more details on parameter options.

COMPONENTS

Name: **entry**
Class: **Entry**

Text entry area where the current selection is displayed. If the Combobox is editable and its state is normal, the user can edit the contents of this item.

Name: **list**
Class: **Scrolledlistbox**

Scrollable list which stores all the items which the user can select from. For dropdown Comboboxes, this component is hidden until the user pops it up by pressing on the arrow button to the right of the entry component. For simple Comboboxes this component is always visible just beneath the entry component.

DEFAULT BINDINGS

The Combobox generally has the same bindings as its primary component items - the Scrolledlistbox and Entryfield. However it also adds these:

[1] Button-1 mouse press on the arrow key of a dropdown Combobox causes the list to be popped up. If the combobox is non-editable, a Button-1 press on the entry field area will also pop up the list.

[2] Button-1 mouse press anywhere on the display removes a dropdown listbox which has been popped up, unless the keypress is upon one of the Combobox scrollbars which scrolls the list. If it is pressed upon an item in the list area, that item will be selected before the list is removed.

[3] Button-3 mouse press on the arrow key of a dropdown Combobox causes the next item to be selected. Shift-Button-3 causes the previous item to be selected.

[4] Escape keypress removes a dropdown list which has been popped up.

[5] The <space> and <Return> keystrokes select the current item. They also remove the popped up list for dropdown comboboxes.

[6] Up and Down arrow keypresses from the entry field and arrow button component cause the previous and next items in the listbox to be selected respectively. Ctl-P and Ctl-N are similarly mapped for emacs emulation.

[7] Entry field and arrow button component Shift-Up and Shift-Down arrow keys pop up and down the list-box of a dropdown Combobox. The arrow button component also maps <Return> and <space> similarly.

EXAMPLE

```
proc selectCmd {} {
  puts stdout "[.cb2 getcurselection]"
}

#
# Non-editable Dropdown Combobox
#
combobox .cb1 -labeltext Month: \
   -selectioncommand {puts "selected: [.cb1 getcurselection]"} \
   -editable false -listheight 185 -popupcursor hand1
.cb1 insert list end Jan Feb Mar Apr May June Jul Aug Sept Oct Nov Dec

#
# Editable Dropdown Combobox
#
combobox .cb2 -labeltext "Operating System:" -selectioncommand selectCmd
.cb2 insert list end Linux HP-UX SunOS Solaris Irix
.cb2 insert entry end L

pack .cb1 -padx 10 -pady 10 -fill x
pack .cb2 -padx 10 -pady 10 -fill x
```

ORIGINAL AUTHOR

John S. Sigler

CURRENT MAINTAINER

Mitch Gorman (logain@erols.com)

KEYWORDS

combobox, entryfield, scrolledlistbox, itk::Widget, entry, listbox, widget, iwidgets

NAME

dateentry − Create and manipulate a dateentry widget

SYNOPSIS

dateentry *pathName* ?*options*?

INHERITANCE

itk::Widget <- LabeledWidget <- Datefield <- Dateentry

STANDARD OPTIONS

background	**borderWidth**	**cursor**	**exportSelection**
foreground	**highlightColor**	**highlightThickness**	**insertBackground**
justify	**relief**		

See the "options" manual entry for details on the standard options.

INHERITED OPTIONS

disabledForeground	**labelBitmap**	**labelFont**	**labelImage**
labelMargin	**labelPos**	**labelText**	**labelVariable**
state			

See the "labeledwidget" class manual entry for details on these inherited options.

command	**iq**	**state**	**textBackground**
textFont			

See the "datefield" class manual entry for details on these inherited options.

ASSOCIATED OPTIONS

backwardImage	**buttonForeground**	**command**	**currentDateFont**
dateFont	**dayFont**	**days**	**forwardImage**
outline	**selectColor**	**selectThickness**	**startDay**
titleFont	**weekdayBackground**	**weekendBackground**	

See the "calendar" manual entry for details on the associated options.

WIDGET-SPECIFIC OPTIONS

Name: **grab**
Class: **Grab**
Command-Line Switch: **-grab**

Specifies the grab level, **local** or **global**, to be obtained before bringing up the popup calendar. The default is global. For more information concerning grab levels, consult the documentation for Tk's **grab** command.

Name: **icon**
Class: **Icon**
Command-Line Switch: **-icon**

Specifies the calendar icon image to be used in the dateentry. This image must have been created previously with the **image create** command. Should one not be provided, then one will be generated, pixmap if possible, bitmap otherwise.

DESCRIPTION

The **dateentry** command creates a quicken style date entry field with a popup calendar by combining the datefield and calendar widgets together. This allows a user to enter the date via the keyboard or by using the mouse and selecting the calendar icon which brings up a popup calendar.

METHODS

The **dateentry** command creates a new Tcl command whose name is *pathName*. This command may be used to invoke various operations on the widget. It has the following general form:

pathName option ?*arg arg ...*?

Option and the *arg*s determine the exact behavior of the command. The following commands are possible for dateentry widgets:

INHERITED METHODS

get **isvalid** **show**

See the "datefield" manual entry for details on the associated methods.

WIDGET-SPECIFIC METHODS

pathName **cget** *option*

Returns the current value of the configuration option given by *option*. *Option* may have any of the values accepted by the **dateentry** command.

pathName **configure** ?*option*? ?*value option value ...*?

Query or modify the configuration options of the widget. If no *option* is specified, returns a list describing all of the available options for *pathName* (see **Tk_ConfigureInfo** for information on the format of this list). If *option* is specified with no *value*, then the command returns a list describing the one named option (this list will be identical to the corresponding sublist of the value returned if no *option* is specified). If one or more *option−value* pairs are specified, then the command modifies the given widget option(s) to have the given value(s); in this case the command returns an empty string. *Option* may have any of the values accepted by the **dateentry** command.

COMPONENTS

Name: **label**
Class: **Label**

The label component provides a label component to used to identify the date. See the "label" widget manual entry for details on the label component item.

Name: **iconbutton**
Class: **Label**

The iconbutton component provides a labelbutton component to act as a lightweight button displaying the calendar icon. Upon pressing the labelbutton, the calendar appears. See the "label" widget manual entry for details on the labelbutton component item.

Name: **date**
Class: **Entry**

The date component provides the entry field for date input and display. See the "entry" widget manual entry for details on the date component item.

EXAMPLE

```
dateentry .de
pack .de
```

AUTHOR

Mark L. Ulferts

KEYWORDS

dateentry, widget

NAME

datefield − Create and manipulate a date field widget

SYNOPSIS

datefield *pathName* ?*options*?

INHERITANCE

itk::Widget <- LabeledWidget <- datefield

STANDARD OPTIONS

background	**borderWidth**	**cursor**	**exportSelection**
foreground	**highlightColor**	**highlightThickness**	**insertBackground**
justify	**relief**		

See the "options" manual entry for details on the standard options.

INHERITED OPTIONS

disabledForeground	**labelBitmap**	**labelFont**	**labelImage**
labelMargin	**labelPos**	**labelText**	**labelVariable**
state			

See the "labeledwidget" class manual entry for details on the inherited options.

WIDGET-SPECIFIC OPTIONS

Name: **childSitePos**
Class: **Position**
Command-Line Switch: **-childsitepos**

Specifies the position of the child site in the date field: **n**, **s**, **e**, or **w**. The default is e.

Name: **command**
Class: **Command**
Command-Line Switch: **-command**

Specifies a Tcl command to be executed upon detection of a Return key press event.

Name: **iq**
Class: **Iq**
Command-Line Switch: **-iq**

Specifies the level of intelligence to be shown in the actions taken by the datefield during the processing of keypress events. Valid settings include **high**, **average**, and **low**. With a high iq,the date prevents the user from typing in an invalid date. For example, if the current date is 05/31/1997 and the user changes the month to 04, then the day will be instantly modified for them to be 30. In addition, leap years are fully taken into account. With average iq, the month is limited to the values of 01-12, but it is possible to type in an invalid day. A setting of low iq instructs the widget to do no validity checking at all during date entry. With both average and low iq levels, it is assumed that the validity will be determined at a later time using the date's **isvalid** command.

Name: **state**
Class: **State**
Command-Line Switch: **-state**

Specifies one of two states for the datefield: **normal** or **disabled**. If the datefield is disabled then input is not accepted. The default is normal.

Name: **textBackground**
Class: **Background**
Command-Line Switch: **-textbackground**

Background color for inside textual portion of the entry field. The value may be given in any of

the forms acceptable to **Tk_GetColor**.

Name: **textFont**
Class: **Font**
Command-Line Switch: **-textfont**

Name of font to use for display of text in datefield. The value may be given in any of the forms acceptable to **Tk_GetFont**.

DESCRIPTION

The **datefield** command creates an enhanced text entry widget for the purpose of date entry with various degrees of built-in intelligence.

METHODS

The **datefield** command creates a new Tcl command whose name is *pathName*. This command may be used to invoke various operations on the widget. It has the following general form:

pathName option ?*arg arg* ...?

Option and the *arg*s determine the exact behavior of the command. The following commands are possible for datefield widgets:

WIDGET-SPECIFIC METHODS

pathName **cget** *option*

Returns the current value of the configuration option given by *option*. *Option* may have any of the values accepted by the **datefield** command.

pathName **configure** ?*option*? ?*value option value* ...?

Query or modify the configuration options of the widget. If no *option* is specified, returns a list describing all of the available options for *pathName* (see **Tk_ConfigureInfo** for information on the format of this list). If *option* is specified with no *value*, then the command returns a list describing the one named option (this list will be identical to the corresponding sublist of the value returned if no *option* is specified). If one or more *option−value* pairs are specified, then the command modifies the given widget option(s) to have the given value(s); in this case the command returns an empty string. *Option* may have any of the values accepted by the **datefield** command.

pathName **get** ?**format**?

Returns the current contents of the datefield in a format of string or as an integer clock value using the **-string** and **-clicks** format options respectively. The default is by string. Reference the clock command for more information on obtaining dates and their formats.

pathName **isvalid**

Returns a boolean indication of the validity of the currently displayed date value. For example, 03/03/1960 is valid whereas 02/29/1997 is invalid.

pathName **show** *date*

Changes the currently displayed date to be that of the date argument. The date may be specified either as a string, an integer clock value or the keyword "now". Reference the clock command for more information on obtaining dates and their formats.

COMPONENTS

Name: **date**
Class: **Entry**

The date component provides the entry field for date input and display. See the "entry" widget manual entry for details on the date component item.

EXAMPLE

```
proc returnCmd {} {
  puts [.df get]
}

datefield .df -command returnCmd
pack .df -fill x -expand yes -padx 10 -pady 10
```

AUTHOR

Mark L. Ulferts

KEYWORDS

datefield, widget

NAME

dialog − Create and manipulate a dialog widget

SYNOPSIS

dialog *pathName* ?*options*?

INHERITANCE

itk::Toplevel <- Shell <- Dialogshell <- Dialog

STANDARD OPTIONS

background	**cursor**	**foreground**	

See the "options" manual entry for details on the standard options.

INHERITED OPTIONS

buttonBoxPadX	**buttonBoxPadY**	**buttonBoxPos**	**padX**
padY	**separator**	**thickness**	

See the "dialogshell" manual entry for details on the above inherited options.

height	**master**	**modality**	**width**

See the "shell" manual entry for details on the above inherited options.

title

See the "Toplevel" manual entry for details on the above inherited options.

DESCRIPTION

The **dialog** command creates a dialog box providing standard buttons and a child site for use in derived classes. The buttons include ok, apply, cancel, and help. Methods and Options exist to configure the buttons and their containing box.

METHODS

The **dialog** command creates a new Tcl command whose name is *pathName*. This command may be used to invoke various operations on the widget. It has the following general form:

pathName option ?*arg arg* ...?

Option and the *arg*s determine the exact behavior of the command. The following commands are possible for dialog widgets:

INHERITED METHODS

add	**buttonconfigure**	**default**	**hide**
index	**insert**	**invoke**	**show**

See the "buttonbox" manual entry for details on the above inherited methods.

childsite

See the "dialogshell" manual entry for details on the above inherited methods.

activate	**center**	**deactivate**

See the "shell" manual entry for details on the above inherited methods.

WIDGET-SPECIFIC METHODS

pathName **cget** *option*

Returns the current value of the configuration option given by *option*. *Option* may have any of the

values accepted by the **dialog** command.

pathName **configure** ?*option*? ?*value option value ...*?
Query or modify the configuration options of the widget. If no *option* is specified, returns a list describing all of the available options for *pathName* (see **Tk_ConfigureInfo** for information on the format of this list). If *option* is specified with no *value*, then the command returns a list describing the one named option (this list will be identical to the corresponding sublist of the value returned if no *option* is specified). If one or more *option−value* pairs are specified, then the command modifies the given widget option(s) to have the given value(s); in this case the command returns an empty string. *Option* may have any of the values accepted by the **dialog** command.

EXAMPLE

```
dialog .d -modality global
.d buttonconfigure OK -command {puts OK; .d deactivate 1}
.d buttonconfigure Apply -command {puts Apply}
.d buttonconfigure Cancel -command {puts Cancel; .d deactivate 0}
.d buttonconfigure Help -command {puts Help}

listbox [.d childsite].lb -relief sunken
pack [.d childsite].lb -expand yes -fill both

if {[.d activate]} {
   puts "Exit via OK button"
} else {
   puts "Exit via Cancel button"
}
```

AUTHOR

Mark L. Ulferts

Bret A. Schuhmacher

KEYWORDS

dialog, dialogshell, shell, widget

NAME

dialogshell − Create and manipulate a dialog shell widget

SYNOPSIS

dialogshell *pathName* ?*options*?

INHERITANCE

itk::Toplevel <- Shell <- Dialogshell

STANDARD OPTIONS

background **cursor** **foreground**

See the "options" manual entry for details on the standard options.

INHERITED OPTIONS

height **master** **modality** **width**

See the "shell" manual entry for details on the above inherited options.

title

See the "Toplevel" manual entry for details on the above inherited options.

WIDGET-SPECIFIC OPTIONS

Name: **buttonBoxPadX**
Class: **Pad**
Command-Line Switch: **-buttonboxpadx**

Specifies a non-negative padding distance to leave between the button group and the outer edge of the button box in the x direction. The value may be given in any of the forms accpetable to **Tk_GetPixels**. The default is 5 pixels.

Name: **buttonBoxPadY**
Class: **Pad**
Command-Line Switch: **-buttonboxpady**

Specifies a non-negative padding distance to leave between the button group and the outer edge of the button box in the y direction. The value may be given in any of the forms accpetable to **Tk_GetPixels**. The default is 5 pixels.

Name: **buttonBoxPos**
Class: **Position**
Command-Line Switch: **-buttonboxpos**

Attaches buttons to the given side of the dialog: **n**, **s**, **e** or **w**. The default is s.

Name: **padX**
Class: **Pad**
Command-Line Switch: **-padx**

Specifies a padding distance for the childsite in the X-direction in any of the forms acceptable to **Tk_GetPixels**. The default is 10.

Name: **padY**
Class: **Pad**
Command-Line Switch: **-pady**

Specifies a padding distance for the childsite in the Y-direction in any of the forms acceptable to **Tk_GetPixels**. The default is 10.

Name: **separator**
Class: **Separator**
Command-Line Switch: **-separator**

Specifies whether a line is drawn to separate the buttons from the dialog box contents in any of the forms acceptable to **Tcl_GetBoolean**. The default is true.

Name: **thickness**
Class: **Thickness**
Command-Line Switch: **-thickness**

Specifies the thickness of the separator in any of the forms acceptable to **Tk_GetPixels**. The default is 3 pixels.

DESCRIPTION

The **dialogshell** command creates a dialog shell which is a top level widget composed of a button box, separator, and child site area. The class also has methods to control button construction.

METHODS

The **dialogshell** command create a new Tcl command whose name is *pathName*. This command may be used to invoke various operations on the widget. It has the following general form:

pathName option ?*arg arg* ...?

Option and the *arg*s determine the exact behavior of the command. The following commands are possible for dialogshell widgets:

INHERITED METHODS

activate **center** **deactivate**

See the "shell" manual entry for details on the above inherited methods.

ASSOCIATED METHODS

add **buttonconfigure** **default** **delete**
hide **index** **insert** **invoke**
show

See the "buttonbox" manual entry for details on the associated methods.

WIDGET-SPECIFIC METHODS

pathName **cget** *option*

Returns the current value of the configuration option given by *option*. *Option* may have any of the values accepted by the **dialogshell** command.

pathName **childsite**

Returns the pathname of the child site widget.

pathName **configure** ?*option*? ?*value option value* ...?

Query or modify the configuration options of the widget. If no *option* is specified, returns a list describing all of the available options for *pathName* (see **Tk_ConfigureInfo** for information on the format of this list). If *option* is specified with no *value*, then the command returns a list describing the one named option (this list will be identical to the corresponding sublist of the value returned if no *option* is specified). If one or more *option−value* pairs are specified, then the command modifies the given widget option(s) to have the given value(s); in this case the command returns an empty string. *Option* may have any of the values accepted by the **dialogshell** command.

COMPONENTS

Name: **dschildsite**
Class: **frame**

The dschildsite component is the user child site for the dialog shell. See the "frame" widget manual entry for details on the dschildsite component item.

Name: **separator**
Class: **frame**

The separator component devides the area between the user child site and the button box. See the "frame" widget manual entry for details on the separator component item.

Name: **bbox**
Class: **ButtonBox**

The bbox component is the button box containing the buttons for the dialog shell. See the "ButtonBox" widget manual entry for details on the bbox component item.

EXAMPLE

```
dialogshell .ds -modality none

.ds add OK -text "OK"
.ds add Cancel -text "Cancel"
.ds default OK

.ds activate
```

AUTHOR

Mark L. Ulferts

KEYWORDS

dialogshell, dialog, shell, widget

NAME

disjointlistbox − Create and manipulate a disjointlistbox widget

SYNOPSIS

disjointlistbox *pathName* ?*options*?

INHERITANCE

itk::Widget <- Disjointlistbox

STANDARD OPTIONS

activeBackground selectBorderWidth selectForeground
activeForeground activeRelief background
borderWidth buttonPlacement clientData
cursor foreground highlightColor
highlightThickness disabledForeground elementBorderWidth

See the "options" manual entry for details on the standard options.

ASSOCIATED OPTIONS

lhsButtonLabel rhsButtonLabel

See the "button" widget manual entry for details on the above associated options.

labelFont lhsLabelText rhsLabelText

See the "label" widget manual entry for details on the above associated options.

jump troughColor

See the "scrollbar" widget class manual entry for details on the above associated options.

textBackground textFont
lhsItems rhsItems

See the "scrolledlistbox" widget manual entry for details on the above associated options.

WIDGET-SPECIFIC OPTIONS

Name: **buttonPlacement**
Class: **ButtonPlacement**
Command-Line Switch: **-buttonplacement**

Specifies the placement of the insertion and removal buttons relative to the scrolledlistbox widgets**n**, **bottom**, or **center**. The default is bottom.

Name: **lhsLabelText**
Class: **LabelText**
Command-Line Switch: **-lhslabeltext**

Specifies the text for the label of the lhs scrolledlistbox. The default is "Available".

Name: **rhsLabelText**
Class: **LabelText**
Command-Line Switch: **-rhslabeltext**

Specifies the text for the label of the rhs scrolledlistbox. The default is "Available".

Name: **lhsButtonLabel**
Class: **LabelText**
Command-Line Switch: **-lhsbuttonlabel**

Specifies the text for the button of the lhs scrolledlistbox. The default is "Insert >>".

Name:	**rhsButtonLabel**
Class:	**LabelText**
Command-Line Switch:	**-rhsbuttonlabel**

Specifies the text for the button of the rhs scrolledlistbox. The default is "<< Remove".

DESCRIPTION

The **disjointlistbox** command creates a disjoint pair of listboxs similar to the OSF/Motif "Book" printing dialog of the "FrameMaker" program. It is implementation constists of a two Scrolledlistboxs, 2 buttons, and 2 labels.

The disjoint behavior of this widget exists between the interaction of the two Scrolledlistboxes with one another. That is, a given instance of a Disjointlistbox will never exist, without the aid of a hack magician, which has Scrolledlistbox widgets with items in common. That means the relationship between the two is maintained similar to that of disjoint sets.

Users may transfer items between the two Listbox widgets using the the two buttons.

Options exists which include the ability to configure the "items" displayed by the 2 Scrolledlistboxes and to control the placement of the insertion and removal buttons.

METHODS

The **disjointlistbox** command creates a new Tcl command whose name is *pathName*. This command may be used to invoke various operations on the widget. It has the following general form:

pathName option ?*arg arg ...*?

Option and the *arg*s determine the exact behavior of the command. The following commands are possible for disjointlistbox widgets:

WIDGET-SPECIFIC METHODS

pathName **cget** *option*

Returns the current value of the configuration option given by *option*. *Option* may have any of the values accepted by the **disjointlistbox** command.

pathName **configure** ?*option*? ?*value option value ...*?

Query or modify the configuration options of the widget. If no *option* is specified, returns a list describing all of the available options for *pathName* (see **Tk_ConfigureInfo** for information on the format of this list). If *option* is specified with no *value*, then the command returns a list describing the one named option (this list will be identical to the corresponding sublist of the value returned if no *option* is specified). If one or more *option−value* pairs are specified, then the command modifies the given widget option(s) to have the given value(s); in this case the command returns an empty string. *Option* may have any of the values accepted by the **disjointlistbox** command.

pathName **setlhs**

Set the current contents of the left-most Scrolledlistbox with the input list of items. Removes all (if any) items from the right-most Scrolledlistbox which exist in the input list option to maintain the disjoint property between the two

pathName **setrhs**

Set the current contents of the right-most Scrolledlistbox with the input list of items. Removes all (if any) items from the left-most Scrolledlistbox which exist in the input list option to maintain the

disjoint property between the two

pathName **getlhs**
Returns the current contents of the left-most Scrolledlistbox

pathName **getrhs**
Returns the current contents of the right-most Scrolledlistbox

pathName **insertlhs**
Add the input list of items to the current contents of the left-most Scrolledlistbox. Removes all (if any) items from the right-most Scrolledlistbox which exist in the input list option to maintain the disjoint property between the two

pathName **insertrhs**
Add the input list of items to the current contents of the right-most Scrolledlistbox. Removes all (if any) items from the left-most Scrolledlistbox which exist in the input list option to maintain the disjoint property between the two.

Name: **lhs**
Class: **Scrolledlistbox**

The lhs component is the scrolledlistbox for the rhs button. See the "scrolledlistbox" widget manual entry for details on the lhs component item.

Name: **rhs**
Class: **Scrolledlistbox**

The rhs component is the scrolledlistbox for the rhs button. See the "scrolledlistbox" widget manual entry for details on the rhs component item.

Name: **lhsbutton**
Class: **utton**

The lhsbutton component is the button for users to remove selected items from the lhs Scrolledlistbox. See the "button" widget manual entry for details on the lhs button component.

Name: **rhsbutton**
Class: **Button**

The rhsbutton component is the button for users to remove selected items from the rhs Scrolledlistbox. See the "button" widget manual entry for details on the rhs button component.

Name: **lhsCount**
Class: **Label**

The lhsCount component is the label for displaying a count of the current items in the Scrolledlistbox. See the "Label" widget manual entry for details on the lhsCount label component.

Name: **rhsCount**
Class: **Label**

The rhsCount component is the label for displaying a count of the current items in the Scrolledlistbox. See the "Label" widget manual entry for details on the rhsCount label component.

EXAMPLE

```
disjointlistbox .dlb
pack .dlb -padx 10 -pady 10 -fill both -expand yes
```

AUTHOR(S)

John A. Tucker

Anthony Parent

KEYWORDS

disjointlistbox, widget

NAME

entryfield − Create and manipulate a entry field widget

SYNOPSIS

entryfield *pathName* ?*options*?

INHERITANCE

itk::Widget <- LabeledWidget <- entryfield

STANDARD OPTIONS

background	**borderWidth**	**cursor**	**exportSelection**
foreground	**highlightColor**	**highlightThickness**	**insertBackground**
insertBorderWidth	**insertOffTime**	**insertOnTime**	**insertWidth**
justify	**relief**	**selectBackground**	**selectBorderWidth**
selectForeground	**textVariable**	**width**	

See the "options" manual entry for details on the standard options.

ASSOCIATED OPTIONS

show	**state**

See the "entry" manual entry for details on the associated options.

INHERITED OPTIONS

disabledForeground	**labelBitmap**	**labelFont**	**labelImage**
labelMargin	**labelPos**	**labelText**	**labelVariable**
state			

See the "labeledwidget" class manual entry for details on the inherited options.

WIDGET-SPECIFIC OPTIONS

Name: **childSitePos**
Class: **Position**
Command-Line Switch: **-childsitepos**

Specifies the position of the child site in the entry field: **n**, **s**, **e**, or **w**. The default is e.

Name: **command**
Class: **Command**
Command-Line Switch: **-command**

Specifies a Tcl command to be executed upon detection of a Return key press event.

Name: **fixed**
Class: **Fixed**
Command-Line Switch: **-fixed**

Restrict entry to the specified number of chars. A value of 0, which is the default, denotes no limit. The value is the maximum number of chars the user may type into the field, regardles of field width. For example, if the field width is set to 20 and the fixed value is 10, the user will only be able to type 10 characters into the field which is 20 characters long.

Name: **focusCommand**
Class: **Command**
Command-Line Switch: **-focuscommand**

Specifies a Tcl command to be executed upon reception of focus.

Name: **invalid**
Class: **Command**
Command-Line Switch: **-invalid**

Specifies a Tcl command to be executed upon determination of invalid input. The default is bell.

Name: **textBackground**
Class: **Background**
Command-Line Switch: **-textbackground**

Background color for inside textual portion of the entry field. The value may be given in any of the forms acceptable to **Tk_GetColor**.

Name: **textFont**
Class: **Font**
Command-Line Switch: **-textfont**

Name of font to use for display of text in entryfield. The value may be given in any of the forms acceptable to **Tk_GetFont**.

Name: **validate**
Class: **Command**
Command-Line Switch: **-validate**

The validate option allows specification of a validation mechanism. Standard character validation such as **numeric**, **alphabetic**, **integer**, **hexidecimal**, **real**, and **alphanumeric** can be handled through the use of keywords. Should more extensive validation be necessary, the value may contain the name of a command script. The script should return a boolean value. True for valid, false for invalid. If false is returned, then the procedure associated with the invalid option will be invoked. If the validation script contains any **%** characters, then the script will not be executed directly. Instead, a new script will be generated by replacing each **%**, and the character following it, with information from the entryfield. The replacement depends on the character following the **%**, as defined in the list below.

%c Replaced with the current input character.

%P Replaced with the contents of the entryfield modified to include the latest keystoke. This is equivalent to peeking at the future contents, enabling rejection prior to the update.

%S Replaced with the current contents of the entryfield prior to the latest keystroke being added.

%W Replaced with the entryfield widget pathname.

DESCRIPTION

The **entryfield** command creates an enhanced text entry widget with an optional associated label. Addtional options support validation and establishing a upper limit on the number of characters which may be entered in the field.

METHODS

The **entryfield** command creates a new Tcl command whose name is *pathName*. This command may be used to invoke various operations on the widget. It has the following general form:

pathName option ?*arg arg ...*?

Option and the *arg*s determine the exact behavior of the command. The following commands are possible for entryfield widgets:

ASSOCIATED METHODS

delete	**get**	**icursor**	**index**
insert	**scan**	**selection**	**xview**

See the "entry" manual entry for details on the associated methods.

WIDGET-SPECIFIC METHODS

pathName **cget** *option*

Returns the current value of the configuration option given by *option*. *Option* may have any of the values accepted by the **entryfield** command.

pathName **childsite**

Returns the path name of the child site.

pathName **clear**

Clear entry widget

pathName **configure** ?*option*? ?*value option value ...*?

Query or modify the configuration options of the widget. If no *option* is specified, returns a list describing all of the available options for *pathName* (see **Tk_ConfigureInfo** for information on the format of this list). If *option* is specified with no *value*, then the command returns a list describing the one named option (this list will be identical to the corresponding sublist of the value returned if no *option* is specified). If one or more *option−value* pairs are specified, then the command modifies the given widget option(s) to have the given value(s); in this case the command returns an empty string. *Option* may have any of the values accepted by the **entryfield** command.

COMPONENTS

Name: **efchildsite**
Class: **frame**

The efchildsite component is the user child site for the entry field. See the "frame" widget manual entry for details on the efchildsite component item.

Name: **entry**
Class: **entry**

The entry component provides the entry field for user text input and display. See the "entry" widget manual entry for details on the entry component item.

EXAMPLE

```
option add *textBackground white

proc returnCmd {} {
  puts stdout "Return Pressed"
}

proc invalidCmd {} {
  puts stdout "Alphabetic contents invalid"
}

entryfield .ef -command returnCmd

entryfield .fef -labeltext "Fixed:" -fixed 10 -width 12

entryfield .nef -labeltext "Numeric:" -validate numeric -width 12
```

```
entryfield .aef -labeltext "Alphabetic:" \
  -validate alphabetic -width 12 -invalid invalidCmd

entryfield .pef -labeltext "Password:" \
  -show 267 -width 12 -command returnCmd

Labeledwidget::alignlabels .ef .fef .nef .aef .pef

pack .ef -fill x -expand yes -padx 10 -pady 5
pack .fef -fill x -expand yes -padx 10 -pady 5
pack .nef -fill x -expand yes -padx 10 -pady 5
pack .aef -fill x -expand yes -padx 10 -pady 5
pack .pef -fill x -expand yes -padx 10 -pady 5
```

AUTHOR

Sue Yockey

Mark L. Ulferts

KEYWORDS

entryfield, widget

NAME

extfileselectionbox − Create and manipulate a file selection box widget

SYNOPSIS

extfileselectionbox *pathName* ?*options*?

INHERITANCE

itk::Widget <- Extfileselectionbox

STANDARD OPTIONS

activeBackground	**background**	**borderWidth**	**cursor**
foreground	**highlightColor**	**highlightThickness**	**insertBackground**
insertBorderWidth	**insertOffTime**	**insertOnTime**	**insertWidth**
selectBackground	**selectBorderWidth**	**selectForeground**	

See the "options" manual entry for details on the standard options.

ASSOCIATED OPTIONS

popupCursor	**textBackground**	**textFont**

See the "combobox" widget manual entry for details on the above associated options.

labelFont

See the "labeledwidget" widget manual entry for details on the above associated options.

sashCursor

See the "panedwindow" widget manual entry for details on the above associated options.

activeRelief	**elementBorderWidth**	**jump**	**troughColor**

See the "scrollbar" widget class manual entry for details on the above associated options.

textBackground	**textFont**

See the "scrolledlistbox" widget manual entry for details on the above associated options.

WIDGET-SPECIFIC OPTIONS

Name: **childSitePos**
Class: **Position**
Command-Line Switch: **-childsitepos**

Specifies the position of the child site in the extended fileselection box: **n**, **s**, **e**, **w**, **top**, or **bottom**. The default is s.

Name: **directory**
Class: **Directory**
Command-Line Switch: **-directory**

Specifies the initial default directory. The default is the present working directory.

Name: **dirSearchCommand**
Class: **Command**
Command-Line Switch: **-dirsearchcommand**

Specifies a Tcl command to be executed to perform a directory search. The command will receive the current working directory and filter mask as arguments. The command should return a list of files which will be placed into the directory list.

Name: **dirsLabel**
Class: **Text**
Command-Line Switch: **-dirslabel**

Specifies the text of the label for the directory list. The default is "Directories".

Name: **dirsOn**
Class: **DirsOn**
Command-Line Switch: **-dirson**

Specifies whether or not to display the directory list. The value may be given in any of the forms acceptable to **Tcl_GetBoolean**. The default is true.

Name: **fileSearchCommand**
Class: **Command**
Command-Line Switch: **-filesearchcommand**

Specifies a Tcl command to be executed to perform a file search. The command will receive the current working directory and filter mask as arguments. The command should return a list of files which will be placed into the file list.

Name: **filesLabel**
Class: **Text**
Command-Line Switch: **-fileslabel**

Specifies the text of the label for the files list. The default is "Files".

Name: **filesOn**
Class: **FilesOn**
Command-Line Switch: **-fileson**

Specifies whether or not to display the files list. The value may be given in any of the forms acceptable to **Tcl_GetBoolean**. The default is true.

Name: **fileType**
Class: **FileType**
Command-Line Switch: **-filetype**

Specify the type of files which may appear in the file list: **regular**, **directory**, or **any**. The default is regular.

Name: **filterCommand**
Class: **Command**
Command-Line Switch: **-filtercommand**

Specifies a Tcl command to be executed upon hitting the Return key in the filter combobox widget.

Name: **filterLabel**
Class: **Text**
Command-Line Switch: **-filterlabel**

Specifies the text of the label for the filter combobox. The default is "Filter".

Name: **filterOn**
Class: **FilterOn**
Command-Line Switch: **-filteron**

Specifies whether or not to display the filter combobox. The value may be given in any of the forms acceptable to **Tcl_GetBoolean**. The default is true.

Name: **height**
Class: **Height**
Command-Line Switch: **-height**

Specifies the height of the selection box. The value may be specified in any of the forms acceptable to Tk_GetPixels. The default is 300 pixels.

Name: **invalid**
Class: **Command**
Command-Line Switch: **-invalid**

Command to be executed should the filter contents be proven invalid. The default is {bell}.

Name: **mask**
Class: **Mask**
Command-Line Switch: **-mask**

Specifies the initial file mask string. The default is "*".

Name: **noMatchString**
Class: **NoMatchString**
Command-Line Switch: **-nomatchstring**

Specifies the string to be displayed in the files list should no files match the mask. The default is "".

Name: **selectDirCommand**
Class: **Command**
Command-Line Switch: **-selectdirommand**

Specifies a Tcl command to be executed following selection of a directory in the directory list.

Name: **selectFileCommand**
Class: **Command**
Command-Line Switch: **-selectfileommand**

Specifies a Tcl command to be executed following selection of a file in the files list.

Name: **selectionCommand**
Class: **Command**
Command-Line Switch: **-selectioncommand**

Specifies a Tcl command to be executed upon hitting the Return key in the selection combobox widget.

Name: **selectionLabel**
Class: **Text**
Command-Line Switch: **-selectionlabel**

Specifies the text of the label for the selection combobox. The default is "Selection".

Name: **selectionOn**
Class: **SelectionOn**
Command-Line Switch: **-selectionon**

Specifies whether or not to display the selection combobox. The value may be given in any of the forms acceptable to **Tcl_GetBoolean**. The default is true.

Name: **width**
Class: **Width**
Command-Line Switch: **-width**

Specifies the width of the selection box. The value may be specified in any of the forms acceptable to Tk_GetPixels. The default is 350 pixels.

DESCRIPTION

The **extfileselectionbox** command creates an extended file selection box which is slightly different than the fileselectionbox widget. The differences are mostly cosmetic in that the listboxes are within a panedwindow and the entryfields for the filter and selection have been replaced by comboboxes. Other than that the interface is practically the same.

METHODS

The **extfileselectionbox** command creates a new Tcl command whose name is *pathName*. This command may be used to invoke various operations on the widget. It has the following general form:

pathName option ?*arg arg ...*?

Option and the *arg*s determine the exact behavior of the command. The following commands are possible for extfileselectionbox widgets:

WIDGET-SPECIFIC METHODS

pathName **cget** *option*

Returns the current value of the configuration option given by *option*. *Option* may have any of the values accepted by the **extfileselectionbox** command.

pathName **childsite**

Returns the child site widget path name.

pathName **configure** ?*option*? ?*value option value ...*?

Query or modify the configuration options of the widget. If no *option* is specified, returns a list describing all of the available options for *pathName* (see **Tk_ConfigureInfo** for information on the format of this list). If *option* is specified with no *value*, then the command returns a list describing the one named option (this list will be identical to the corresponding sublist of the value returned if no *option* is specified). If one or more *option−value* pairs are specified, then the command modifies the given widget option(s) to have the given value(s); in this case the command returns an empty string. *Option* may have any of the values accepted by the **extfileselectionbox** command.

pathName **filter**

Update the current contents of the extended file selection box based on the current filter combobox value.

pathName **get**

Returns the current value of the selection combobox widget.

COMPONENTS

Name: **dirs**
Class: **Scrolledlistbox**

The dirs component is the directory list box for the extended fileselection box. See the "scrolledlistbox" widget manual entry for details on the dirs component item.

Name: **files**
Class: **Scrolledlistbox**

The files component is the file list box for the extended fileselection box. See the "scrolledlistbox" widget manual entry for details on the files component item.

Name: **filter**
Class: **Combobox**

The filter component is the field for user input of the filter value. See the "combobox" widget manual entry for details on the filter component item.

Name: **selection**
Class: **Combobox**

The selection component is the field for user input of the currently selected file value. See the "combobox" widget manual entry for details on the selection component item.

EXAMPLE

```
extfileselectionbox .fsb
pack .fsb -padx 10 -pady 10 -fill both -expand yes
```

AUTHOR(S)

Mark L. Ulferts

Anthony Parent

KEYWORDS

extfileselectionbox, widget

NAME

extfileselectiondialog − Create and manipulate a file selection dialog widget

SYNOPSIS

extfileselectiondialog *pathName* ?*options*?

INHERITANCE

itk::Toplevel <- Shell <- Dialogshell <- Dialog <- Extfileselectiondialog

STANDARD OPTIONS

activeBackground	**background**	**borderWidth**	**cursor**
foreground	**highlightColor**	**highlightThickness**	**insertBackground**
insertBorderWidth	**insertOffTime**	**insertOnTime**	**insertWidth**
selectBackground	**selectBorderWidth**	**selectForeground**	

See the "options" manual entry for details on the standard options.

ASSOCIATED OPTIONS

popupCursor	**textBackground**	**textFont**

See the "combobox" widget manual entry for details on the above associated options.

childSitePos	**directory**	**dirsLabel**	**dirSearchCommand**
dirsOn	**filesLabel**	**filesLabelOn**	**fileSearchCommand**
filesOn	**fileType**	**filterLabel**	**filterOn**
invalid	**mask**	**noMatchString**	**selectionLabel**
selectionOn			

See the "extfileselectionbox" widget manual entry for details on the above associated options.

labelFont

See the "labeledwidget" widget manual entry for details on the above associated options.

sashCursor

See the "panedwindow" widget manual entry for details on the above associated options.

labelFont

See the "labeledwidget" widget manual entry for details on the above associated options.

activeRelief	**elementBorderWidth**	**jump**	**troughColor**

See the "scrollbar" widget class manual entry for details on the above associated options.

textBackground	**textFont**

See the "scrolledlistbox" widget manual entry for details on the above associated options.

INHERITED OPTIONS

buttonBoxPadX	**buttonBoxPadY**	**buttonBoxPos**	**padX**
padY	**separator**	**thickness**	

See the "dialogshell" widget manual entry for details on the above inherited options.

height	**master**	**modality**	**width**

See the "shell" widget manual entry for details on the above inherited options.

title

See the "Toplevel" widget manual entry for details on the above inherited options.

DESCRIPTION

The **extfileselectiondialog** command creates an extended file selection dialog which is slightly different than the fileselectiondialog widget. The differences are mostly cosmetic in that the listboxes are within a panedwindow and the entryfields for the filter and selection have been replaced by comboboxes. Other than that the interface is practically the same.

METHODS

The **extfileselectiondialog** command creates a new Tcl command whose name is *pathName*. This command may be used to invoke various operations on the widget. It has the following general form:

pathName option ?*arg arg ...*?

Option and the *arg*s determine the exact behavior of the command. The following commands are possible for extfileselectiondialog widgets:

ASSOCIATED METHODS

get **childsite** **filter**

See the "fileselectionbox" class manual entry for details on the associated methods.

INHERITED METHODS

add **buttonconfigure** **default** **hide**
insert **invoke** **show**

See the "buttonbox" widget manual entry for details on the above inherited methods.

activate **center** **deactivate**

See the "shell" widget manual entry for details on the above inherited methods.

WIDGET-SPECIFIC METHODS

pathName **cget** *option*

Returns the current value of the configuration option given by *option*. *Option* may have any of the values accepted by the **extfileselectiondialog** command.

pathName **configure** ?*option*? ?*value option value ...*?

Query or modify the configuration options of the widget. If no *option* is specified, returns a list describing all of the available options for *pathName* (see **Tk_ConfigureInfo** for information on the format of this list). If *option* is specified with no *value*, then the command returns a list describing the one named option (this list will be identical to the corresponding sublist of the value returned if no *option* is specified). If one or more *option−value* pairs are specified, then the command modifies the given widget option(s) to have the given value(s); in this case the command returns an empty string. *Option* may have any of the values accepted by the **extfileselectiondialog** command.

COMPONENTS

Name: **fsb**
Class: **Fileselectionbox**

The fsb component is the extfileselectionbox for the extfileselectiondialog. See the "extfileselectionbox" widget manual entry for details on the fsb component item.

EXAMPLE

```
#
# Non-modal example
```

```
#
proc okCallback {} {
   puts "You selected [.nmfsd get]"
   .nmfsd deactivate
}

extfileselectiondialog .nmfsd -title Non-Modal
.nmfsd buttonconfigure OK -command okCallback

.nmfsd activate

#
# Modal example
#
extfileselectiondialog .mfsd -modality application
.mfsd center

if {[.mfsd activate]} {
   puts "You selected [.mfsd get]"
} else {
   puts "You cancelled the dialog"
}
```

AUTHOR

Mark L. Ulferts

Anthony L. Parent

KEYWORDS

extfileselectiondialog, extfileselectionbox, dialog, dialogshell, shell, widget

NAME

feedback − Create and manipulate a feedback widget to display feedback on the current status of an ongoing operation to the user.

SYNOPSIS

feedback *pathName* ?*options*?

INHERITANCE

itk::Widget <- Labeledwidget <- Feedback

STANDARD OPTIONS

background	**cursor**	**foreground**	**highlightColor**
highlightThickness			

See the "options" manual entry for details on the standard options.

INHERITED OPTIONS

labelBitmap	**labelFont**	**labelImage**	**labelMargin**
labelPos	**labelText**	**labelVariable**	

See the "labeledwidget" class manual entry for details on the inherited options.

WIDGET-SPECIFIC OPTIONS

Name: **barcolor**
Class: **BarColor**
Command-Line Switch: **-barcolor**

Specifies the color of the status bar, in any of the forms acceptable to **Tk_GetColor**. The default is DodgerBlue.

Name: **barheight**
Class: **BarHeight**
Command-Line Switch: **-barheight**

Specifies the height of the status bar, in any of the forms acceptable to **Tk_GetPixels**. The default is 20.

Name: **troughColor**
Class: **TroughColor**
Command-Line Switch: **-troughcolor**

Specifies the color of the frame in which the status bar sits, in any of the forms acceptable to **Tk_GetColor**. The default is white.

Name: **steps**
Class: **Steps**
Command-Line Switch: **-steps**

Specifies the total number of steps for the status bar. The default is 10.

DESCRIPTION

The **feedback** command creates a widget to display feedback on the current status of an ongoing operation to the user. Display is given as a percentage and as a thermometer type bar. Options exist for adding a label and controlling its position.

METHODS

The **feedback** command creates a new Tcl command whose name is *pathName*. This command may be used to invoke various operations on the widget. It has the following general form:

pathName option ?*arg arg ...*?

Option and the *arg*s determine the exact behavior of the command. The following commands are possible for scrolledtext widgets:

WIDGET-SPECIFIC METHODS

pathName **cget** *option*

Returns the current value of the configuration option given by *option*. *Option* may have any of the values accepted by the **scrolledhtml** command.

pathName **configure** ?*option*? ?*value option value ...*?

Query or modify the configuration options of the widget. If no *option* is specified, returns a list describing all of the available options for *pathName* (see **Tk_ConfigureInfo** for information on the format of this list). If *option* is specified with no *value*, then the command returns a list describing the one named option (this list will be identical to the corresponding sublist of the value returned if no *option* is specified). If one or more *option−value* pairs are specified, then the command modifies the given widget option(s) to have the given value(s); in this case the command returns an empty string. *Option* may have any of the values accepted by the **feedback** command.

pathName **reset**

Reset the current number of steps completed to 0, and configures the percentage complete label text to 0%

pathName **step** ?*inc*?

Increase the current number of steps completed by the amount specified by *inc*. *Inc* defaults to 1.

EXAMPLE

```
feedback .fb -labeltext "Status" -steps 20
pack .fb -padx 10 -pady 10 -fill both -expand yes

for {set i 0} {$i < 20} {incr i} {
   .fb step
   after 500
}
```

ACKNOWLEDGEMENTS

Sam Shen

This code is based largely on his feedback.tcl code from tk inspect. The original feedback code is copyright 1995 Lawrence Berkeley Laboratory.

AUTHOR

Kris Raney

KEYWORDS

feedback, widget

NAME

fileselectionbox − Create and manipulate a file selection box widget

SYNOPSIS

fileselectionbox *pathName* ?*options*?

INHERITANCE

itk::Widget <- Fileselectionbox

STANDARD OPTIONS

activeBackground	**background**	**borderWidth**	**cursor**
foreground	**highlightColor**	**highlightThickness**	**insertBackground**
insertBorderWidth	**insertOffTime**	**insertOnTime**	**insertWidth**
selectBackground	**selectBorderWidth**	**selectForeground**	

See the "options" manual entry for details on the standard options.

ASSOCIATED OPTIONS

textBackground **textFont**

See the "entryfield" widget manual entry for details on the above associated options.

labelFont

See the "labeledwidget" widget manual entry for details on the above associated options.

activeRelief **elementBorderWidth** **jump** **troughColor**

See the "scrollbar" widget class manual entry for details on the above associated options.

textBackground **textFont**

See the "scrolledlistbox" widget manual entry for details on the above associated options.

WIDGET-SPECIFIC OPTIONS

Name: **childSitePos**
Class: **Position**
Command-Line Switch: **-childsitepos**

Specifies the position of the child site in the selection box: **n**, **s**, **e**, **w**, **top**, **bottom**, or **center**. The default is s.

Specifies a Tcl command procedure which is called when an file list item is double clicked. Typically this occurs when mouse button 1 is double clicked over a file name.

Name: **directory**
Class: **Directory**
Command-Line Switch: **-directory**

Specifies the initial default directory. The default is the present working directory.

Name: **dirSearchCommand**
Class: **Command**
Command-Line Switch: **-dirsearchcommand**

Specifies a Tcl command to be executed to perform a directory search. The command will receive the current working directory and filter mask as arguments. The command should return a list of files which will be placed into the directory list.

Name: **dirsLabel**
Class: **Text**
Command-Line Switch: **-dirslabel**

Specifies the text of the label for the directory list. The default is "Directories".

Name: **dirsOn**
Class: **DirsOn**
Command-Line Switch: **-dirson**

Specifies whether or not to display the directory list. The value may be given in any of the forms acceptable to **Tcl_GetBoolean**. The default is true.

Name: **fileSearchCommand**
Class: **Command**
Command-Line Switch: **-filesearchcommand**

Specifies a Tcl command to be executed to perform a file search. The command will receive the current working directory and filter mask as arguments. The command should return a list of files which will be placed into the file list.

Name: **filesLabel**
Class: **Text**
Command-Line Switch: **-fileslabel**

Specifies the text of the label for the files list. The default is "Files".

Name: **filesOn**
Class: **FilesOn**
Command-Line Switch: **-fileson**

Specifies whether or not to display the files list. The value may be given in any of the forms acceptable to **Tcl_GetBoolean**. The default is true.

Name: **fileType**
Class: **FileType**
Command-Line Switch: **-filetype**

Specify the type of files which may appear in the file list: **regular**, **directory**, or **any**. The default is regular.

Name: **filterCommand**
Class: **Command**
Command-Line Switch: **-filtercommand**

Specifies a Tcl command to be executed upon hitting the Return key in the filter entry widget.

Name: **filterLabel**
Class: **Text**
Command-Line Switch: **-filterlabel**

Specifies the text of the label for the filter entry field. The default is "Filter".

Name: **filterOn**
Class: **FilterOn**
Command-Line Switch: **-filteron**

Specifies whether or not to display the filter entry. The value may be given in any of the forms acceptable to **Tcl_GetBoolean**. The default is true.

Name: **height**
Class: **Height**
Command-Line Switch: **-height**

Specifies the height of the selection box. The value may be specified in any of the forms acceptable to Tk_GetPixels. The default is 360 pixels.

Name: **invalid**
Class: **Command**
Command-Line Switch: **-invalid**

Command to be executed should the filter contents be proven invalid. The default is {bell}.

Name: **mask**
Class: **Mask**
Command-Line Switch: **-mask**

Specifies the initial file mask string. The default is "*".

Name: **noMatchString**
Class: **NoMatchString**
Command-Line Switch: **-nomatchstring**

Specifies the string to be displayed in the files list should no files match the mask. The default is "".

Name: **selectDirCommand**
Class: **Command**
Command-Line Switch: **-selectdirommand**

Specifies a Tcl command to be executed following selection of a directory in the directory list.

Name: **selectFileCommand**
Class: **Command**
Command-Line Switch: **-selectfileommand**

Specifies a Tcl command to be executed following selection of a file in the files list.

Name: **selectionCommand**
Class: **Command**
Command-Line Switch: **-selectioncommand**

Specifies a Tcl command to be executed upon hitting the Return key in the selection entry widget.

Name: **selectionLabel**
Class: **Text**
Command-Line Switch: **-selectionlabel**

Specifies the text of the label for the selection entry field. The default is "Selection".

Name: **selectionOn**
Class: **SelectionOn**
Command-Line Switch: **-selectionon**

Specifies whether or not to display the selection entry. The value may be given in any of the forms acceptable to **Tcl_GetBoolean**. The default is true.

Name: **width**
Class: **Width**
Command-Line Switch: **-width**

Specifies the width of the selection box. The value may be specified in any of the forms acceptable to Tk_GetPixels. The default is 470 pixels.

DESCRIPTION

The **fileselectionbox** command creates a file selection box similar to the OSF/Motif standard Xmfileselectionbox composite widget. The fileselectionbox is composed of directory and file scrolled lists as well as filter and selection entry fields. Bindings are in place such that selection of a directory list item loads the filter entry field and selection of a file list item loads the selection entry field. Options exist to control the appearance and actions of the widget.

METHODS

The **fileselectionbox** command creates a new Tcl command whose name is *pathName*. This command may be used to invoke various operations on the widget. It has the following general form:

pathName option ?*arg arg ...*?

Option and the *arg*s determine the exact behavior of the command. The following commands are possible for fileselectionbox widgets:

WIDGET-SPECIFIC METHODS

pathName **cget** *option*

Returns the current value of the configuration option given by *option*. *Option* may have any of the values accepted by the **fileselectionbox** command.

pathName **childsite**

Returns the child site widget path name.

pathName **configure** ?*option*? ?*value option value ...*?

Query or modify the configuration options of the widget. If no *option* is specified, returns a list describing all of the available options for *pathName* (see **Tk_ConfigureInfo** for information on the format of this list). If *option* is specified with no *value*, then the command returns a list describing the one named option (this list will be identical to the corresponding sublist of the value returned if no *option* is specified). If one or more *option−value* pairs are specified, then the command modifies the given widget option(s) to have the given value(s); in this case the command returns an empty string. *Option* may have any of the values accepted by the **fileselectionbox** command.

pathName **filter**

Update the current contents of the file selection box based on the current filter entry field value.

pathName **get**

Returns the current value of the selection entry widget.

COMPONENTS

Name: **dirs**
Class: **Scrolledlistbox**

The dirs component is the directory list box for the file selection box. See the "scrolledlistbox" widget manual entry for details on the dirs component item.

Name: **files**
Class: **Scrolledlistbox**

The files component is the file list box for the file selection box. See the "scrolledlistbox" widget manual entry for details on the files component item.

Name: **filter**
Class: **Entryfield**

The filter component is the entry field for user input of the filter value. See the "entryfield" widget

manual entry for details on the filter component item.

Name: **selection**
Class: **Entryfield**

The selection component is the entry field for user input of the currently selected file value. See the "entryfield" widget manual entry for details on the selection component item.

EXAMPLE

```
fileselectionbox .fsb
pack .fsb -padx 10 -pady 10 -fill both -expand yes
```

AUTHOR(S)

Mark L. Ulferts

KEYWORDS

fileselectionbox, widget

NAME

fileselectiondialog − Create and manipulate a file selection dialog widget

SYNOPSIS

fileselectiondialog *pathName* ?*options*?

INHERITANCE

itk::Toplevel <- Shell <- Dialogshell <- Dialog <- Fileselectiondialog

STANDARD OPTIONS

activeBackground	**background**	**borderWidth**	**cursor**
foreground	**highlightColor**	**highlightThickness**	**insertBackground**
insertBorderWidth	**insertOffTime**	**insertOnTime**	**insertWidth**
selectBackground	**selectBorderWidth**	**selectForeground**	

See the "options" manual entry for details on the standard options.

ASSOCIATED OPTIONS

textBackground	**textFont**

See the "entryfield" widget manual entry for details on the above associated options.

childSitePos	**directory**	**dirsLabel**	**dirSearchCommand**
dirsOn	**filesLabel**	**filesLabelOn**	**fileSearchCommand**
filesOn	**fileType**	**filterLabel**	**filterOn**
invalid	**mask**	**noMatchString**	**selectionLabel**
selectionOn			

See the "fileselectionbox" widget manual entry for details on the above associated options.

labelFont

See the "labeledwidget" widget manual entry for details on the above associated options.

textBackground	**textFont**

See the "scrolledlistbox" widget manual entry for details on the above associated options.

activeRelief	**elementBorderWidth**	**jump**	**troughColor**

See the "scrollbar" widget class manual entry for details on the above associated options.

INHERITED OPTIONS

buttonBoxPadX	**buttonBoxPadY**	**buttonBoxPos**	**padX**
padY	**separator**	**thickness**	

See the "dialogshell" widget manual entry for details on the above inherited options.

height	**master**	**modality**	**width**

See the "shell" widget manual entry for details on the above inherited options.

title

See the "Toplevel" widget manual entry for details on the above inherited options.

DESCRIPTION

The **fileselectiondialog** command creates a file selection dialog similar to the OSF/Motif standard composite widget. The fileselectiondialog is derived from the Dialog class and is composed of a FileSelectionBox with attributes set to manipulate the dialog buttons.

METHODS

The **fileselectiondialog** command creates a new Tcl command whose name is *pathName*. This command may be used to invoke various operations on the widget. It has the following general form:

pathName option ?*arg arg ...*?

Option and the *arg*s determine the exact behavior of the command. The following commands are possible for fileselectiondialog widgets:

ASSOCIATED METHODS

get **childsite** **filter**

See the "fileselectionbox" class manual entry for details on the associated methods.

INHERITED METHODS

add **buttonconfigure** **default** **hide**
insert **invoke** **show**

See the "buttonbox" widget manual entry for details on the above inherited methods.

activate **center** **deactivate**

See the "shell" widget manual entry for details on the above inherited methods.

WIDGET-SPECIFIC METHODS

pathName **cget** *option*

Returns the current value of the configuration option given by *option*. *Option* may have any of the values accepted by the **fileselectiondialog** command.

pathName **configure** ?*option*? ?*value option value ...*?

Query or modify the configuration options of the widget. If no *option* is specified, returns a list describing all of the available options for *pathName* (see **Tk_ConfigureInfo** for information on the format of this list). If *option* is specified with no *value*, then the command returns a list describing the one named option (this list will be identical to the corresponding sublist of the value returned if no *option* is specified). If one or more *option−value* pairs are specified, then the command modifies the given widget option(s) to have the given value(s); in this case the command returns an empty string. *Option* may have any of the values accepted by the **fileselectiondialog** command.

COMPONENTS

Name: **fsb**
Class: **Fileselectionbox**

The fsb component is the file selection box for the file selection dialog. See the "fileselectionbox" widget manual entry for details on the fsb component item.

EXAMPLE

```
#
# Non-modal example
#
proc okCallback {} {
   puts "You selected [.nmfsd get]"
   .nmfsd deactivate
}
```

```
fileselectiondialog .nmfsd -title Non-Modal
.nmfsd buttonconfigure OK -command okCallback

.nmfsd activate

#
# Modal example
#
fileselectiondialog .mfsd -modality application
.mfsd center

if {[.mfsd activate]} {
   puts "You selected [.mfsd get]"
} else {
   puts "You cancelled the dialog"
}
```

AUTHOR

Mark L. Ulferts

KEYWORDS

fileselectiondialog, fileselectionbox, dialog, dialogshell, shell, widget

NAME

finddialog − Create and manipulate a find dialog widget

SYNOPSIS

finddialog *pathName* ?*options*?

INHERITANCE

itk::Toplevel <- Shell <- Dialogshell <- Finddialog

STANDARD OPTIONS

activeBackground	**activeForeground**	**background**	**borderWidth**
cursor	**disabledForeground**	**font**	**foreground**
highlightColor	**highlightThickness**	**insertBackground**	**insertBorderWidth**
insertOffTime	**insertOnTime**	**insertWidth**	**selectBackground**
selectBorderWidth	**selectColor**	**selectForeground**	

See the "options" manual entry for details on the standard options.

ASSOCIATED OPTIONS

selectColor

See the "checkbutton" widget manual entry for details on the above associated options.

selectColor

See the "entryfield" widget manual entry for details on the above associated options.

labelFont

See the "labeledwidget" widget manual entry for details on the above associated options.

INHERITED OPTIONS

buttonBoxPadX	**buttonBoxPadY**	**buttonBoxPos**	**padX**
padY	**separator**	**thickness**	

See the "dialogshell" widget manual entry for details on the above inherited options.

height	**master**	**modality**	**width**

See the "shell" widget manual entry for details on the above inherited options.

title

See the "Toplevel" widget manual entry for details on the above inherited options.

WIDGET-SPECIFIC OPTIONS

Name: **clearCommand**
Class: **Command**
Command-Line Switch: **-clearcommand**

Specifies a command to be invoked following a clear operation. The option is meant to be used as means of notification that the clear has taken place and allow other actions to take place such as disabling a find again menu.

Name: **matchCommand**
Class: **Command**
Command-Line Switch: **-matchcommand**

Specifies a command to be invoked following a find operation. The command is called with a match point as an argument which identifies where exactly where in the text or scrolledtext widget that the match is located. Should a match not be found the match point is {}. The option is meant to be used as a means of notification that the find operation has completed and allow other actions to take place such as disabling a find again menu option if the match point was {}.

Name: **patternBackground**
Class: **Background**
Command-Line Switch: **-patternbackground**

Specifies the background color of the text matching the search pattern. It may have any of the forms accepted by Tk_GetColor. The default is gray44.

Name: **patternForeground**
Class: **Background**
Command-Line Switch: **-patternforeground**

Specifies the foreground color of the text matching the search pattern. It may have any of the forms accepted by Tk_GetColor. The default is white.

Name: **searchBackground**
Class: **Background**
Command-Line Switch: **-searchbackground**

Specifies the background color of the line containing the matching the search pattern. It may have any of the forms accepted by Tk_GetColor. The default is gray77.

Name: **searchForeground**
Class: **Background**
Command-Line Switch: **-searchforeground**

Specifies the foreground color of the line containing the matching the search pattern. It may have any of the forms accepted by Tk_GetColor. The default is black.

Name: **textWidget**
Class: **TextWidget**
Command-Line Switch: **-textwidget**

Specifies the text or scrolledtext widget to be searched.

DESCRIPTION

The **finddialog** command creates a find dialog that works in conjunction with a text or scrolledtext widget to provide a means of performing search operations. The user is prompted for a text pattern to be found in the text or scrolledtext widget. The search can be for all occurances, by regular expression, considerate of the case, or backwards.

METHODS

The **finddialog** command creates a new Tcl command whose name is *pathName*. This command may be used to invoke various operations on the widget. It has the following general form:

pathName option ?*arg arg* ...?

Option and the *arg*s determine the exact behavior of the command. The following commands are possible for finddialog widgets:

INHERITED METHODS

add **buttonconfigure** **default** **hide**
invoke **show**

See the "buttonbox" widget manual entry for details on the above inherited methods.

activate **center** **deactivate**

See the "shell" widget manual entry for details on the above inherited methods.

WIDGET-SPECIFIC METHODS

pathName **cget** *option*

Returns the current value of the configuration option given by *option*. *Option* may have any of the values accepted by the **finddialog** command.

pathName **clear**

Clears the pattern in the entry field and the pattern matchin indicators in the text or scrolledtext widget.

pathName **configure** ?*option*? ?*value option value ...*?

Query or modify the configuration options of the widget. If no *option* is specified, returns a list describing all of the available options for *pathName* (see **Tk_ConfigureInfo** for information on the format of this list). If *option* is specified with no *value*, then the command returns a list describing the one named option (this list will be identical to the corresponding sublist of the value returned if no *option* is specified). If one or more *option−value* pairs are specified, then the command modifies the given widget option(s) to have the given value(s); in this case the command returns an empty string. *Option* may have any of the values accepted by the **finddialog** command.

pathName **find**

Search for a specific text string in the text widget given by the -textwidget option. This method is the standard callback for the Find button. It is made available such that it can be bound to a find again action.

COMPONENTS

Name: **all**
Class: **Checkbutton**

The all component specifies that all the matches of the pattern should be found when performing the search. See the "checkbutton" widget manual entry for details on the all component item.

Name: **backwards**
Class: **Checkbutton**

The backwards component specifies that the search should continue in a backwards direction towards the beginning of the text or scrolledtext widget. See the "checkbutton" widget manual entry for details on the backwards component item.

Name: **case**
Class: **Checkbutton**

The case component specifies that the case of the pattern should be taken into consideration when performing the search. See the "checkbutton" widget manual entry for details on the case component item.

Name: **pattern**
Class: **Entryfield**

The pattern component provides the pattern entry field. See the "entryfield" widget manual entry for details on the pattern component item.

Name: **regexp**
Class: **Checkbutton**

The regexp component specifies that the pattern is a regular expression. See the "checkbutton" widget manual entry for details on the regexp component item.

EXAMPLE

```
scrolledtext .st
pack .st
.st insert end "Now is the time for all good men\n"
.st insert end "to come to the aid of their country"

finddialog .fd -textwidget .st
.fd center .st
.fd activate
```

AUTHOR

Mark L. Ulferts

KEYWORDS

finddialog, dialogshell, shell, widget

NAME

hierarchy − Create and manipulate a hierarchy widget

SYNOPSIS

hierarchy *pathName* ?*options*?

INHERITANCE

itk::Widget <- Labeledwidget <- Scrolledwidget <- Hierarchy

STANDARD OPTIONS

activeBackground	**activeForeground**	**background**	**borderWidth**
cursor	**disabledForeground**	**foreground**	**highlightColor**
highlightThickness	**relief**	**selectBackground**	**selectForeground**

See the "options" manual entry for details on the standard options.

ASSOCIATED OPTIONS

activeRelief	**elementBorderWidth**	**jump**	**troughColor**

See the "scrollbar" widget manual entry for details on the above associated options.

spacing1	**spacing2**	**spacing3**	**tabs**

See the "text" widget manual entry for details on the above associated options.

INHERITED OPTIONS

labelBitmap	**labelFont**	**labelImage**	**labelMargin**
labelPos	**labelText**	**labelVariable**	

See the "labeledwidget" class manual entry for details on the inherited options.

WIDGET-SPECIFIC OPTIONS

Name: **alwaysQuery**
Class: **AlwaysQuery**
Command-Line Switch: **-alwaysquery**

Boolean flag which tells the hierarchy widget weather or not each refresh of the display should be via a new query using the command value of the -querycommand option or use the values previous found the last time the query was made. The default is no.

Name: **closedIcon**
Class: **Icon**
Command-Line Switch: **-closedicon**

Specifies the name of an existing closed icon image to be used in the hierarchy before those nodes that are collapsed. Should one not be provided, then a folder icon will be generated, pixmap if possible, bitmap otherwise.

Name: **expanded**
Class: **Expanded**
Command-Line Switch: **-expanded**

When true, the hierarchy will be completely expanded when it is first displayed. A fresh display can be triggered by resetting the -querycommand option. The default is false.

Name: **filter**
Class: **Filter**
Command-Line Switch: **-filter**

When true only the branch nodes and selected items are displayed. This gives a compact view of important items. The default is false.

Name: **height**
Class: **Height**
Command-Line Switch: **-height**

Specifies the height of the hierarchy as an entire unit. The value may be specified in any of the forms acceptable to **Tk_GetPixels**. Any additional space needed to display the other components such as labels, margins, and scrollbars force the hierarchy to be compressed. A value of zero along with the same value for the width causes the value given for the visibleitems option to be applied which administers geometry constraints in a different manner. The default height is zero.

Name: **iconCommand**
Class: **Command**
Command-Line Switch: **-iconcommand**

Specifies a command to be executed upon user selection via mouse button one of any additional icons given in the values returned by the command associated with the -querycommand option. If this command contains "%n", it is replaced with the name of the node the icon belongs to. Should it contain "%i" then the icon name is substituted.

Name: **markBackground**
Class: **Foreground**
Command-Line Switch: **-markbackground**

Specifies the background color to use when displaying marked nodes.

Name: **markForeground**
Class: **Background**
Command-Line Switch: **-markforeground**

Specifies the foreground color to use when displaying marked nodes.

Name: **menuCursor**
Class: **Cursor**
Command-Line Switch: **-menucursor**

Specifies the mouse cursor to be used for the item and background menus. The value may have any of the forms accept able to Tk_GetCursor.

Name: **nodeIcon**
Class: **Icon**
Command-Line Switch: **-nodeicon**

Specifies the name of an existing node icon image to be used in the hierarchy before those nodes that are leafs. Should one not be provided, then a dog-eared page icon will be generated, pixmap if possible, bitmap otherwise.

Name: **openIcon**
Class: **Icon**
Command-Line Switch: **-openicon**

Specifies the name of an existing open icon image to be used in the hierarchy before those nodes that are expanded. Should one not be provided, then an open folder icon will be generated, pixmap if possible, bitmap otherwise.

Name: **queryCommand**
Class: **Command**
Command-Line Switch: **-querycommand**

Specifies the command executed to query the contents of each node. If this command contains "%n", it is replaced with the name of the desired node. In its simpilest form it should return the children of the given node as a list which will be depicted in the display. Since the names of the

children are used as tags in the underlying text widget, each child must be unique in the hierarchy. Due to the unique requirement, the nodes shall be reffered to as uids or uid in the singular sense. The format of returned list is

{uid [uid ...]}

where uid is a unique id and primary key for the hierarchy entry

Should the unique requirement pose a problem, the list returned can take on another more extended form which enables the association of text to be displayed with the uids. The uid must still be unique, but the text does not have to obey the unique rule. In addition, the format also allows the specification of additional tags to be used on the same entry in the hierarchy as the uid and additional icons to be displayed just before the node. The tags and icons are considered to be the property of the user in that the hierarchy widget will not depend on any of their values. The extended format is

{{uid [text [tags [icons]]]} {uid [text [tags [icons]]]} ...}

where uid is a unique id and primary key for the hierarchy entry
text is the text to be displayed for this uid
tags is a list of user tags to be applied to the entry
icons is a list of icons to be displayed in front of the text

The hierarchy widget does a look ahead from each node to determine if the node has a children. This can be cost some performace with large hierarchies. User's can avoid this by providing a hint in the user tags. A tag of "leaf" or "branch" tells the hierarchy widget the information it needs to know thereby avoiding the look ahead operation.

Name: **hscrollMode**
Class: **ScrollMode**
Command-Line Switch: **-hscrollmode**

Specifies the the display mode to be used for the horizontal scrollbar: **static, dynamic,** or **none**. In static mode, the scroll bar is displayed at all times. Dynamic mode displays the scroll bar as required, and none disables the scroll bar display. The default is static.

Name: **sbWidth**
Class: **Width**
Command-Line Switch: **-sbwidth**

Specifies the width of the scrollbar in any of the forms acceptable to **Tk_GetPixels**.

Name: **scrollMargin**
Class: **Margin**
Command-Line Switch: **-scrollmargin**

Specifies the distance between the text portion of the hierarchy and the scrollbars in any of the forms acceptable to **Tk_GetPixels**. The default is 3 pixels.

Name: **textBackground**
Class: **Background**
Command-Line Switch: **-textbackground**

Specifies the background color for the text portion of the hierarchy in any of the forms acceptable to **Tk_GetColor**.

Name: **textFont**
Class: **Font**
Command-Line Switch: **-textfont**

Specifies the font to be used in the text portion of the hierarchy.

Name: **visibleitems**
Class: **VisibleItems**
Command-Line Switch: **-visibleitems**

Specifies the widthxheight in characters and lines for the hierarchy. This option is only administered if the width and height options are both set to zero, otherwise they take precedence. The default value is 80x24. With the visibleitems option engaged, geometry constraints are maintained only on the text portion of the hierarchy. The size of the other components such as labels, margins, and scroll bars, are additive and independent, effecting the overall size of the hierarchy. In contrast, should the width and height options have non zero values, they are applied to the hierarchy as a whole. The hierarchy is compressed or expanded to maintain the geometry constraints.

Name: **vscrollMode**
Class: **ScrollMode**
Command-Line Switch: **-vscrollmode**

Specifies the the display mode to be used for the vertical scrollbar: **static, dynamic,** or **none**. In static mode, the scroll bar is displayed at all times. Dynamic mode displays the scroll bar as required, and none disables the scroll bar display. The default is static.

Name: **width**
Class: **Width**
Command-Line Switch: **-width**

Specifies the width of the hierarchy as an entire unit. The value may be specified in any of the forms acceptable to **Tk_GetPixels**. Any additional space needed to display the other components such as labels, margins, and scrollbars force the text portion of the hierarchy to be compressed. A value of zero along with the same value for the height causes the value given for the visibleitems option to be applied which administers geometry constraints in a different manner. The default width is zero.

DESCRIPTION

The **hierarchy** command creates a hierarchical data view widget. It allows the graphical management of a a list of nodes that can be expanded or collapsed. Individual nodes can be highlighted. Clicking with the right mouse button on any item brings up a special item menu. Clicking on the background area brings up a different popup menu. Options exist to provide user control over the loading of the nodes and actions associated with node selection. Since the hierarchy is based on the scrolledtext widget, it includes options to control the method in which the scrollbars are displayed, i.e. statically or dynamically. Options also exist for adding a label to the hierarchy and controlling its position.

METHODS

The **hierarchy** command creates a new Tcl command whose name is *pathName*. This command may be used to invoke various operations on the widget. It has the following general form:

pathName option ?*arg arg ...*?

Option and the *arg*s determine the exact behavior of the command. The following commands are possible for hierarchy widgets:

ASSOCIATED METHODS

bbox	**compare**	**debug**	**delete**
dlineinfo	**dump**	**get**	**index**
insert	**scan**	**search**	**see**

tag **window** **xview** **yview**

See the "text" manual entry for details on the standard methods.

WIDGET-SPECIFIC METHODS

pathName **cget** *option*

Returns the current value of the configuration option given by *option*. *Option* may have any of the values accepted by the **hierarchy** command.

pathName **clear**

Removes all items from the hierarchy display including all tags and icons. The display will remain empty until the -filter or -querycommand options are set.

pathName **collapse** *uid*

Collapses the hierarchy beneath the node with the specified unique id by one level. Since this can take a moment for large hierarchies, the cursor will be changed to a watch during the collapse. Also, if any of the nodes beneath the node being collapsed are selected, their status is changed to unselected.

pathName **configure** ?*option*? ?*value option value* ...?

Query or modify the configuration options of the widget. If no *option* is specified, returns a list describing all of the available options for *pathName* (see **Tk_ConfigureInfo** for information on the format of this list). If *option* is specified with no *value*, then the command returns a list describing the one named option (this list will be identical to the corresponding sublist of the value returned if no *option* is specified). If one or more *option−value* pairs are specified, then the command modifies the given widget option(s) to have the given value(s); in this case the command returns an empty string. *Option* may have any of the values accepted by the **hierarchy** command.

pathName **current**

Returns the tags for the node that was most recently selected by the right mouse button when the item menu was posted. Usually used by the code in the item menu to figure out what item is being manipulated.

pathName **draw** ?*when*?

Performs a complete redraw of the entire hierarchy. When may be either -now or -eventually where the latter means the draw can be performed after idle.

pathName **expand** *uid*

Expands the hierarchy beneath the node with the specified unique id by one level. Since this can take a moment for large hierarchies, the cursor will be changed to a watch during the expansion.

pathName **mark** *option ?arg arg ...?*

This command is used to manipulate marks which is quite similar to selection, adding a secondary means of hilighting an item in the hierarchy. The exact behavior of the command depends on the *option* argument that follows the **mark** argument. The following forms of the command are currently supported:

pathName **mark clear**

Clears all the currently marked nodes in the hierarchy.

pathName **mark add** *uid* ?*uid uid* ...?

Marks the nodes with the specified uids in the hierarchy using the **-markbackground** and **-markforeground** options and without affecting the mark state of any other nodes that were already marked.

pathName **mark remove** *uid* ?*uid uid* ...?

Unmarks the nodes with the specified uids in the hierarchy without affecting the mark state of any other nodes that were already marked.

pathName **mark get**
Returns a list of the unique ids that are currently marked.

pathName **refresh** *uid*
Performs a redraw of a specific node that has the given uid. If the node is not currently visible or in other words already drawn on the text, then no action is taken.

pathName **prune** *uid*
Removes the node specified by the given uid from the hierarchy. Should the node have children, then all of its children will be removed as well.

pathName **selection** *option* ?*arg arg ...*?
This command is used to manipulate the selection of nodes in the hierarchy. The exact behavior of the command depends on the *option* argument that follows the **selection** argument. The following forms of the command are currently supported:

pathName **selection clear**
Clears all the currently selected nodes in the hierarchy.

pathName **selection add** *uid* ?*uid uid ...*?
Selects the nodes with the specified uids in the hierarchy using the **-selectionbackground** and **-selectionforeground** options and without affecting the selection state of any other nodes that were already selected.

pathName **selection remove** *uid* ?*uid uid ...*?
Deselects the nodes with the specified uids in the hierarchy without affecting the selection state of any other nodes that were already selected.

pathName **selection get**
Returns a list of the unique ids that are currently selected.

A nodes selection status is also dependent on it being visible. If a node is selected and its parent is then collapsed making the selected node not visible, then its selection status is changed to unselected.

pathName **toggle** *uid*
Toggles the hierarchy beneath the node with the specified unique id. If the hierarchy is currently expanded, then it is collapsed, and vice-versa.

COMPONENTS

Name: **list**
Class: **Text**

The list component is the text widget in which the hierarchy is displayed. See the "text" widget manual entry for details on the text component item.

Name: **bgMenu**
Class: **Menu**

The bgMenu component is the popup menu which is displayed upon pressing the right mouse button in the background, i.e. not over a specific node. Menu items can be added along with their commands via the component command. See the "menu" widget manual entry for details on the bgMenu component item.

Name: **horizsb**
Class: **Scrollbar**

The horizsb component is the horizontal scroll bar. See the "scrollbar" widget manual entry for details on the horizsb component item.

Name: **itemMenu**
Class: **Menu**

The itemMenu component is the popup menu which is displayed upon selection of a hierarchy node with the right mouse button. Menu items can be added along with their commands via the component command. See the "menu" widget manual entry for details on the itemMenu component item.

Name: **vertsb**
Class: **Scrollbar**

The vertsb component is the vertical scroll bar. See the "scrollbar" widget manual entry for details on the vertsb component item.

EXAMPLE

```
proc get_files {file} {
   global env

   if {$file == ""} {
      set dir $env(HOME)
   } else {
      set dir $file
   }

   if {[catch {cd $dir}] != 0} {
      return ""
   }

   set rlist ""

   foreach file [lsort [glob -nocomplain *]] {
      lappend rlist [list [file join $dir $file] $file]
   }

   return $rlist
}

hierarchy .h -querycommand "get_files %n" -visibleitems 30x15     -labeltext $env(HOME)
pack .h -side left -expand yes -fill both
```

AUTHOR

Mark L. Ulferts

Michael J. McLennan

KEYWORDS

hierarchy, text, widget

NAME

hyperhelp − Create and manipulate a hyperhelp widget

SYNOPSIS

hyperhelp *pathName* ?*options*?

INHERITANCE

itk::Toplevel <- shell <- hyperhelp

STANDARD OPTIONS

activeBackground	**background**	**borderWidth**
closecmd	**cursor**	**exportSelection**
foreground	**highlightColor**	**highlightThickness**
insertBackground	**insertBorderWidth**	**insertOffTime**
insertOnTime	**insertWidth**	**padX**
padY	**relief**	**repeatDelay**
repeatInterval	**selectBackground**	**selectBorderWidth**
selectForeground	**setGrid**	

See the "options" manual entry for details on the standard options.

ASSOCIATED OPTIONS

hscrollmode	**vscrollmode**	**textbackground**	**fontname**
fontsize	**fixedfont**	**link**	**linkhighlight**
width	**height**	**state**	**wrap**
unknownimage			

See the "scrolledhtml" widget manual entry for details on the above associated options.

INHERITED OPTIONS

modality	**title**

See the "shell" manual entry for details on the above inherited options.

WIDGET-SPECIFIC OPTIONS

Name: **topics**
Class: **Topics**
Command-Line Switch: **-topics**

Specifies a list of help topics in the form {?*topic*? ... }. *Topic* may either be a topic name, in which case the document associated with the topic should be in the file **helpdir**/*topic*.html, or it may be of the form {*name file*}. In the latter case, *name* is displayed in the topic menu, and selecting the name loads *file*. If file has a relative path, it is assumed to be relative to helpdir.

Name: **helpdir**
Class: **Directory**
Command-Line Switch: **-helpdir**

Specifies the directory where help files are located.

Name: **closeCmd**
Class: **CloseCmd**
Command-Line Switch: **-closecmd**

Specifies the tcl command to be executed when the close option is selected from the topics menu.

Name: **maxHistory**
Class: **MaxHistory**
Command-Line Switch: **-maxhistory**

Specifies the maximum number of entries stored in the history list

Name: **beforelink**
Class: **BeforeLink**
Command-Line Switch: **-beforelink**

Specifies a command to be eval'ed before a new link is displayed. The path of the link to be displayed is appended before evaling the command. A suggested use might be to busy the widget while a new page is being displayed.

Name: **afterlink**
Class: **AfterLink**
Command-Line Switch: **-afterlink**

Specifies a command to be eval'ed after a new link is completely displayed. The path of the link that was displayed is appended before evaling the command.

DESCRIPTION

The **hyperhelp** command creates a shell window with a pulldown menu showing a list of topics. The topics are displayed by importing a HTML formatted file named **helpdir**/*topic*.html. For a list of supported HTML tags, see **scrolledhtml(n)**.

METHODS

The **hyperhelp** command creates a new Tcl command whose name is *pathName*. This command may be used to invoke various operations on the widget. It has the following general form:

pathName option ?*arg arg ...*?

Option and the *arg*s determine the exact behavior of the command. The following commands are possible for dialog widgets:

INHERITED METHODS

activate **center** **childsite** **deactivate**

See the "shell" manual entry for details on the above inherited methods.

WIDGET-SPECIFIC METHODS

pathName **cget** *option*

Returns the current value of the configuration option given by *option*. *Option* may have any of the values accepted by the **hyperhelp** command.

pathName **configure** ?*option*? ?*value option value ...*?

Query or modify the configuration options of the widget. If no *option* is specified, returns a list describing all of the available options for *pathName* (see **Tk_ConfigureInfo** for information on the format of this list). If *option* is specified with no *value*, then the command returns a list describing the one named option (this list will be identical to the corresponding sublist of the value returned if no *option* is specified). If one or more *option−value* pairs are specified, then the command modifies the given widget option(s) to have the given value(s); in this case the command returns an empty string. *Option* may have any of the values accepted by the **hyperhelp** command.

pathName **showtopic** *topic*

Display html file **helpdir**/*topic*.html. *Topic* may optionally be of the form *topicname#anchorname*. In this form, either *topicname* or *anchorname* or both may be empty. If *topicname* is empty, the current topic is assumed. If *anchorname* is empty, the top of the document is assumed

pathName **followlink** *href*

Display html file *href*. *Href* may be optionally be of the form *filename#anchorname*. In this form, either *filename* or *anchorname* or both may be empty. If *filename* is empty, the current document is assumed. If *anchorname* is empty, the top of the document is assumed.

pathName **forward**
Display html file one forward in history list, if applicable.

pathName **back**
Display html file one back in history list, if applicable.

EXAMPLE

```
hyperhelp .h -topics { Intro Help } -helpdir ~/help
.h showtopic Intro
```

AUTHOR

Kris Raney

KEYWORDS

hyperhelp, html, help, shell, widget

NAME

labeledframe − Create and manipulate a labeled frame widget

SYNOPSIS

labeledframe *pathName* ?*options*?

INHERITANCE

itk::Archetype <- labeledframe

STANDARD OPTIONS

background	**borderwidth**	**cursor**
foreground	**relief**	

See the "options" manual entry for details on the standard options.

WIDGET-SPECIFIC OPTIONS

Name: **iPadX**
Class: **IPad**
Command-Line Switch: **-ipadx**

Specifies horizontal padding space between the border and the childsite. The value may have any of the forms acceptable to **Tk_GetPixels**. The default is 0.

Name: **iPadY**
Class: **IPad**
Command-Line Switch: **-ipady**

Specifies vertical padding space between the border and the childsite. The value may have any of the forms acceptable to **Tk_GetPixels**. The default is 0.

Name: **labelBitmap**
Class: **Bitmap**
Command-Line Switch: **-labelbitmap**

Specifies a bitmap to display in the label, in any of the forms acceptable to **Tk_GetBitmap**. This option overrides the *labeltext* option.

Name: **labelImage**
Class: **Image**
Command-Line Switch: **-labelimage**

Specifies a image to be used as the label. The image may be any of the values created by the **image create** command. This option overrides both the *labelbitmap* and *labeletext* options.

Name: **labelMargin**
Class: **Margin**
Command-Line Switch: **-labelmargin**

Specifies the distance between the inner ede of the hull frames relief, and the label in any of the forms acceptable to **Tk_GetPixels**. The default is 10 pixels.

Name: **labelText**
Class: **Text**
Command-Line Switch: **-labeltext**

Specifies the text of the label around the childsite.

Name: **labelVariable**
Class: **Variable**
Command-Line Switch: **-labelvariable**

Specifies the text variable of the label around the childsite.

Name: **labelFont**
Class: **Font**
Command-Line Switch: **-labelfont**

Specifies the font of the label around the childsite.

Name: **labelPos**
Class: **Position**
Command-Line Switch: **-labelpos**

Specifies the position of the label within the grooved relief of the hull widget. **ne, n, nw, se, s, sw, en, e, es, wn, w, ws** Default is **n**.

DESCRIPTION

The **labeledframe** command creates a hull frame with a grooved relief, a label positioned within the grooved relief of the hull frame, and a frame childsite. The frame childsite can filled with any widget via a derived class or though the use of the childsite method. This class was designed to be a general purpose base class for supporting the combination of labeled frame and a childsite. The options include the ability to position the label at configurable locations within the grooved relief of the hull frame, and control the display of the label.

METHODS

The **labeledframe** command creates a new Tcl command whose name is *pathName*. This command may be used to invoke various operations on the widget. It has the following general form:

pathName option ?*arg arg ...*?

Option and the *arg*s determine the exact behavior of the command. The following commands are possible for labeledframe widgets:

WIDGET-SPECIFIC METHODS

pathName **childsite**

Return the path name of the child site.

pathName **cget** *option*

Returns the current value of the configuration option given by *option*. *Option* may have any of the values accepted by the **labeledframe** command.

pathName **configure** ?*option*? ?*value option value ...*?

Query or modify the configuration options of the widget. If no *option* is specified, returns a list describing all of the available options for *pathName* (see **Tk_ConfigureInfo** for information on the format of this list). If *option* is specified with no *value*, then the command returns a list describing the one named option (this list will be identical to the corresponding sublist of the value returned if no *option* is specified). If one or more *option−value* pairs are specified, then the command modifies the given widget option(s) to have the given value(s); in this case the command returns an empty string. *Option* may have any of the values accepted by the **labeledframe** command.

COMPONENTS

Name: **label**
Class: **label**

The label component provides the label for the labeled widget. See the "label" widget manual entry for details on the label component item.

EXAMPLE

The labeledframe was primarily meant to be a base class. The Radiobox is a good example of a derived classe of the labeledframe class. In order to provide equal support for composite classes, the ’childsite’ methods also exists. The following is an example of ’childsite’ method usage.

```
labeledframe .lw -labeltext "Entry Frame" -labelpos n
pack .lw -fill both -expand yes -padx 10 -pady 10
set cs [.lw childsite]

pack [Entryfield $cs.entry1 -labeltext "Name:"] -side top -fill x
pack [Spinint $cs.entry2 -labeltext "Number:"] -side top -fill x
pack [Pushbutton $cs.entry3 -text "Details:"] -side top -fill x
```

AUTHOR

John A. Tucker

KEYWORDS

labeledframe, widget

NAME

labeledwidget − Create and manipulate a labeled widget

SYNOPSIS

labeledwidget *pathName* ?*options*?

INHERITANCE

itk::Widget <- labeledwidget

STANDARD OPTIONS

background **cursor** **foreground**

See the "options" manual entry for details on the standard options.

WIDGET-SPECIFIC OPTIONS

Name: **disabledForeground**
Class: **DisabledForeground**
Command-Line Switch: **-disabledforeground**

Specifies the foreground to be used when the state is disabled.

Name: **labelBitmap**
Class: **Bitmap**
Command-Line Switch: **-labelbitmap**

Specifies a bitmap to display in the widget, in any of the forms acceptable to **Tk_GetBitmap**. This option overrides the *labeltext* option.

Name: **labelFont**
Class: **Font**
Command-Line Switch: **-labelfont**

Specifies the font to be used for the label.

Name: **labelImage**
Class: **Image**
Command-Line Switch: **-labelimage**

Specifies a image to be used as the label. The image may be any of the values created by the **image create** command. This option overrides both the *labelbitmap* and *labeletext* options.

Name: **labelMargin**
Class: **Margin**
Command-Line Switch: **-labelmargin**

Specifies the distance between the childsite and label in any of the forms acceptable to **Tk_GetPixels**. The default is 2 pixel.

Name: **labelPos**
Class: **Position**
Command-Line Switch: **-labelpos**

Specifies the position of the label along the side of the childsite: **nw**, **n**, **ne**, **sw**, **s**, **se**, **en**, **e**, **es**, **wn**, **w**, or **ws**. The default is w.

Name: **labelText**
Class: **Text**
Command-Line Switch: **-labeltext**

Specifies the text of the label around the childsite.

Name: **labelVariable**
Class: **Variable**
Command-Line Switch: **-labelvariable**

Specifies the text variable of the label around the childsite.

Name: **state**
Class: **State**
Command-Line Switch: **-state**

Specifies one of two states for the label: **normal** or **disabled**. If the label is disabled then it is displayed in a disabled foreground color. The default is normal.

DESCRIPTION

The **labeledwidget** command creates a labeled widget which contains a label and child site. The child site is a frame which can filled with any widget via a derived class or though the use of the childsite method. This class was designed to be a general purpose base class for supporting the combination of label widget and a childsite. The options include the ability to position the label around the childsite widget, modify the font and margin, and control the display of the labels.

METHODS

The **labeledwidget** command creates a new Tcl command whose name is *pathName*. This command may be used to invoke various operations on the widget. It has the following general form:

pathName option ?*arg arg ...*?

Option and the *arg*s determine the exact behavior of the command. The following commands are possible for labeledwidget widgets:

WIDGET-SPECIFIC METHODS

pathName **childsite**

Return the path name of the child site.

pathName **cget** *option*

Returns the current value of the configuration option given by *option*. *Option* may have any of the values accepted by the **labeledwidget** command.

pathName **configure** ?*option*? ?*value option value ...*?

Query or modify the configuration options of the widget. If no *option* is specified, returns a list describing all of the available options for *pathName* (see **Tk_ConfigureInfo** for information on the format of this list). If *option* is specified with no *value*, then the command returns a list describing the one named option (this list will be identical to the corresponding sublist of the value returned if no *option* is specified). If one or more *option−value* pairs are specified, then the command modifies the given widget option(s) to have the given value(s); in this case the command returns an empty string. *Option* may have any of the values accepted by the **labeledwidget** command.

STATIC METHODS

Labeledwidget::alignlabels *widget* ?*widget ...*?

The alignlabels procedure takes a list of widgets derived from the Labeledwidget class and uses the label margin to make each widget have the same total space for the combination of label and margin. The net effect is to left align the labels. Generally, this method is only useful with a label position of w, which is the default.

COMPONENTS

Name: **label**
Class: **label**

The label component provides the label for the labeled widget. See the "label" widget manual entry for details on the label component item.

Name: **lwchildsite**
Class: **frame**

The lwchildsite component is the user child site for the labeled widget. See the "frame" widget manual entry for details on the lwchildsite component item.

EXAMPLE

The labeledwidget was primarily meant to be a base class. The ScrolledListBox and EntryField are good examples of derived classes of the labeledwidget class. In order to provide equal support for composite classes, the ’childsite’ methods also exists. The following is an example of ’childsite’ method usage.

```
labeledwidget .lw -labeltext "Canvas Widget" -labelpos s
pack .lw -fill both -expand yes -padx 10 -pady 10

set cw [canvas [.lw childsite].c -relief raised -width 200 -height 200]
pack $cw -padx 10 -pady 10
```

AUTHOR

Mark L. Ulferts

KEYWORDS

labeledwidget, widget

NAME

mainwindow − Create and manipulate a mainwindow widget

SYNOPSIS

mainwindow *pathName* ?*options*?

INHERITANCE

itk::Toplevel <- shell <- mainwindow

STANDARD OPTIONS

background	**cursor**	**disabledForeground**	**font**
foreground	**highlightBackground**	**highlightColor**	**highlightThickness**

See the "options" manual entry for details on the standard options.

ASSOCIATED OPTIONS

balloonBackground	**balloonDelay1**	**balloonDelay2**	**ballonFont**
balloonForeground			

See the "toolbar" manual entry for details on the above associated options.

INHERITED OPTIONS

title

See the "Toplevel" manual entry for details on the above inherited options.

height	**master**	**modality**	**padX**
padY	**width**		

See the "shell" manual entry for details on the above inherited options.

WIDGET-SPECIFIC OPTIONS

Name: **helpLine**
Class: **HelpLine**
Command-Line Switch: **-helpline**

Specifies whether or not to display the help line. The value may be given in any of the forms acceptable to Tk_GetBoolean. The default is yes.

Name: **menuBarBackground**
Class: **Background**
Command-Line Switch: **-menubarbackground**

Specifies the normal background color for the menubar.

Name: **menuBarFont**
Class: **Font**
Command-Line Switch: **-menubarfont**

Specifies the font to use when drawing text inside the menubar.

Name: **menuBarForeround**
Class: **Foreground**
Command-Line Switch: **-menubarforeground**

Specifies the normal foreground color for the menubar.

Name: **statusLine**
Class: **StatusLine**
Command-Line Switch: **-statusline**

Specifies whether or not to display the status line. The value may be given in any of the forms acceptable to Tk_GetBoolean. The default is yes.

Name: **toolBarBackground**
Class: **Background**
Command-Line Switch: **-toolbarbackground**

Specifies the normal background color for the toolbar.

Name: **toolBarFont**
Class: **Font**
Command-Line Switch: **-toolbarfont**

Specifies the font to use when drawing text inside the toolbar.

Name: **toolBarForeround**
Class: **Foreground**
Command-Line Switch: **-toolbarforeground**

Specifies the normal foreground color for the toolbar.

DESCRIPTION

The **mainwindow** command creates a mainwindow shell which contains a menubar, toolbar, mousebar, childsite, status line, and help line. Each item may be filled and configured to suit individual needs.

METHODS

The **mainwindow** command create a new Tcl command whose name is *pathName*. This command may be used to invoke various operations on the widget. It has the following general form:

pathName option ?*arg arg ...*?

Option and the *arg*s determine the exact behavior of the command. The following commands are possible for mainwindow widgets:

INHERITED METHODS

activate **center** **deactivate**

See the "shell" manual entry for details on the above inherited methods.

WIDGET-SPECIFIC METHODS

pathName **cget** *option*

Returns the current value of the configuration option given by *option*. *Option* may have any of the values accepted by the **mainwindow** command.

pathName **childsite**

Returns the pathname of the child site widget.

pathName **configure** ?*option*? ?*value option value ...*?

Query or modify the configuration options of the widget. If no *option* is specified, returns a list describing all of the available options for *pathName* (see **Tk_ConfigureInfo** for information on the format of this list). If *option* is specified with no *value*, then the command returns a list describing the one named option (this list will be identical to the corresponding sublist of the value returned if no *option* is specified). If one or more *option−value* pairs are specified, then the command modifies the given widget option(s) to have the given value(s); in this case the command returns an empty string. *Option* may have any of the values accepted by the **mainwindow** command.

pathName **menubar** ?*args*?

The **menubar** method provides access to the menubar. Invoked with no arguments it returns the

pathname of the menubar. With arguments, they are evaluated against the menubar which in effect provides access to the entire API of the menubar. See the "menubar" manual entry for details on the commands available in the menubar.

pathName **mousebar** ?*args*?
The **mousebar** method provides access to the mousebar which is a vertical toolbar. Invoked with no arguments it returns the pathname of the mousebar. With arguments, they are evaluated against the mousebar which in effect provides access to the entire API of the underlying toolbar. See the "toolbar" manual entry for details on the commands available in the mousebar.

pathName **msgd** ?*args*?
The **msgd** method provides access to the messagedialog contained in the mainwindow. Invoked with no arguments it returns the pathname of the messagedialog. With arguments, they are evaluated against the messagedialog which in effect provides access to the entire API of the messagedialog. See the "messagedialog" manual entry for details on the commands available in the messagedialog.

pathName **toolbar** ?*args*?
The **toolbar** method provides access to the toolbar. Invoked with no arguments it returns the pathname of the toolbar. With arguments, they are evaluated against the toolbar which in effect provides access to the entire API of the toolbar. See the "toolbar" manual entry for details on the commands available in the toolbar.

COMPONENTS

Name: **help**
Class: **Label**

The help component provides a location for displaying any help strings provided in the menubar, toolbar, or mousebar. See the "label" widget manual entry for details on the help component item.

Name: **menubar**
Class: **Menubar**

The menubar component is the menubar located at the top of the window. See the "menubar" widget manual entry for details on the menubar component item.

Name: **mousebar**
Class: **Toolbar**

The mousebar component is the vertical toolbar located on the right side of the window. See the "toolbar" widget manual entry for details on the mousebar component item.

Name: **msgd**
Class: **Messagedialog**

The msgd component is a messagedialog which may be reconfigured as needed and used repeatedly throughout the application. See the "messagedialog" widget manual entry for details on the messagedialog component item.

Name: **status**
Class: **Label**

The status component provides a location for displaying application status information. See the "label" widget manual entry for details on the status component item.

Name: **toolbar**
Class: **Toolbar**

The toolbar component is the horizontal toolbar located on the top of the window. See the "toolbar" widget manual entry for details on the toolbar component item.

EXAMPLE

```
mainwindow .mw

#
# Add a File menubutton
#
.mw menubar add menubutton file -text "File" -underline 0 -padx 8 -pady 2 \
  -menu {options -tearoff no
                                command new -label "New" -underline 0 \
                                   -helpstr "Create a new file"
                                command open -label "Open ..." -underline 0 \
                                   -helpstr "Open an existing file"
                                command save -label "Save" -underline 0 \
                                   -helpstr "Save the current file"
                                command saveas -label "Save As ..." -underline 5 \
                                   -helpstr "Save the file as a differnet name"
                                command print -label "Print" -underline 0 \
                                   -helpstr "Print the file"
        separator sep1
                                command close -label "Close" -underline 0 \
                                   -helpstr "Close the file"
                                separator sep2
                                command exit -label "Exit" -underline 1 \
                                   -helpstr "Exit this application"
  }

#
# Install a scrolledtext widget in the childsite.
#
scrolledtext [.mw childsite].st
pack [.mw childsite].st -fill both -expand yes

#
# Activate the main window.
#
.mw activate
```

AUTHOR

Mark L. Ulferts

John A. Tucker

KEYWORDS

mainwindow, shell, widget

NAME

menubar − Create and manipulate menubar menu widgets

SYNOPSIS

menubar *pathName* ?*options*?

INHERITANCE

itk::Widget <- menubar

STANDARD OPTIONS

activeBackground	**borderWidth**	**highlightBackground**	**padY**
activeBorderWidth	**cursor**	**highligthThickness**	**relief**
activeForeground	**disabledForeground**	**highlightColor**	**wrapLength**
anchor	**font**	**justify**	
background	**foreground**	**padX**	

See the "options" manual entry for details on the standard options.

WIDGET-SPECIFIC OPTIONS

Name: **helpVariable**
Class: **HelpVariable**
Command-Line Switch: **-helpvariable**

Specifies the global variable to update whenever the mouse is in motion over a menu entry. This global variable is updated with the current value of the active menu entry's **helpStr**. Other widgets can "watch" this variable with the trace command, or as is the case with entry or label widgets, they can set their **textVariable** to the same global variable. This allows for a simple implementation of a help status bar. Whenever the mouse leaves a menu entry, the helpVariable is set to the empty string {}. The mainwindow(1) associates its helpstatus and its menubar in this fashion.

Name: **menuButtons**
Class: **MenuButtons**
Command-Line Switch: **-menubuttons**

The menuButton option is a string which specifies the arrangement of menubuttons on the menubar frame. Each menubutton entry is delimited by the newline character.

```
menubar .mb -menubuttons {
        menubutton file -text File
        menubutton edit -text Edit
        menubutton options -text Options
}
```

specifies that three menubuttons will be added to the menubar (file, edit, options). Each entry is translated into an add command call.

The **menuButtons** option can accept embedded variables, commands, and backslash quoting. Embedded variables and commands must be enclosed in curly braces ({}) to ensure proper parsing of the substituted values.

DESCRIPTION

The **menubar** command creates a new window (given by the *pathName* argument) and makes it into a **menubar** menu widget. Additional options, described above may be specified on the command line or in the option database to configure aspects of the menubar such as its colors and font. The **menubar** command returns its *pathName* argument. At the time this command is invoked, there must not exist a window named pathName, but pathName's parent must exist.

A **menubar** is a widget that simplifies the task of creating menu hierarchies. It encapsulates a **frame** widget, as well as **menubuttons**, **menus**, and menu **entries**. The menubar allows menus to be specified and referenced in a more consistent manner than using Tk to build menus directly.

Menubar allows a menu tree to be expressed in a hierachical "language". The **menubar** accepts a **menuButtons** option that allows a list of menubuttons to be added to the menubar. In turn, each menubutton accepts a **menu** option that specifies a list of menu entries to be added to the menubutton's menu. Cascade entries also accept the **menu** option for specifying a list of menu entries to be added to the cascade's menu.

Additionally, the menubar allows each component of the menubar system to be referenced by a simple *menuPathName* syntax. The menubar also extends the set of options for menu entries to include a **helpStr** option.

MENU PATH NAMES

A *menuPathName* is a series of component names separated by the '.' character. Each menubar component can be referenced via these *menuPathNames*. *menuPathNames* are similar to widget pathNames in Tk. Some correspond directly to a widget pathName (components of type **menu** or **menubutton**), others correspond to a menu entry type. Every widget and entry in a menubar can be referenced with the *menuPathName* naming convention. A menubar can have four types of components:

frame. A menubar holds exactly one frame which manages menubuttons. The frame is always signified by the '.' character as the path name.

menubutton. A menubutton corresponds directly to a Tk menubutton. See menubutton(n).

menu. A menu is attached to a menubutton and corresponds directly to Tk's menu widget. A menu is always signified by the *menuPathName* ending with the keyword **menu**. See menu(n).

entry. An entry corresponds directly to Tk's menu widget entries. Menus consist of a column of one line entries. Entries may be of type: **command**, **checkbutton**, **radiobutton**, **separator**, or **cascade**. For a complete description of these types see the discussion on **ENTRIES** in menu(n).

The suffix of a *menuPathName* may have the form of:

tkWidgetName Specifies the name of the component, either a **frame**, **menubutton**, **menu**, or an **entry**. This is the normal naming of widgets. For example, .file references a **menubutton** named *file*.

The *menuPathName* is a series of segment names, each separated by the '.' character. Segment names may be one of the following forms:

number Specifies the index of the the component. For menubuttons, 0 corresponds to the left-most menubutton of the menu bar frame. As an example, *.1* would correspond to the second menubutton on the menu bar frame.

For entries, 0 corresponds to the top-most entry of the menu. For example, .file.0 would correspond to the first entry on the menu attached to the menubutton named *file*.

end Specifes the last component. For menubuttons, it specifies the right-most entry of the menu bar frame. For menu entries, it specifies the bottom-most entry of the menu.

last Same as end.

Finally, menu components always end with the **menu** keyword. These components are automatically created via the -menu option on menubuttons and cascades or via the **add** or **insert** commands.

menu Specifes the menu pane that is associated with the given menubutton prefix. For example, *.file.menu* specifies the menu pane attached to the *.file* menubutton.

For example, the path *.file.new* specifies the entry named new on the menu associated with the file menubutton located on the menu bar. The path *.file.menu* specifies the menu pane associated with the menubutton *.file*. The path *.last* specifies the last menu on the menu bar. The path *.0.last* would specify the first menu (file) and the last entry on that menu (quit), yielding *.file.quit*.

As a restriction, the last name segment of *menuPathName* cannot be one of the keywords last, menu, end, nor may it be a numeric value (integer).

WIDGET-SPECIFIC METHODS

The **menubar** command creates a new Tcl command whose name is *pathName*. This command may be used to invoke various operations on the widget. It has the following general form:

pathName option ?*arg arg ...*?

option and the *arg*s determine the exact behavior of the command.

In addition, many of the widget commands for menubar take as one argument a path name to a menu component. These path names are called *menuPathName*s. See the discussion on **MENUBAR PATH NAMES** above.

The following commands are possible for menubar widgets:

pathName **add** *type menuPathName* ?*option value option value*?

Adds either a menu to the menu bar or a menu entry to a menu pane.

If additional arguments are present, they specify *option*s available to component type **entry**. See the man pages for **menu**(1) in the section on **ENTRIES**.

If *type* is one of **cascade**, **checkbutton**, **command**, **radiobutton**, or **separator** it adds a new entry to the bottom of the menu denoted by the prefix of *menuPathName*. If additonal arguments are present, they specify options available to menu **entry** widgets. In addition, the **helpStr** option is added by the menubar widget to all components of type entry.

-helpstr *value*

Specifes the string to associate with the entry. When the mouse moves over the associated entry, the variable denoted by **helpVariable** is set. Another widget can bind to the helpVariable and thus display status help.

If the type of the component added is **menubutton** or **cascade**, a menubutton or cascade is added to the menubar. If additional arguments are present, they specify options available to menubutton or cascade widgets. In addition, the **menu** option is added by the menubar widget to all menubutton and cascade widgets.

-menu *menuSpec*

This is only valid for *menuPathName*s of type **menubutton** or **cascade**. Specifes an option set and/or a set of entries to place on a menu and associate with the menubutton or cascade. The **option** keyword allows the menu widget to be configured. Each item in the *menuSpec* is treated as add commands (each with the possibility of having other -menu options). In this way a menu can be recursively built.

The last segment of *menuPathName* cannot be one of the keywords **last**, **menu**, **end**. Additionally, it may not be a *number*. However the *menuPathName* may be referenced in this manner (see discussion of **COMPONENT PATH NAMES**).

Note that the same curly brace quoting rules apply to **-menu** option strings as did to **-menubuttons** option strings. See the earlier discussion on **umenubuttons** in the "**WIDGET-SPECIFIC OPTIONS**" section.

pathName **cget** *option*
Returns the current value of the configuration option given by *option*.

pathName **configure** ?*options value option value*?
Query or modify the configuration options of the widget. If no *option* is specified, returns a list describing all of the available options for **pathName** (see **Tk_ConfigureInfo** for information on the format of this list). If *option* is specified with no value, then the command returns a list describing the one named option (this list will be identical to the corresponding sublist of the value returned if no option is specified). If one or more option-value pairs are specified, then the command modifies the given widget option(s) to have the given value(s); in this case the command returns an empty string.

pathName **delete** *menuPathName* ?*menuPathName2*?
If *menuPathName* is of component type **Menubutton** or **Menu**, delete operates on menus. If *menuPathName* is of component type **Entry**, delete operates on menu entries.

This command deletes all components between *menuPathName* and *menuPathName2* inclusive. If *menuPathName2* is omitted then it defaults to *menuPathName*. Returns an empty string.

If *menuPathName* is of type menubar, then all menus and the menu bar frame will be destroyed. In this case *menuPathName2* is ignored.

pathName **index** *menuPathName*
If *menuPathName* is of type menubutton or menu, it returns the position of the menu/menubutton on the menubar frame.

If *menuPathName* is of type **command**, **separator**, **radiobutton**, **checkbutton**, or **cascade**, it returns the menu widget's numerical index for the entry corresponding to *menuPathName*. If path is not found or the path is equal to ".", a value of -1 is returned.

pathName **insert** *menuPathName type name* ?*option value*?
Insert a new component named name before the component specified by *menuPathName*.

If *menuPathName* is of type **Menubutton** or **Menu**, the new component inserted is of type **Menu** and given the name name. In this case valid *option value* pairs are those accepted by menubuttons.

If *menuPathName* is of type **Entry**, the new component inserted is of type **entry** and given the name *name*. In this case, valid *option value* pairs are those accepted by menu entries. *Name* cannot be one of the keywords **last**, **menu**, **end**. Additionally, it may not be a number. However the *menuPathName* may be referenced in this manner (see discussion of **COMPONENT PATH NAMES**).

pathName **invoke** *menuPathName*
Invoke the action of the menu entry denoted by *menuPathName*. See the sections on the individual entries in the menu(1) man pages. If the menu entry is disabled then nothing happens. If the entry has a command associated with it then the result of that command is returned as the result of the **invoke** widget command. Otherwise the result is an empty string.

If *menuPathName* is not a menu entry, an error is issued.

pathName **menucget** *menuPathName option*
Returns the current value of the configuration option given by *option*. The component type of *menuPathName* determines the valid available options.

pathName **menuconfigure** *menuPathName* ?*option value*?
Query or modify the configuration options of the componet of the menubar specified by *menuPathName*. If no *option* is specified, returns a list describing all of the available options for

menuPathName (see **Tk_ConfigureInfo** for information on the format of this list). If *option* is specified with no value, then the command returns a list describing the one named option (this list will be identical to the corresponding sublist of the value returned if no option is specified). If one or more option-value pairs are specified, then the command modifies the given widget option(s) to have the given value(s); in this case the command returns an empty string. The component type of *menuPathName* determines the valid available options.

pathName **path** ?*mode*? *pattern*

Returns a fully formed *menuPathName* that matches *pattern*. If no match is found it returns -1. The *mode* argument indicates how the search is to be matched against *pattern* and it must have one of the following values:

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

-regexp Pattern is treated as a regular expression and matched against each component of the *menuPathName* using the same rules as the regexp command. The default mode is -glob.

pathName **type** *menuPathName*

Returns the type of the component specified by *menuPathName*. For menu entries, this is the type argument passed to the **add**/**insert** widget command when the entry was created, such as **command** or **separator**. Othewise it is either a **menubutton** or a **menu**.

pathName **yposition** *menuPathName*

Returns a decimal string giving the y-coordinate within the menu window of the topmost pixel in the entry specified by *menuPathName*. If the *menuPathName* is not an entry, an error is issued.

EXAMPLE ONE: USING GRAMMAR

The following example creates a menubar with "File", "Edit", "Options" menubuttons. Each of these menubuttons has an associated menu. In turn the File menu has menu entries, as well as the Edit menu and the Options menu. The Options menu is a tearoff menu with selectColor (for radiobuttons) set to blue. In addition, the Options menu has a cascade titled More, with several menu entries attached to it as well. An entry widget is provided to display help status.

```
menubar .mb -helpvariable helpVar -menubuttons {
  menubutton file -text File -menu {
    options -tearoff false
    command new -label New \
      -helpstr "Open new document" \
      -command {puts NEW}
    command close -label Close \
      -helpstr "Close current document" \
      -command {puts CLOSE}
    separator sep1
    command exit -label Exit -command {exit} \
      -helpstr "Exit application"
  }
  menubutton edit -text Edit -menu {
    options -tearoff false
    command undo -label Undo -underline 0 \
      -helpstr "Undo last command" \
      -command {puts UNDO}
    separator sep2
    command cut -label Cut -underline 1 \
      -helpstr "Cut selection to clipboard" \
      -command {puts CUT}
```

```
    command copy -label Copy -underline 1 \
      -helpstr "Copy selection to clipboard" \
      -command {puts COPY}
    command paste -label Paste -underline 0 \
      -helpstr "Paste clipboard contents" \
      -command {puts PASTE}
  }
  menubutton options -text Options -menu {
    options -tearoff false -selectcolor blue
    radiobutton byName -variable viewMode \
      -value NAME -label "by Name" \
      -helpstr "View files by name order" \
      -command {puts NAME}
    radiobutton byDate -variable viewMode \
      -value DATE -label "by Date" \
      -helpstr "View files by date order" \
      -command {puts DATE}
    cascade prefs -label Preferences -menu {
      command colors -label Colors... \
        -helpstr "Change text colors" \
        -command {puts COLORS}
      command fonts -label Fonts... \
        -helpstr "Change text font" \
        -command {puts FONT}
    }
  }
```

} frame .fr -width 300 -height 300 entry .ef -textvariable helpVar pack .mb -anchor nw -fill x -expand yes pack .fr -fill both -expand yes pack .ef -anchor sw -fill x -expand yes Alternatively the same menu could be created by using the add and configure methods:

```
menubar .mb
.mb configure -menubuttons {
        menubutton file -text File -menu {
                command new -label New
                command close -label Close
                separator sep1
                command     quit -label Quit
        }
        menubutton edit -text Edit
}
.mb add command .edit.undo -label Undo -underline 0
.mb add separator .edit.sep2
.mb add command .edit.cut -label Cut -underline 1
.mb add command .edit.copy -label Copy -underline 1
.mb add command .edit.paste -label Paste -underline 0

.mb add menubutton .options -text Options -menu {
        radiobutton byName -variable viewMode \
                -value NAME -label "by Name"
        radiobutton byDate -variable viewMode \
                -value DATE -label "by Date"
```

```
}

.mb add cascade .options.prefs -label Preferences -menu {
                                command colors -label Colors...
                                command fonts -label Fonts...
}
pack .mb -side left -anchor nw -fill x -expand yes
```

CAVEATS

The **-menubuttons** option as well as the **-menu** option is evaluated by menubar with the **subst** command. The positive side of this is that the option string may contain variables, commands, and/or backslash substitutions. However, substitutions might expand into more than a single word. These expansions can be protected by enclosing candidate substitutions in curly braces ({}). This ensures, for example, a value for an option will still be treated as a single value and not multiple values. The following example illustrates this case:

```
set fileMenuName "File Menu"
set var {}
menubar .mb -menubuttons {
        menubutton file -text {$fileMenuName}
        menubutton edit -text Edit -menu {
                checkbutton check \
                        -label Check \
                        -variable {[scope var]} \
                        -onvalue 1 \
                        -offvalue 0
        }
        menubutton options -text Options
}
```

The variable *fileMenuName* will expand to "File Menu" when the **subst** command is used on the menubutton specification. In addition, the [**scope**...] command will expand to @scope :: var. By enclosing these inside {} they stay as a single value. Note that only {} work for this. [list...], "" etc. will not protect these from the subst command.

ACKNOWLEDGMENTS

Bret Schumaker

1994 - Early work on a menubar widget.

Mark Ulferts, Mark Harrison, John Sigler

Invaluable feedback on grammar and usability of the menubar widget

AUTHOR

Bill W. Scott

KEYWORDS

frame, menu, menubutton, entries, help

NAME

messagebox − Create and manipulate a messagebox text widget

SYNOPSIS

messagebox *pathName* ?*options*?

INHERITANCE

itk::Widget <- Labeledwidget <- Scrolledwidget <- Messagebox

STANDARD OPTIONS

activeBackground	**activeForeground**	**background**	**borderWidth**
cursor	**exportSelection**	**font**	**foreground**
highlightColor	**highlightThickness**	**padX**	**padY**
relief	**setGrid**		

See the "options" manual entry for details on the standard options.

ASSOCIATED OPTIONS

labelBitmap	**labelFont**	**labelImage**	**labelMargin**
labelPos	**labelText**	**labelVariable**	

See the "labeledwidget" class manual entry for details on the above associated options.

activeRelief	**elementBorderWidth**	**jump**	**troughColor**

See the "scrollbar" widget manual entry for details on the above associated options.

height	**hscrollMode**	**sbWidth**	**scrollMargin**
textBackground	**visibleItems**	**vscrollMode**	**width**

See the "scrolledtext" widget manual entry for details on the above associated options.

spacing1	**spacing2**	**spacing3**

See the "text" widget manual entry for details on the above associated options.

WIDGET-SPECIFIC OPTIONS

Name: **fileName**
Class: **FileName**
Command-Line Switch: **-filename**

Specifies the filename to be displayed in the file selection dialog when it pops up during a save of the messagebox contents operation.

Name: **maxLines**
Class: **MaxLines**
Command-Line Switch: **-maxlines**

Specifies the maximum number of lines allowed in the text area of the messagebox. When this limit is reached, the oldest line will be deleted such that the total number of lines remains *max-lines*.

Name: **saveDir**
Class: **SaveDir**
Command-Line Switch: **-savedir**

Specifies the default directory to display when the file selection dialog pops up during a save of the messagebox contents operation. If this parameter is not specified, then the files in the current working directory are displayed.

DESCRIPTION

The **messagebox** command creates a scrolled information messages area widget. Message types can be user defined and configured. Their options include foreground, background, font, bell, and their display mode of on or off. This allows message types to defined as needed, removed when no longer so, and modified when necessary. An export method is provided for file I/O.

The number of lines displayed may be limited with the default being 1000. When this limit is reached, the oldest line is removed. A popup menu which appears when the right mouse button has been pressed in the message area has been predefined. The contents of the popup menu by default support clearing the area and saving its contents to a file. Additional operations may be defined or existing operations removed by using the component command to access the popup menu.

MESSAGE TYPES

The display characteristics of messages issued to the messagebox vary with the message type. Types are defined by the user and they may be added, removed, and configured. The options of the message type control the display include the following:

−background *color*

Color specifies the background color to use for characters associated with the message type. It may have any of the forms accepted by **Tk_GetColor**.

−bell *boolean*

Specifies whether or not to ring the bell whenenver a message of this type is issued. *Boolean* may have any of the forms accepted by **Tk_GetBoolean**. The default is 0.

-font *fontName*

FontName is the name of a font to use for drawing characters. It may have any of the forms accepted by Tk_GetFontStruct.

−foreground *color*

Color specifies the foreground color to use for characters associated with the message type. It may have any of the forms accepted by **Tk_GetColor**.

−show *boolean*

Specifies whether of not to display this message type when issued. *Boolean* may have any of the forms accepted by **Tk_GetBoolean**. The default is 1.

METHODS

The **messagebox** command creates a new Tcl command whose name is *pathName*. This command may be used to invoke various operations on the widget. It has the following general form:

pathName option ?*arg arg* ...?

Option and the *arg*s determine the exact behavior of the command. The following commands are possible for messagebox widgets:

WIDGET-SPECIFIC METHODS

pathName **cget** *option*

Returns the current value of the configuration option given by *option*. *Option* may have any of the values accepted by the **messagebox** command.

pathName **clear**
Clear the messagebox of all messages.

pathName **export** *filename*
Write text to a file. If *filename* exists then contents are replaced with text widget contents.

pathName **configure** ?*option*? ?*value option value* ...?
Query or modify the configuration options of the widget. If no *option* is specified, returns a list describing all of the available options for *pathName* (see **Tk_ConfigureInfo** for information on the format of this list). If *option* is specified with no *value*, then the command returns a list describing the one named option (this list will be identical to the corresponding sublist of the value returned if no *option* is specified). If one or more *option−value* pairs are specified, then the command modifies the given widget option(s) to have the given value(s); in this case the command returns an empty string. *Option* may have any of the values accepted by the **messagebox** command.

pathName **type** *option msgtype* ?*arg arg* ...?
This command is used to manipulate message types. The behavior of the command depends on the option argument that follows the type keyword. The following forms of the command are supported:

Name: **itemMenu**
Class: **Menu**

This is the popup menu that gets displayed when you right-click in the text area of the messagebox. Its contents may be modified via the component command.

Name: **text**
Class: **Scrolledtext**

The text component is the scrolledtext widget. See the "scrolledtext" widget manual entry for details on the text component item.

EXAMPLE

```
messagebox .mb -hscrollmode dynamic -labeltext "Messages" -labelpos n \
                                -height 120 -width 550 -savedir "/tmp" -textbackground #d9d9d9

pack .mb -padx 5 -pady 5 -fill both -expand yes

.mb type add ERROR -background red -foreground white -bell 1
.mb type add WARNING -background yellow -foreground black
.mb type add INFO -background white -foreground black

.mb issue "This is an error message in red with a beep" ERROR
.mb issue "This warning message in yellow" WARNING
.mb issue "This is an informational message" INFO
```

AUTHOR

Alfredo Jahn V

Mark L. Ulferts

KEYWORDS

messagebox, scrolledtext, text, widget

NAME

messagedialog − Create and manipulate a message dialog widget

SYNOPSIS

messagedialog *pathName* ?*options*?

INHERITANCE

itk::Toplevel <- Shell <- Dialogshell <- Dialog <- Messagedialog

STANDARD OPTIONS

background	**bitmap**	**cursor**	**font**
foreground	**image**	**text**	

See the "options" manual entry for details on the standard options.

INHERITED OPTIONS

buttonBoxPadX	**buttonBoxPadY**	**buttonBoxPos**	**padX**
padY	**separator**	**thickness**	

See the "dialogshell" widget manual entry for details on the above inherited options.

master	**modality**

See the "shell" widget manual entry for details on the above inherited options.

title

See the "Toplevel" widget manual entry for details on the above inherited options.

WIDGET-SPECIFIC OPTIONS

Name: **imagePos**
Class: **Position**
Command-Line Switch: **-imagepos**

Specifies the image position relative to the message text: **n**, **s**, **e**, or **w**. The default is w.

Name: **textPadX**
Class: **Pad**
Command-Line Switch: **-textpadx**

Specifies a non-negative value indicating how much extra space to request for the message text in the X direction. The value may have any of the forms acceptable to Tk_GetPixels.

Name: **textPadY**
Class: **Pad**
Command-Line Switch: **-textpady**

Specifies a non-negative value indicating how much extra space to request for the message text in the X direction. The value may have any of the forms acceptable to Tk_GetPixels.

DESCRIPTION

The **messagedialog** command creates a message dialog composite widget. The messagedialog is derived from the Dialog class and is composed of an image and associated message text with commands to manipulate the dialog buttons.

METHODS

The **messagedialog** command creates a new Tcl command whose name is *pathName*. This command may be used to invoke various operations on the widget. It has the following general form:

pathName option ?*arg arg* ...?

Option and the *arg*s determine the exact behavior of the command. The following commands are possible for messagedialog widgets:

INHERITED METHODS

add	**buttonconfigure**	**default**	**hide**
insert	**invoke**	**show**	

See the "buttonbox" widget manual entry for details on the above inherited methods.

childsite

See the "dialogshell" widget manual entry for details on the above inherited methods.

activate	**center**	**deactivate**

See the "dialogshell" widget manual entry for details on the above inherited methods.

WIDGET-SPECIFIC METHODS

pathName **cget** *option*

Returns the current value of the configuration option given by *option*. *Option* may have any of the values accepted by the **messagedialog** command.

pathName **configure** ?*option*? ?*value option value* ...?

Query or modify the configuration options of the widget. If no *option* is specified, returns a list describing all of the available options for *pathName* (see **Tk_ConfigureInfo** for information on the format of this list). If *option* is specified with no *value*, then the command returns a list describing the one named option (this list will be identical to the corresponding sublist of the value returned if no *option* is specified). If one or more *option−value* pairs are specified, then the command modifies the given widget option(s) to have the given value(s); in this case the command returns an empty string. *Option* may have any of the values accepted by the **messagedialog** command.

COMPONENTS

Name: **image**
Class: **Label**

The image component is the bitmap or image of the message dialog. See the "label" widget manual entry for details on the image component item.

Name: **message**
Class: **Label**

The message component provides the textual portion of the message dialog. See the "label" widget manual entry for details on the message component item.

EXAMPLE

```
#
# Standard question message dialog used for confirmation.
#
messagedialog .md -title "Message Dialog" -text "Are you sure ?" \
                                -bitmap questhead -modality global

.md buttonconfigure OK -text Yes
.md buttonconfigure Cancel -text No
```

```
if {[.md activate]} {
  .md configure -text "Are you really sure ?"
  if {[.md activate]} {
                                puts stdout "Yes"
  } else {
                                puts stdout "No"
  }
} else {
  puts stdout "No"
}

destroy .md

#
# Copyright notice with automatic deactivation.
#
messagedialog .cr -title "Copyright" -bitmap @dsc.xbm -imagepos n \
   -text "Copyright 1995 DSC Communications Corporation\n \
                                    All rights reserved"

.cr hide Cancel

.cr activate
after 10000 ".cr deactivate"
```

AUTHOR

Mark L. Ulferts

KEYWORDS

messagedialog, dialog, dialogshell, shell, widget

NAME

notebook − create and manipulate notebook widgets

SYNOPSIS

notebook *pathName* ?*options*?

INHERITANCE

itk::Widget <- notebook

STANDARD OPTIONS

background	**foreground**	**scrollCommand**	**width**
cursor	**height**		

See the "options" manual entry for details on the standard options.

WIDGET-SPECIFIC OPTIONS

Name: **auto**
Class: **Auto**
Command-Line Switch: **-auto**

Specifies whether to use the automatic packing/unpacking algorithm of the notebook. A value of **true** indicates that page frames will be unpacked and packed acoording to the algorithm described in the **select** command. A value of **false** leaves the current page packed and subsequent selects, next, or previous commands do not switch pages automatically. In either case the page's associated command (see the **add** command's description of the **command** option) is invoked. The value may have any of the forms accepted by the **Tcl_GetBoolean**, such as true, false, 0, 1, yes, or no.

For example, if a series of pages in a notebook simply change certain display configurations of a graphical display, the **-auto** flag could be used. By setting it, the **-command** procs could do the appropriate reconfiguring of the page when the page is switched.

DESCRIPTION

The **notebook** command creates a new window (given by the pathName argument) and makes it into a notebook widget. Additional options, described above may be specified on the command line or in the option database to configure aspects of the notebook such as its colors, font, and text. The **notebook** command returns its *pathName* argument. At the time this command is invoked, there must not exist a window named pathName, but pathName's parent must exist.

A notebook is a widget that contains a set of pages. It displays one page from the set as the selected page. When a page is selected, the page's contents are displayed in the page area. When first created a notebook has no pages. Pages may be added or deleted using widget commands described below.

NOTEBOOK PAGES

A notebook's pages area contains a single child site **frame**. When a new page is created it is a child of this frame. The page's child site frame serves as a geometry container for applications to pack widgets into. It is this frame that is automatically unpacked or packed when the **auto** option is **true**. This creates the effect of one page being visible at a time. When a new page is selected, the previously selected page's child site frame is automatically unpacked from the notebook's child site frame and the newly selected page's child site is packed into the notebook's child site frame.

However, sometimes it is desirable to handle page changes in a different manner. By specifying the **auto** option as **false**, child site packing can be disabled and done differently. For example, all widgets might be packed into the first page's child site frame. Then when a new page is selected, the application can reconfigure the widgets and give the appearance that the page was flipped.

In both cases the **command** option for a page specifies a Tcl Command to execute when the page is selected. In the case of **auto** being **true**, it is called between the unpacking of the previously selected page and the packing of the newly selected page.

WIDGET-SPECIFIC METHODS

The **notebookfR command creates a new Tcl command whose name is** *pathName*. This command may be used to invoke various operations on the widget. It has the following general form:

pathName option ?*arg arg ...*?

option and the *arg*s determine the exact behavior of the command.

Many of the widget commands for a notebook take as one argument an indicator of which page of the notebook to operate on. These indicators are called indexes and may be specified in any of the following forms:

number Specifies the index of the the component. For menus, 0 corresponds to the left-most menu of the menu bar. For entries, 0 corresponds to the top-most entry of the menu. *number* Specifies the page numerically, where 0 corresponds to the first page in the notebook, 1 to the second, and so on.

select Specifies the currently selected page's index. If no page is currently selected, the value -1 is returned.

end Specifes the last page in the notebooks's index. If the notebook is empty this will return -1.

pattern If the index doesn't satisfy the form of a number, then this form is used. Pattern is pattern-matched against the **label** of each page in the notebook, in order from the first to the last page, until a matching entry is found. The rules of **Tcl_StringMatch** are used.

The following commands are possible for notebook widgets:

pathName **add** ?*option value*?

Add a new page at the end of the notebook. A new child site frame is created. Returns the child site pathName. If additional arguments are present, they specify any of the following options:

-background *value*

Specifies a background color to use for displaying the child site frame of this page. If this option is specified as an empty string (the default), then the background option for the overall notebook is used.

-command *value*

Specifies a Tcl command to be executed when this page is selected. This allows the programmer a hook to reconfigure this page's widgets or any other page's widgets.

If the notebook has the auto option set to true, when a page is selected this command will be called immediately after the previously selected page is unpacked and immediately before this page is selected. The index value select is valid during this Tcl command. 'index select' will return this page's page number.

If the auto option is set to false, when a page is selected the unpack and pack calls are bypassed. This Tcl command is still called.

-foreground *value*

Specifies a foreground color to use for displaying tab labels when tabs are in their normal unselected state. If this option is specified as an empty string (the default), then the foreground option for the overall notebook is used.

-label *value*

Specifies a string to associate with this page. This label serves as an additional identifier used to reference the page. This label may be used for the index value in widget

commands.

pathName **childSite** ?*index*?

If passed no arguments, returns a list of pathNames for all the pages in the notebook. If the notebook is empty, an empty list is returned

If index is passed, it returns the pathName for the page's child site frame specified by index. Widgets that are created with this pathName will be displayed when the associated page is selected. If index is not a valid index, an empty string is returned.

pathName **cget** *option*

Returns the current value of the configuration option given by *option*.

pathName **configure** ?*option*? ?*value option value* ...?

Query or modify the configuration options of the widget. If no *option* is specified, returns a list describing all of the available options for *pathName* (see **Tk_ConfigureInfo** for information on the format of this list). If *option* is specified with no *value*, then the command returns a list describing the one named option (this list will be identical to the corresponding sublist of the value returned if no option is specified). If one or more option-value pairs are specified, then the command modifies the given widget option(s) to have the given value(s); in this case the command returns an empty string. *Option* may have any of the values accepted by the **notebook** command.

pathName **delete** *index1* ?index2?

Delete all of the pages between *index1* and *index2* inclusive. If *index2* is omitted then it defaults to *index1*. Returns an empty string.

pathName **index** *index*

Returns the numerical index corresponding to *index*.

pathName insert *index* ?*option value*?

Insert a new page in the notebook before the page specified by *index*. A new child site **frame** is created. See the **add** command for valid options. Returns the child site pathName.

pathName **next**

Advances the selected page to the next page (order is determined by insertion order). If the currently selected page is the last page in the notebook, the selection wraps around to the first page in the notebook.

For notebooks with auto set to true the current page's child site is unpacked from the notebook's child site frame. Then the next page's child site is packed into the notebooks child site frame. The Tcl command given with the command option will be invoked between these two operations.

For notebooks with auto set to false the Tcl command given with the command option will be invoked.

pathName **pagecget** *index* ?*option*?

Returns the current value of the configuration option given by *option* for the page specified by *index*. The valid available options are the same as available to the **add** command.

pathName **pageconfigure** *index* ?*option*? ?*value option value* ...?

This command is similar to the configure command, except that it applies to the options for an individual page, whereas configure applies to the options for the notebook. Options may have any of the values accepted by the add widget command. If options are specified, options are modified as indicated in the command and the command returns an empty string. If no options are specified, returns a list describing the current options for page *index* (see **Tk_ConfigureInfo** for information on the format of this list).

pathName **prev**

Moves the selected page to the previous page (order is determined by insertion order). If the

currently selected page is the first page in the notebook, the selection wraps around to the last page in the notebook.

For notebooks with **auto** set to **true** the current page's child site is unpacked from the notebook's child site frame. Then the previous page's child site is packed into the notebooks child site frame. The Tcl command given with the command option will be invoked between these two operations.

For notebooks with **auto** set to **false** the Tcl command given with the command option will be invoked.

pathName **select** *index*
Selects the page specified by *index* as the currently selected page.

For notebooks with **auto** set to **true** the current page's child site is unpacked from the notebook's child site frame. Then the index page's child site is packed into the notebooks child site frame. The Tcl command given with the command option will be invoked between these two operations.

For notebooks with **auto** set to **false** the Tcl command given with the command option will be invoked.

pathName **view**
Returns the currently selected page. This command is for compatibility with the scrollbar widget.

pathName **view** *index*
Selects the page specified by *index* as the currently selected page. This command is for compatibility with the scrollbar widget.

pathName **view** *moveto fraction*
Uses the fraction value to determine the corresponding page to move to. This command is for compatibility with the scrollbar widget.

pathName **view** *scroll num what*
Uses the *num* value to determine how many pages to move forward or backward (num can be negative or positive). The *what* argument is ignored. This command is for compatibility with the scrollbar widget.

EXAMPLE

Following is an example that creates a notebook with two pages. In this example, we use a scrollbar widget to control the notebook widget.

```
# Create the notebook widget and pack it.
  notebook .nb -width 100 -height 100
  pack .nb -anchor nw \
                -fill both \
                -expand yes \
                -side left \
                -padx 10 \
                -pady 10

# Add two pages to the notebook, labelled
# "Page One" and "Page Two", respectively.
  .nb add -label "Page One"
  .nb add -label "Page Two"

# Get the child site frames of these two pages.
  set page1CS [.nb childsite 0]
  set page2CS [.nb childsite "Page Two"]

# Create buttons on each page of the notebook
```

```
  button $page1CS.b -text "Button One"
  pack $page1CS.b
  button $page2CS.b -text "Button Two"
  pack $page2CS.b

# Select the first page of the notebook
  .nb select 0

# Create the scrollbar and associate teh scrollbar
# and the notebook together, then pack the scrollbar
  ScrollBar .scroll -command ".nb view"
  .nb configure -scrollcommand ".scroll set"
  pack .scroll -fill y -expand yes -pady 10
```

AUTHOR

Bill W. Scott

KEYWORDS

notebook page

NAME

optionmenu − Create and manipulate a option menu widget

SYNOPSIS

optionmenu *pathName* ?*options*?

INHERITANCE

itk::Widget <- Labeledwidget <- optionmenu

STANDARD OPTIONS

activeBackground	**activeBorderWidth**	**activeForeground**	**background**
borderWidth	**cursor**	**disabledForeground**	**font**
foreground	**highlightColor**	**highlightThickness**	**relief**

See the "options" manual entry for details on the standard options.

INHERITED OPTIONS

disabledForeground	**labelBitmap**	**labelFont**	**labelImage**
labelMargin	**labelPos**	**labelText**	**labelVariable**
state			

See the "LabeledWidget" manual entry for details on the inherited options.

WIDGET-SPECIFIC OPTIONS

Name: **clickTime**
Class: **ClickTime**
Command-Line Switch: **-clicktime**

Interval time, in msec, used to determine that a single mouse click has occurred. Used to post menu on a "quick" mouse click. **Note**: changing this value may cause the sigle-click functionality to not work properly. The default is 150 msec.

Name: **command**
Class: **Command**
Command-Line Switch: **-command**

Specifies a Tcl command procedure to be evaluated following a change in the current option menu selection.

Name: **cyclicOn**
Class: **CyclicOn**
Command-Line Switch: **-cyclicon**

Turns on/off the 3rd mouse button capability. The value may be specified in any of the forms acceptable to **Tcl_GetBoolean**. This feature allows the right mouse button to cycle through the popup menu list without poping it up. The right mouse button cycles through the menu in reverse order. The default is true.

Name: **popupCursor**
Class: **Cursor**
Command-Line Switch: **-popupcursor**

Specifies the mouse cursor to be used for the popup menu. The value may have any of the forms acceptable to **Tk_GetCursor**.

Name: **state**
Class: **State**
Command-Line Switch: **-state**

Specified one of two states for the optionmenu: **normal**, or **disabled**. If the optionmenu is disabled, then option menu selection is ignored.

Name: **width**
Class: **Width**
Command-Line Switch: **-width**

Specifies a fixed size for the menu button label in any of the forms acceptable to Tk_GetPixels. If the text is too small to fit in the label, the text is clipped. Note: Normally, when a new list is created, or new items are added to an existing list, the menu button label is resized automatically. Setting this option overrides that functionality.

DESCRIPTION

The **optionmenu** command creates an option menu widget with options to manage it. An option menu displays a frame containing a label and a button. A pop-up menu will allow for the value of the button to change.

METHODS

The **optionmenu** command creates a new Tcl command whose name is *pathName*. This command may be used to invoke various operations on the widget. It has the following general form:

pathName option ?*arg arg ...*?

Option and the *arg*s determine the exact behavior of the command.

Many of the widget commands for an optionmenu take as one argument an indicator of which entry of the option menu to operate on. These indicators are called *index*es and may be specified in any of the following forms:

number Specifies the entry numerically, where 0 corresponds to the top-most entry of the option menu, 1 to the entry below it, and so on.

end Indicates the bottommost entry in the menu. If there are no entries in the menu then zero is returned.

select Returns the numerical index of the currently selected option menu entry. If no entries exist in the menu, then -1 is returned.

pattern If the index doesn't satisfy one of the above forms then this form is used. *Pattern* is pattern-matched against the label of each entry in the option menu, in order from the top down, until a matching entry is found. The rules of **Tcl_StringMatch** are used.

The following widget commands are possible for optionmenu widgets:

WIDGET-SPECIFIC METHODS

pathName **cget** *option*

Returns the current value of the configuration option given by *option*. *Option* may have any of the values accepted by the **optionmenu** command.

pathName **configure** ?*option*? ?*value option value ...*?

Query or modify the configuration options of the widget. If no *option* is specified, returns a list describing all of the available options for *pathName* (see **Tk_ConfigureInfo** for information on the format of this list). If *option* is specified with no *value*, then the command returns a list describing the one named option (this list will be identical to the corresponding sublist of the value returned if no *option* is specified). If one or more *option−value* pairs are specified, then the command modifies the given widget option(s) to have the given value(s); in this case the command returns an empty string. *Option* may have any of the values accepted by the **optionmenu**

command.

pathName **delete** *first* ?*last*?
Delete all of the option menu entries between *first* and *last* inclusive. If *last* is omitted then it defaults to *first*.

pathName **disable** *index*
Disable the option menu entry specified by *index*. Disabling a menu item will prevent the user from being able to select this item from the menu. This only effects the state of the item in the menu, in other words, should the item be the currently selected item, the programmer is responsible for determining this condition and taking appropriate action.

pathName **enable** *index*
Enable the option menu entry specified by *index*. Enabling a menu item allows the user to select this item from the menu.

pathName **get** ?*first*? ?*last*?
If no arguments are specified, this operation returns the currently selected option menu item. Otherwise, it returns the name of the option at index *first*, or a range of options between *first* and *last*.

pathName **index** *index*
Returns the numerical index corresponding to *index*.

pathName **insert** *index string* ?*string*?
Insert an item, or list of items, into the menu at location *index*.

pathName **select** *index*
Select an item from the option menu to be displayed as the currently selected item.

pathName **sort** *mode*
Sort the current menu in either **ascending**, or **descending** order. The values **increasing**, or **decreasing** are also accepted.

COMPONENTS

Name: **menuBtn**
Class: **Menubutton**

The menuBtn component is the option menu button which displays the current choice from the popup menu. See the "menubutton" widget manual entry for details on the menuBtn component item.

Name: **popupMenu**
Class: **Menu**

The popupMenu component is menu displayed upon selection of the menu button. The menu contains the choices for the option menu. See the "menu" widget manual entry for details on the popupMenu component item.

EXAMPLE

```
optionmenu .om -labelmargin 5 \
   -labelon true -labelpos w -labeltext "Operating System :"

.om insert end Unix VMS Linux OS/2 {Windows NT} DOS
.om sort ascending
.om select Linux

pack .om  -padx 10 -pady 10
```

ACKNOWLEDGEMENTS:

Michael J. McLennan

Borrowed some ideas (next & previous) from OptionButton class.

Steven B. Jaggers

Provided an initial prototype in [incr Tcl].

Bret Schuhmacher

Helped with popup menu functionality.

AUTHOR

Alfredo Jahn

KEYWORDS

optionmenu, widget

NAME

panedwindow − Create and manipulate a paned window widget

SYNOPSIS

panedwindow *pathName* ?*options*?

INHERITANCE

itk::Widget <- panedwindow

STANDARD OPTIONS

background **cursor**

See the "options" manual entry for details on the standard options.

WIDGET-SPECIFIC OPTIONS

Name: **height**
Class: **Height**
Command-Line Switch: **-height**

Specifies the overall height of the paned window in any of the forms acceptable to **Tk_GetPixels**. The default is 10 pixels.

Name: **orient**
Class: **Orient**
Command-Line Switch: **-orient**

Specifies the orientation of the separators: **vertical** or **horizontal**. The default is horizontal.

Name: **sashBorderWidth**
Class: **BorderWidth**
Command-Line Switch: **-sashborderwidth**

Specifies a value indicating the width of the 3-D border to draw around the outside of the sash in any of the forms acceptable to **Tk_GetPixels**. The default is 2 pixels.

Name: **sashCursor**
Class: **Cursor**
Command-Line Switch: **-sashcursor**

Specifies the type of cursor to be displayed in the sash. The default is crosshair.

Name: **sashHeight**
Class: **Height**
Command-Line Switch: **-sashheight**

Specifies the height of the sash in any of the forms acceptable to **Tk_GetPixels**. The default is 10 pixels.

Name: **sashIndent**
Class: **SashIndent**
Command-Line Switch **sashindent**

Specifies the placement of the sash along the panes in any of the forms acceptable to **Tk_GetPixels**. A positive value causes the sash to be offset from the near (left/top) side of the pane, and a negative value causes the sash to be offset from the far (right/bottom) side. If the offset is greater than the width, then the sash is placed flush against the side. The default is -10 pixels.

Name: **sashWidth**
Class: **Width**
Command-Line Switch: **-sashwidth**

Specifies the width of the sash in any of the forms acceptable to **Tk_GetPixels**. The default is 10

pixels.

Name: **thickness**
Class: **Thickness**
Command-Line Switch: **-thickness**

Specifies the thickness of the separators in any of the forms acceptable to **Tk_GetPixels**. The default is 3 pixels.

Name: **width**
Class: **Width**
Command-Line Switch: **-width**

Specifies the overall width of the paned window in any of the forms acceptable to **Tk_GetPixels**. The default is 10 pixels.

DESCRIPTION

The **panedwindow** command creates a multiple paned window widget capable of orienting the panes either vertically or horizontally. Each pane is itself a frame acting as a child site for other widgets. The border separating each pane contains a sash which allows user positioning of the panes relative to one another.

METHODS

The **panedwindow** command creates a new Tcl command whose name is *pathName*. This command may be used to invoke various operations on the widget. It has the following general form:

pathName option ?*arg arg ...*?

Option and the *arg*s determine the exact behavior of the command.

Many of the widget commands for the **panedwindow** take as one argument an indicator of which pane of the paned window to operate on. These indicators are called *indexes* and allow reference and manipulation of panes regardless of their current map state. Paned window indexes may be specified in any of the following forms:

number Specifies the pane numerically, where 0 corresponds to the nearest (top/left-most) pane of the paned window.

end Indicates the farthest (bottom/right-most) pane of the paned window.

pattern If the index doesn't satisfy one of the above forms then this form is used. *Pattern* is pattern-matched against the tag of each pane in the panedwindow, in order from left/top to right/left, until a matching entry is found. The rules of **Tcl_StringMatch** are used.

WIDGET-SPECIFIC METHODS

pathName **add** *tag* ?*option value option value*?

Adds a new pane to the paned window on the far side (right/bottom). The following options may be specified:

-margin *value*

Specifies the border distance between the pane and pane contents is any of the forms acceptable to **Tk_GetPixels**. The default is 8 pixels.

-minimum *value*

Specifies the minimum size that a pane's contents may reach not inclusive of twice the margin in any of the forms acceptable to **Tk_GetPixels**. The default is 10 pixels.

The **add** method returns the path name of the pane.

pathName **cget** *option*
Returns the current value of the configuration option given by *option*. *Option* may have any of the values accepted by the **panedwindow** command.

pathName **childsite** ?*index*?
Returns a list of the child site path names or a specific child site given an index. The list is constructed from the near side (left/top) to the far side (right/bottom).

pathName **configure** ?*option*? ?*value option value ...*?
Query or modify the configuration options of the widget. If no *option* is specified, returns a list describing all of the available options for *pathName* (see **Tk_ConfigureInfo** for information on the format of this list). If *option* is specified with no *value*, then the command returns a list describing the one named option (this list will be identical to the corresponding sublist of the value returned if no *option* is specified). If one or more *option−value* pairs are specified, then the command modifies the given widget option(s) to have the given value(s); in this case the command returns an empty string. *Option* may have any of the values accepted by the **panedwindow** command.

pathName **delete** *index*
Deletes a specified pane given an *index*.

pathName **fraction** *percentage percentage* ?*percentage percentage ...*?
Sets the visible percentage of the panes. Specifies a set of percentages which are applied to the visible panes from the near side (left/top). The number of percentages must be equal to the current number of visible (mapped) panes and add up to 100.

pathName **hide** *index*
Changes the visiblity of the specified pane, allowing a previously displayed pane to be visually removed rather than deleted.

pathName **index** *index*
Returns the numerical index corresponding to index.

pathName **insert** *index tag* ?*option value option value ...*?
Same as the **add** command except that it inserts the new pane just before the one given by *index*, instead of appending to the end of the panedwindow. The *option*, and *value* arguments have the same interpretation as for the **add** widget command.

pathName **paneconfigure** *index* ?*options*?
This command is similar to the **configure** command, except that it applies to the options for an individual pane, whereas **configure** applies to the options for the paned window as a whole. *Options* may have any of the values accepted by the **add** widget command. If *options* are specified, options are modified as indicated in the command and the command returns an empty string. If no *options* are specified, returns a list describing the current options for entry *index* (see **Tk_ConfigureInfo** for information on the format of this list).

pathName **reset**
Redisplays the pane window using default percentages.

pathName **show** *index*
Changes the visiblity of the specified pane, allowing a previously hidden pane to be displayed.

NOTES

Dynamic changing of the margin and or minimum options to values which make the current configuration invalid will block subsequent sash movement until the fractions are modified via the fraction method. For example a panedwindow is created with three panes and the minimum and

margin options are at their default settings. Next the user moves the sashes to compact the panes to one side. Now, if the minimum is increased on the most compressed pane via the paneconfigure method to a large enough value, then sash movement is blocked until the fractions are adjusted. This situation is unusual and under normal operation of the panedwindow, this problem will never occur.

EXAMPLE

```
panedwindow .pw -width 300 -height 300
.pw add top
.pw add middle -margin 10
.pw add bottom -margin 10 -minimum 10

pack .pw -fill both -expand yes

foreach pane [.pw childSite] {
  button $pane.b -text $pane -relief raised -borderwidth 2
  pack $pane.b -fill both -expand yes
}

.pw fraction 50 30 20
.pw paneconfigure 0 -minimum 20
.pw paneconfigure bottom -margin 15
```

ACKNOWLEDGEMENTS:

Jay Schmidgall

1994 - Base logic posted to comp.lang.tcl

Joe Hidebrand <hildjj@fuentez.com>

07/25/94 - Posted first multipane version to comp.lang.tcl

07/28/94 - Added support for vertical panes

Ken Copeland <ken@hilco.com>

09/28/95 - Smoothed out the sash movement and added squeezable panes.

AUTHOR

Mark L. Ulferts

KEYWORDS

panedwindow, widget

NAME

promptdialog − Create and manipulate a prompt dialog widget

SYNOPSIS

promptdialog *pathName* ?*options*?

INHERITANCE

itk::Toplevel <- dialogshell <- dialog <- promptdialog

STANDARD OPTIONS

background	**borderWidth**	**cursor**	**exportSelection**
foreground	**highlightColor**	**highlightThickness**	**insertBackground**
insertBorderWidth	**insertOffTime**	**insertOnTime**	**insertWidth**
relief	**selectBackground**	**selectBorderWidth**	**selectForeground**

See the "options" manual entry for details on the standard options.

ASSOCIATED OPTIONS

show

See the "entry" widget manual entry for details on the above associated options.

invalid	**textBackground**	**textFont**	**validate**

See the "entryfield" widget manual entry for details on the above associated options.

labelFont	**labelPos**	**labelText**

See the "labeledwidget" widget manual entry for details on the above associated options.

INHERITED OPTIONS

buttonBoxPadX	**buttonBoxPadY**	**buttonBoxPos**	**padX**
padY	**separator**	**thickness**	

See the "dialogshell" widget manual entry for details on the above inherited options.

height	**master**	**modality**	**width**

See the "shell" widget manual entry for details on the above inherited options.

title

See the "Toplevel" widget manual entry for details on the above inherited options.

DESCRIPTION

The **promptdialog** command creates a prompt dialog similar to the OSF/Motif standard prompt dialog composite widget. The promptdialog is derived from the dialog class and is composed of a EntryField with commands to manipulate the dialog buttons.

METHODS

The **promptdialog** command creates a new Tcl command whose name is *pathName*. This command may be used to invoke various operations on the widget. It has the following general form:

pathName option ?*arg arg* ...?

Option and the *arg*s determine the exact behavior of the command. The following commands are possible for promptdialog widgets:

ASSOCIATED METHODS

delete	**get**	**icursor**	**index**
insert	**scan**	**selection**	**xview**

See the "entry" widget manual entry for details on the above associated methods.

clear

See the "entryfield" widget manual entry for details on the above associated methods.

INHERITED METHODS

add	**buttonconfigure**	**default**	**hide**
invoke	**show**		

See the "buttonbox" widget manual entry for details on the above inherited methods.

childsite

See the "dialogshell" widget manual entry for details on the above inherited methods.

activate	**center**	**deactivate**

See the "shell" widget manual entry for details on the above inherited methods.

WIDGET-SPECIFIC METHODS

pathName **cget** *option*

Returns the current value of the configuration option given by *option*. *Option* may have any of the values accepted by the **promptdialog** command.

pathName **configure** ?*option*? ?*value option value ...*?

Query or modify the configuration options of the widget. If no *option* is specified, returns a list describing all of the available options for *pathName* (see **Tk_ConfigureInfo** for information on the format of this list). If *option* is specified with no *value*, then the command returns a list describing the one named option (this list will be identical to the corresponding sublist of the value returned if no *option* is specified). If one or more *option−value* pairs are specified, then the command modifies the given widget option(s) to have the given value(s); in this case the command returns an empty string. *Option* may have any of the values accepted by the **promptdialog** command.

COMPONENTS

Name:	**prompt**
Class:	**Entryfield**

The prompt component is the entry field for user input in the prompt dialog. See the "entryfield" widget manual entry for details on the prompt component item.

EXAMPLE

```
option add *textBackground white

promptdialog .pd -modality global -title Password -labeltext Password: -show *
.pd hide Apply

if {[.pd activate]} {
   puts "Password entered: [.pd get]"
} else {
   puts "Password prompt cancelled"
}
```

AUTHOR

Mark L. Ulferts

KEYWORDS

promptdialog, dialog, dialogshell, shell, widget

NAME

pushbutton − Create and manipulate a push button widget

SYNOPSIS

pushbutton *pathName* ?*options*?

INHERITANCE

itk::Widget <- pushbutton

STANDARD OPTIONS

activeBackground	**activeForeground**	**background**	**bitmap**
borderWidth	**command**	**cursor**	**disabledForeground**
font	**foreground**	**highlightBackground**	**highlightColor**
highlightThickness	**image**	**padX**	**padY**
state	**text**	**underline**	**wrapLength**

See the "options" manual entry for details on the standard options.

WIDGET-SPECIFIC OPTIONS

Name: **defaultRing**
Class: **DefaultRing**
Command-Line Switch: **-defaultring**

Boolean describing whether the button displays its default ring given in any of the forms acceptable to **Tcl_GetBoolean**. The default is false.

Name: **defaultRingPad**
Class: **Pad**
Command-Line Switch: **-defaultringpad**

Specifies the amount of space to be allocated to the indentation of the default ring ring given in any of the forms acceptable to **Tcl_GetPixels**. The option has no effect if the defaultring option is set to false. The default is 2 pixels.

Name: **height**
Class: **Height**
Command-Line Switch: **-height**

Specifies the height of the button inclusive of any default ring given in any of the forms acceptable to **Tk_GetPixels**. A value of zero lets the push button determine the height based on the requested height plus highlightring and defaultringpad.

Name: **width**
Class: **Width**
Command-Line Switch: **-width**

Specifies the width of the button inclusive of any default ring given in any of the forms acceptable to **Tk_GetPixels**. A value of zero lets the push button determine the width based on the requested width plus highlightring and defaultringpad.

DESCRIPTION

The **pushbutton** command creates a push button with an optional default ring used for default designation and traversal.

METHODS

The **pushbutton** command creates a new Tcl command whose name is *pathName*. This command may be used to invoke various operations on the widget. It has the following general form:

pathName option ?*arg arg ...*?

Option and the *arg*s determine the exact behavior of the command. The following commands are possible for pushbutton widgets:

ASSOCIATED METHODS

flash **invoke**

See the "button" manual entry for details on the associated methods.

WIDGET-SPECIFIC METHODS

pathName **cget** *option*

Returns the current value of the configuration option given by *option*. *Option* may have any of the values accepted by the **pushbutton** command.

pathName **configure** ?*option*? ?*value option value ...*?

Query or modify the configuration options of the widget. If no *option* is specified, returns a list describing all of the available options for *pathName* (see **Tk_ConfigureInfo** for information on the format of this list). If *option* is specified with no *value*, then the command returns a list describing the one named option (this list will be identical to the corresponding sublist of the value returned if no *option* is specified). If one or more *option−value* pairs are specified, then the command modifies the given widget option(s) to have the given value(s); in this case the command returns an empty string. *Option* may have any of the values accepted by the **pushbutton** command.

COMPONENTS

Name: **pushbutton**
Class: **Button**

The pushbutton component is the button surrounded by the optional default ring. See the "button" widget manual entry for details on the pushbutton component item.

EXAMPLE

```
pushbutton .pb -text "Hello" -command {puts "Hello World"} -defaultring 1
pack .pb -padx 10 -pady 10
```

AUTHOR

Bret A. Schuhmacher

Mark L. Ulferts

KEYWORDS

pushbutton, widget

NAME

radiobox − Create and manipulate a radiobox widget

SYNOPSIS

radiobox *pathName* ?*options*?

INHERITANCE

itk::Widget <- labeledframe <- radiobox

STANDARD OPTIONS

background	**borderWidth**	**cursor**	**disabledForeground**
foreground	**relief**	**selectColor**	

See the "options" manual entry for details on the standard options.

INHERITED OPTIONS

labelBitmap	**labelFont**	**labelImage**	**labelMargin**
labelPos	**labelText**	**labelVariable**	

See the "labeledframe" class manual entry for details on the inherited options.

WIDGET-SPECIFIC OPTIONS

Name: **command**
Class: **Command**
Command-Line Switch: **-command**

Specifies a Tcl command procedure to be evaluated following a change in the current radio box selection.

DESCRIPTION

The **radiobox** command creates a radio button box widget capable of adding, inserting, deleting, selecting, and configuring radiobuttons as well as obtaining the currently selected button.

METHODS

The **radiobox** command creates a new Tcl command whose name is *pathName*. This command may be used to invoke various operations on the widget. It has the following general form:

pathName option ?*arg arg ...*?

Option and the *arg*s determine the exact behavior of the command.

Many of the widget commands for the **radiobox** take as one argument an indicator of which radiobutton of the radiobox to operate on. These indicators are called *indexes* and allow reference and manipulation of radiobuttons. Radiobox indexes may be specified in any of the following forms:

number Specifies the radiobutton numerically, where 0 corresponds to the top radiobutton of the radiobox.

end Indicates the last radiobutton of the radiobox.

pattern If the index doesn't satisfy one of the above forms then this form is used. *Pattern* is pattern-matched against the tag of each radiobutton in the radiobox, in order from top to bottom, until a matching entry is found. The rules of **Tcl_StringMatch** are used.

WIDGET-SPECIFIC METHODS

pathName **add** *tag* ?*option value option value*?

Adds a new radiobutton to the radiobuttond window on the bottom. The command takes additional options which are passed on to the radiobutton as construction arguments. These include the standard Tk radiobutton options. The tag is returned.

pathName **buttonconfigure** *index* ?*options*?

This command is similar to the **configure** command, except that it applies to the options for an individual radiobutton, whereas **configure**applies to the options for the radiobox as a whole. *Options* may have any of the values accepted by the **add** widget command. If *options* are specified, options are modified as indicated in the command and the command returns an empty string. If no *options* are specified, returns a list describing the current options for entry *index* (see **Tk_ConfigureInfo** for information on the format of this list).

pathName **cget** *option*

Returns the current value of the configuration option given by *option*. *Option* may have any of the values accepted by the **radiobox** command.

pathName **configure** ?*option*? ?*value option value ...*?

Query or modify the configuration options of the widget. If no *option* is specified, returns a list describing all of the available options for *pathName* (see **Tk_ConfigureInfo** for information on the format of this list). If *option* is specified with no *value*, then the command returns a list describing the one named option (this list will be identical to the corresponding sublist of the value returned if no *option* is specified). If one or more *option−value* pairs are specified, then the command modifies the given widget option(s) to have the given value(s); in this case the command returns an empty string. *Option* may have any of the values accepted by the **radiobox** command.

pathName **delete** *index*

Deletes a specified radiobutton given an *index*.

pathName **deselect** *index*

Deselects a specified radiobutton given an *index*.

pathName **flash** *index*

Flashes a specified radiobutton given an *index*.

pathName **get**

Returns the tag of the currently selected radiobutton.

pathName **index** *index*

Returns the numerical index corresponding to index.

pathName **insert** *index tag* ?*option value option value ...*?

Same as the **add** command except that it inserts the new radiobutton just before the one given by *index*, instead of appending to the end of the radiobox. The *option*, and *value* arguments have the same interpretation as for the **add** widget command.

pathName **select** *index*

Selects a specified radiobutton given an *index*.

EXAMPLE

```
radiobox .rb -labeltext Fonts
.rb add times -text Times
.rb add helvetica -text Helvetica
.rb add courier -text Courier
.rb add symbol -text Symbol
.rb select courier
```

```
pack .rb -padx 10 -pady 10 -fill both -expand yes
```

AUTHOR

Michael J. McLennan

Mark L. Ulferts

KEYWORDS

radiobox, widget

NAME

scopedobject − Create and manipulate a scoped [incr Tcl] class object.

SYNOPSIS

scopedobject *objName* ?*options*?

INHERITANCE

None

STANDARD OPTIONS

Name: **enterscopecommand:**
Command-Line Switch: **-enterscopecommand**

Specifies a Tcl command to invoke when an object enters scope (i.e. when it is created..). The default is {}.

Name: **enterscopecommand:**
Command-Line Switch: **-enterscopecommand**

Specifies a Tcl command to invoke when an object exits scope (i.e. when it is deleted..). The default is {}.

DESCRIPTION

The **scopedobject** command creates a base class for defining Itcl classes which posses scoped behavior like Tcl variables. The objects are only accessible within the procedure in which they are instantiated and are deleted when the procedure returns. This class was designed to be a general purpose base class for supporting scoped incr Tcl classes. The options include the execute a Tcl script command when an object enters and exits its scope.

METHODS

The **scopedobject** command creates a new Tcl command whose name is *pathName*. This command may be used to invoke various operations on the object. It has the following general form:

pathName option ?*arg arg ...*?

Option and the *arg*s determine the exact behavior of the command. The following commands are possible for scopedobject objects:

OBJECT-SPECIFIC METHODS

pathName **cget** *option*

Returns the current value of the configuration option given by *option*. *Option* may have any of the values accepted by the **scopedobject** command.

pathName **configure** ?*option*? ?*value option value ...*?

Query or modify the configuration options of the object. If no *option* is specified, returns a list describing all of the available options for *pathName*. If *option* is specified with no *value*, then the command returns a list describing the one named option (this list will be identical to the corresponding sublist of the value returned if no *option* is specified). If one or more *option−value* pairs are specified, then the command modifies the given objects option(s) to have the given value(s); in this case the command returns an empty string. *Option* may have any of the values accepted by the **scopedobject** command.

EXAMPLE

The scopedobject was primarily meant to be a base class. The following is an example of usage without inheritance:

```
proc scopedobject_demo {} {
  scopedobject #auto        -exitscopecommand {puts "enter scopedobject_demo"}        -exitscopecommand {puts "e
}

scopedobject_demo
```

AUTHOR

John A. Tucker

KEYWORDS

scopedobject, object

NAME

scrolledcanvas − Create and manipulate scrolled canvas widgets

SYNOPSIS

scrolledcanvas *pathName* ?*options*?

INHERITANCE

itk::Widget <- Labeledwidget <- Scrolledwidget <- Scrolledcanvas

STANDARD OPTIONS

activeBackground	**background**	**borderWidth**	**cursor**
exportSelection	**font**	**foreground**	**highlightColor**
highlightThickness	**insertBorderWidth**	**insertOffTime**	**insertOnTime**
insertWidth	**relief**	**selectBackground**	**selectBorderWidth**
selectForeground			

See the "options" manual entry for details on the standard options.

ASSOCIATED OPTIONS

closeEnough	**confine**	**scrollRegion**	**xScrollIncrement**
yScrollIncrement			

See the "canvas" widget manual entry for details on the above associated options.

activeRelief	**elementBorderWidth**	**jump**	**troughColor**

See the "scrollbar" widget manual entry for details on the above associated options.

INHERITED OPTIONS

disabledForeground	**labelBitmap**	**labelFont**	**labelImage**
labelMargin	**labelPos**	**labelText**	**labelVariable**
state			

See the "labeledwidget" class manual entry for details on the inherited options.

WIDGET-SPECIFIC OPTIONS

Name: **autoMargin**
Class: **AutoMargin**
Command-Line Switch: **-automargin**

Specifies the autoresize extra margin to reserve. This option is only effective with autoresize turned on. The default is 10.

Name: **autoResize**
Class: **AutoResize**
Command-Line Switch: **-autoresize**

Automatically adjusts the scrolled region to be the bounding box covering all the items in the canvas following the execution of any method which creates or destroys items. Thus, as new items are added, the scrollbars adjust accordingly.

Name: **height**
Class: **Height**
Command-Line Switch: **-height**

Specifies the height of the scrolled canvas widget in any of the forms acceptable to **Tk_GetPixels**. The default height is 30 pixels.

Name: **hscrollMode**
Class: **ScrollMode**
Command-Line Switch: **-hscrollmode**

Specifies the the display mode to be used for the horizontal scrollbar: **static, dynamic,** or **none**. In static mode, the scroll bar is displayed at all times. Dynamic mode displays the scroll bar as required, and none disables the scroll bar display. The default is static.

Name: **sbWidth**
Class: **Width**
Command-Line Switch: **-sbwidth**

Specifies the width of the scrollbar in any of the forms acceptable to **Tk_GetPixels**. The default width is 15 pixels..

Name: **scrollMargin**
Class: **ScrollMargin**
Command-Line Switch: **-scrollmargin**

Specifies the distance between the canvas and scrollbar in any of the forms acceptable to **Tk_GetPixels**. The default is 3 pixels.

Name: **textBackground**
Class: **Background**
Command-Line Switch **-textbackground**

Specifies the background color for the canvas. This allows the background within the canvas to be different from the normal background color.

Name: **vscrollMode**
Class: **ScrollMode**
Command-Line Switch: **-vscrollmode**

Specifies the the display mode to be used for the vertical scrollbar: **static, dynamic,** or **none**. In static mode, the scroll bar is displayed at all times. Dynamic mode displays the scroll bar as required, and none disables the scroll bar display. The default is static.

Name: **width**
Class: **Width**
Command-Line Switch: **-width**

Specifies the width of the scrolled canvas widget in any of the forms acceptable to **Tk_GetPixels**. The default height is 30 pixels.

DESCRIPTION

The **scrolledcanvas** command creates a scrolled canvas with additional options to manage horizontal and vertical scrollbars. This includes options to control which scrollbars are displayed and the method, i.e. statically or dynamically.

METHODS

The **scrolledcanvas** command creates a new Tcl command whose name is *pathName*. This command may be used to invoke various operations on the widget. It has the following general form:

pathName option ?*arg arg ...*?

Option and the *arg*s determine the exact behavior of the command. The following commands are possible for scrolledcanvas widgets:

ASSOCIATED METHODS

addtag	**bbox**	**bind**	**canvasx**
canvasy	**coords**	**create**	**dchars**
delete	**dtag**	**find**	**focus**
gettags	**icursor**	**index**	**insert**
itemconfigure	**lower**	**move**	**postscript**
raise	**scale**	**scan**	**select**
type	**xview**	**yview**	

See the "canvas" manual entry for details on the associated methods.

WIDGET-SPECIFIC METHODS

pathName **cget** *option*

Returns the current value of the configuration option given by *option*. *Option* may have any of the values accepted by the **scrolledcanvas** command.

pathName **childsite**

Returns the child site widget path name.

pathName **configure** ?*option*? ?*value option value ...*?

Query or modify the configuration options of the widget. If no *option* is specified, returns a list describing all of the available options for *pathName* (see **Tk_ConfigureInfo** for information on the format of this list). If *option* is specified with no *value*, then the command returns a list describing the one named option (this list will be identical to the corresponding sublist of the value returned if no *option* is specified). If one or more *option−value* pairs are specified, then the command modifies the given widget option(s) to have the given value(s); in this case the command returns an empty string. *Option* may have any of the values accepted by the **scrolledcanvas** command.

pathName **justify** *direction*

Justifies the canvas contents via the scroll bars in one of four directions: **left**, **right**, **top**, or **bottom**.

COMPONENTS

Name: **canvas**
Class: **Canvas**

The canvas component is the canvas widget. See the "canvas" widget manual entry for details on the canvas component item.

Name: **horizsb**
Class: **Scrollbar**

The horizsb component is the horizontal scroll bar. See the "ScrollBar" widget manual entry for details on the horizsb component item.

Name: **vertsb**
Class: **Scrollbar**

The vertsb component is the vertical scroll bar. See the "ScrollBar" widget manual entry for details on the vertsb component item.

EXAMPLE

```
scrolledcanvas .sc

.sc create rectangle 100 100 400 400 -fill red
.sc create rectangle 300 300 600 600 -fill green
```

```
.sc create rectangle 200 200 500 500 -fill blue

pack .sc -padx 10 -pady 10 -fill both -expand yes
```

AUTHOR

Mark L. Ulferts

KEYWORDS

scrolledcanvas, canvas, widget

NAME

scrolledframe − Create and manipulate scrolled frame widgets

SYNOPSIS

scrolledframe *pathName* ?*options*?

INHERITANCE

itk::Widget <- Labeledwidget <-Scrolledwidget <- Scrolledframe

STANDARD OPTIONS

activeBackground	**background**	**borderWidth**	**cursor**
font	**foreground**	**highlightColor**	**highlightThickness**
relief	**selectBackground**	**selectBorderWidth**	**selectForeground**

See the "options" manual entry for details on the standard options.

ASSOCIATED OPTIONS

activeRelief	**elementBorderWidth**	**jump**	**troughColor**

See the "scrollbar" manual entry for details on the associated options.

INHERITED OPTIONS

LabelBitmap	**labelFont**	**labelImage**	**labelMargin**
labelPos	**labelText**	**labelVariable**	

See the "labeledwidget" class manual entry for details on the inherited options.

WIDGET-SPECIFIC OPTIONS

Name: **height**
Class: **Height**
Command-Line Switch: **-height**

Specifies the height of the scrolled frame widget in any of the forms acceptable to **Tk_GetPixels**. The default height is 100 pixels.

Name: **hscrollMode**
Class: **ScrollMode**
Command-Line Switch: **-hscrollmode**

Specifies the the display mode to be used for the horizontal scrollbar: **static**, **dynamic**, or **none**. In static mode, the scroll bar is displayed at all times. Dynamic mode displays the scroll bar as required, and none disables the scroll bar display. The default is static.

Name: **sbWidth**
Class: **Width**
Command-Line Switch: **-sbwidth**

Specifies the width of the scrollbar in any of the forms acceptable to **Tk_GetPixels**. The default width is 15 pixels.

Name: **scrollMargin**
Class: **Margin**
Command-Line Switch: **-scrollmargin**

Specifies the distance between the frame and scrollbar in any of the forms acceptable to **Tk_GetPixels**. The default is 3 pixels.

Name: **vscrollMode**
Class: **ScrollMode**
Command-Line Switch: **-vscrollmode**

Specifies the the display mode to be used for the vertical scrollbar: **static**, **dynamic**, or **none**. In

static mode, the scroll bar is displayed at all times. Dynamic mode displays the scroll bar as required, and none disables the scroll bar display. The default is static.

Name: **width**
Class: **Width**
Command-Line Switch: **-width**

Specifies the width of the scrolled frame widget in any of the forms acceptable to **Tk_GetPixels**. The default height is 100 pixels.

DESCRIPTION

The **scrolledframe** combines the functionallity of scrolling with that of a typical frame widget to implement a clipable viewing area whose visible region may be modified with the scroll bars. This enables the contruction of visually larger areas than which could normally be displayed, containing a heterogenous mix of other widgets. Options exist which allow full control over which scrollbars are displayed and the method, i.e. statically or dynamically. The frame itself may be accessed by the **childsite** method and then filled with other widget combinations.

METHODS

The **scrolledframe** command creates a new Tcl command whose name is *pathName*. This command may be used to invoke various operations on the widget. It has the following general form:

pathName option ?*arg arg ...*?

Option and the *arg*s determine the exact behavior of the command. The following commands are possible for scrolledframe widgets:

ASSOCIATED METHODS

xview **yview**

See the "canvas" manual entry for details on the associated methods.

WIDGET-SPECIFIC METHODS

pathName **cget** *option*

Returns the current value of the configuration option given by *option*. *Option* may have any of the values accepted by the **scrolledframe** command.

pathName **childsite**

Return the path name of the child site.

pathName **configure** ?*option*? ?*value option value ...*?

Query or modify the configuration options of the widget. If no *option* is specified, returns a list describing all of the available options for *pathName* (see **Tk_ConfigureInfo** for information on the format of this list). If *option* is specified with no *value*, then the command returns a list describing the one named option (this list will be identical to the corresponding sublist of the value returned if no *option* is specified). If one or more *option−value* pairs are specified, then the command modifies the given widget option(s) to have the given value(s); in this case the command returns an empty string. *Option* may have any of the values accepted by the **scrolledframe** command.

pathName **justify** *direction*

Justifies the frame contents via the scroll bars in one of four directions: **left**, **right**, **top**, or **bottom**.

COMPONENTS

Name: **horizsb**
Class: **Scrollbar**

The horizsb component is the horizontal scroll bar. See the "ScrollBar" widget manual entry for details on the horizsb component item.

Name: **vertsb**
Class: **Scrollbar**

The vertsb component is the vertical scroll bar. See the "ScrollBar" widget manual entry for details on the vertsb component item.

EXAMPLE

```
scrolledframe .sf -width 150 -height 180 -labelon yes -labeltext scrolledframe

set cs [.sf childsite]
pack [button $cs.b1 -text Hello] -pady 10
pack [button $cs.b2 -text World] -pady 10
pack [button $cs.b3 -text "This is a test"] -pady 10
pack [button $cs.b4 -text "This is a really big button"] -pady 10
pack [button $cs.b5 -text "This is another really big button"] -pady 10
pack [button $cs.b6 -text "This is the last really big button"] -pady 10

pack .sf -expand yes -fill both -padx 10 -pady 10
```

AUTHOR

Mark L. Ulferts

Sue Yockey

KEYWORDS

scrolledframe, frame, widget

NAME

scrolledhtml − Create and manipulate a scrolled text widget with the capability of displaying HTML formatted documents.

SYNOPSIS

scrolledhtml *pathName* ?*options*?

INHERITANCE

itk::Widget <- Labeledwidget <- Scrolledtext <- Scrolledhtml

STANDARD OPTIONS

activeBackground	**background**	**borderWidth**	**cursor**
exportSelection	**foreground**	**highlightColor**	**highlightThickness**
insertBackground	**insertBorderWidth**	**insertOffTime**	**insertOnTime**
insertWidth	**padX**	**padY**	**relief**
repeatDelay	**repeatInterval**	**selectBackground**	**selectBorderWidth**
selectForeground	**setGrid**		

See the "options" manual entry for details on the standard options.

ASSOCIATED OPTIONS

activeRelief	**elementBorderWidth**	**jump**	**troughColor**

See the "scrollbar" widget manual entry for details on the above associated options.

spacing1	**spacing2**	**spacing3**	**state**
wrap			

See the "text" widget manual entry for details on the above associated options.

INHERITED OPTIONS

labelBitmap	**labelFont**	**labelImage**	**labelMargin**
labelPos	**labelText**	**labelVariable**	**height**
hscrollMode	**sbWidth**	**scrollMargin**	**visibleitems**
vscrollMode	**width**		

See the "scrolledtext" class manual entry for details on the inherited options.

WIDGET-SPECIFIC OPTIONS

Name: **feedback**
Class: **FeedBack**
Command-Line Switch: **-feedback**

Specifies the callback command to use to give feedback on current status. The command is executed in the form *command <number of characters remaining>*

Name: **fixedfont**
Class: **FixedFont**
Command-Line Switch: **-fixedfont**

Specifies the name of the font to be used for fixed-width character text (such as <pre>...</pre> or <tt>...</tt>.) The size, style, and other font attributes are determined by the format tags in the document. The default is courier.

Name: **fontname**
Class: **FontName**
Command-Line Switch: **-fontname**

Specifies the name of the font to be used for normal-width character spaced text. The size, style, and other font attributes are determined by the format tags in the document. The default is times.

Name: **fontsize**
Class: **FontSize**
Command-Line Switch: **-fontsize**

Specifies the general size of the fonts used. One of small, medium, large, or huge. The default is medium.

Name: **foreground**
Class: **Foreground**
Command-Line Switch: **-foreground**

Specifies the color of text other than hypertext links, in any of the forms acceptable to **Tk_GetColor**. This value may be overridden in a particular document by the *text* attribute of the **Body** HTML tag.

Name: **link**
Class: **Link**
Command-Line Switch: **-link**

Specifies the default color of hypertext links in any of the forms acceptable to **Tk_GetColor**. This value may be overridden in a particular document by the *link* attribute of the **Body** HTML tag. The default is blue.

Name: **linkcommand**
Class: **LinkCommand**
Command-Line Switch: **-linkcommand**

Specifies the command to execute when the user clicks on a hypertext link. Execution is of the form **linkcommand href**, where **href** is the value given in the *href* attribute of the **A** HTML tag.

Name: **alink**
Class: **alink**
Command-Line Switch: **-alink**

Specifies the color of hypertext links when the cursor is over the link in any of the forms acceptable to **Tk_GetColor**. The default is red.

Name: **textBackground**
Class: **Background**
Command-Line Switch: **-textbackground**

Specifies the background color for the text area in any of the forms acceptable to **Tk_GetColor**. This value may be overridden in a particular document by the *bgcolor* attribute of the **Body** HTML tag.

Name: **unknownimage**
Class: **UnknownImage**
Command-Line Switch: **-unknownimage**

Specifies the name of the image file to display when an **img** specified in the html document cannot be loaded.

Name: **update**
Class: **Update**
Command-Line Switch: **-alink**

A boolean value indicating whether to call update during html rendering.

DESCRIPTION

The **scrolledhtml** command creates a scrolled text widget with the additional capability to display html formatted documents. An import method is provided to read an html document file, and a render method is provided to display a html formatted text string.

METHODS

The **scrolledhtml** command creates a new Tcl command whose name is *pathName*. This command may be used to invoke various operations on the widget. It has the following general form:

pathName option ?*arg arg* ...?

Option and the *arg*s determine the exact behavior of the command. The following commands are possible for scrolledhtml widgets:

ASSOCIATED METHODS

bbox	**compare**	**debug**	**delete**
dlineinfo	**get**	**index**	**insert**
mark	**scan**	**search**	**see**
tag	**window**	**xview**	**yview**

See the "text" manual entry for details on the standard methods.

INHERITED METHODS

export	**clear**

See the "scrolledhtml" manual entry for details on the inherited methods.

WIDGET-SPECIFIC METHODS

pathName **cget** *option*

Returns the current value of the configuration option given by *option*. *Option* may have any of the values accepted by the **scrolledhtml** command.

pathName **configure** ?*option*? ?*value option value* ...?

Query or modify the configuration options of the widget. If no *option* is specified, returns a list describing all of the available options for *pathName* (see **Tk_ConfigureInfo** for information on the format of this list). If *option* is specified with no *value*, then the command returns a list describing the one named option (this list will be identical to the corresponding sublist of the value returned if no *option* is specified). If one or more *option−value* pairs are specified, then the command modifies the given widget option(s) to have the given value(s); in this case the command returns an empty string. *Option* may have any of the values accepted by the **scrolledhtml** command.

pathName **import** ?*option*? *href*

Load html formatted text from a file. *Href* must exist. If *option* is -link, *href* is assumed to be relative to the application's current working directory. Otherwise, *href* is assumed to be relative to the path of the last page loaded. *Href* is either a filename, or a reference of the form *filename#anchorname*. In the latter form, fIFilename and/or *anchorname* may be empty. If *filename* is empty, the current document is assumed. If *anchorname* is empty, the top of the document is assumed.

pathName **pwd**

Print the current working directory of the widget, i.e. the directory of the last page loaded.

pathName **render** *htmltext* ?*wd*?

Display HTML formatted text *htmltext*. *Wd* gives the base path to use for all links and images in the document. *Wd* defaults to the application's current working directory.

pathName **title**
Return the title of the current page, as given in the <title>...</title> field in the document.

HTML COMPLIANCE

This widget is compliant with HTML 3.2 with the following exceptions:

No features requiring a connection to an http server are supported.

Some image alignments aren't supported, because they are not supported by the text widget.

The
 attributes dealing with image alignments aren't supported.

Automatic table sizing is not supported very well, due to limitations of the text widget

EXAMPLE

```
option add *textBackground white

scrolledhtml .sh -fontname helvetica -linkcommand "this import -link"

pack .sh -padx 10 -pady 10 -fill both -expand yes

.sh import ~/public_html/index.html
```

BUGS

Cells in a table can be caused to overlap. ex:
<table border width="100%">
<tr><td>cell1</td><td align=right rowspan=2>cell2</td></tr>
<tr><td colspan=2>cell3 w/ overlap</td>
</table> It hasn't been fixed because 1) it's a pain to fix, 2) it will slow tables down by a significant amount, and 3) netscape has the same bug, as of V3.01.

ACKNOWLEDGEMENTS

Sam Shen

This code is based largely on his tkhtml.tcl code from tk inspect. Tkhtml is copyright 1995 Lawrence Berkeley Laboratory.

AUTHOR

Kris Raney

KEYWORDS

scrolledhtml, html, text, widget

NAME

scrolledlistbox − Create and manipulate scrolled listbox widgets

SYNOPSIS

scrolledlistbox *pathName* ?*options*?

INHERITANCE

itk::Widget <- Labeledwidget <- Scrolledwidget <- Scrolledlistbox

STANDARD OPTIONS

activeBackground	**background**	**borderWidth**	**cursor**
exportSelection	**foreground**	**highlightColor**	**highlightThickness**
relief	**selectBackground**	**selectBorderWidth**	**selectForeground**

See the "options" manual entry for details on the standard options.

ASSOCIATED OPTIONS

selectMode

See the "listbox" widget manual entry for details on the above associated options.

activeRelief	**elementBorderwidth**	**jump**	**troughColor**

See the "scrollbar" widget manual entry for details on the above associated options.

INHERITED OPTIONS

disabledForeground	**labelBitmap**	**labelFont**	**labelImage**
labelMargin	**labelPos**	**labelText**	**labelVariable**
state			

See the "labeledwidget" class manual entry for details on the inherited options.

WIDGET-SPECIFIC OPTIONS

Name: **dblClickCommand**
Class: **Command**
Command-Line Switch: **-dblclickcommand**

Specifies a Tcl command procedure which is called when an item is double clicked. Typically this occurs when mouse button 1 is double clicked over an item. Selection policy does not matter.

Name: **height**
Class: **Height**
Command-Line Switch: **-height**

Specifies the height of the scrolled list box as an entire unit. The value may be specified in any of the forms acceptable to **Tk_GetPixels**. Any additional space needed to display the other components such as labels, margins, and scrollbars force the listbox to be compressed. A value of zero along with the same value for the width causes the value given for the visibleitems option to be applied which administers geometry constraints in a different manner. The default height is zero.

Name: **hscrollMode**
Class: **ScrollMode**
Command-Line Switch: **-hscrollmode**

Specifies the the display mode to be used for the horizontal scrollbar: **static, dynamic,** or **none**. In static mode, the scroll bar is displayed at all times. Dynamic mode displays the scroll bar as required, and none disables the scroll bar display. The default is static.

Name: **sbWidth**
Class: **Width**
Command-Line Switch: **-sbwidth**

Specifies the width of the scrollbar in any of the forms acceptable to **Tk_GetPixels**. The default width is 15 pixels..

Name: **scrollMargin**
Class: **Margin**
Command-Line Switch: **-scrollmargin**

Specifies the distance between the listbox and scrollbar in any of the forms acceptable to **Tk_GetPixels**. The default is 3 pixels.

Name: **selectionCommand**
Class: **Command**
Command-Line Switch: **-selectioncommand**

Specifies a Tcl command procedure which is called when an item is selected. Selection policy does not matter.

Name: **state**
Class: **State**
Command-Line Switch: **-state**

Specifies one of two states for the listbox: **normal** or **disabled**. If the listbox is disabled then selection is ignored. The default is normal.

Name: **textBackground**
Class: **Background**
Command-Line Switch **-textbackground**

Specifies the background color for the listbox. This allows the background within the listbox to be different from the normal background color.

Name: **textFont**
Class: **Font**
Command-Line Switch: **-textfont**

Specifies the font to be used for text in the listbox. This allows for the font associated with text internal to the scrolled listbox to be different than the font for labels.

Name: **visibleitems**
Class: **VisibleItems**
Command-Line Switch: **-visibleitems**

Specifies the widthxheight in characters and lines for the listbox. This option is only administered if the width and height options are both set to zero, otherwise they take precedence. The default value is 20x10. With the visibleitems option engaged, geometry constraints are maintained only on the listbox. The size of the other components such as labels, margins, and scroll bars, are additive and independent, effecting the overall size of the scrolled list box. In contrast, should the width and height options have non zero values, they are applied to the scrolled list box as a whole. The listbox is compressed or expanded to maintain the geometry constraints.

Name: **vscrollMode**
Class: **ScrollMode**
Command-Line Switch: **-vscrollmode**

Specifies the the display mode to be used for the vertical scrollbar: **static, dynamic,** or **none**. In static mode, the scroll bar is displayed at all times. Dynamic mode displays the scroll bar as required, and none disables the scroll bar display. The default is static.

Name: **width**
Class: **Width**
Command-Line Switch: **-width**

Specifies the width of the scrolled list box as an entire unit. The value may be specified in any of the forms acceptable to **Tk_GetPixels**. Any additional space needed to display the other components such as labels, margins, and scrollbars force the listbox to be compressed. A value of zero along with the same value for the height causes the value given for the visibleitems option to be applied which administers geometry constraints in a different manner. The default width is zero.

DESCRIPTION

The **scrolledlistbox** command creates a scrolled listbox with additional options to manage horizontal and vertical scrollbars. This includes options to control which scrollbars are displayed and the method, i.e. statically or dynamically.

METHODS

The **scrolledlistbox** command creates a new Tcl command whose name is *pathName*. This command may be used to invoke various operations on the widget. It has the following general form:

pathName option ?*arg arg ...*?

Option and the *arg*s determine the exact behavior of the command.

Many of the widget commands for a scrolledlistbox take as one argument an indicator of which entry of the list box to operate on. These indicators are called *index*es and may be specified in any of the following forms:

number Specifies the element as a numerical index, where 0 corresponds to the first element in the listbox.

active Indicates the element that has the location cursor. This element will be displayed with an underline when the listbox has the keyboard focus, and it is specified with the **activate** widget command.

anchor Indicates the anchor point for the selection, which is set with the **selection anchor** widget command.

end Indicates the end of the listbox. For some commands this means just after the last element; for other commands it means the last element.

@x,y Indicates the element that covers the point in the listbox window specified by *x* and *y* (in pixel coordinates). If no element covers that point, then the closest element to that point is used.

pattern If the index doesn't satisfy one of the above forms then this form is used. *Pattern* is pattern-matched against the items in the list box, in order from the top down, until a matching entry is found. The rules of **Tcl_StringMatch** are used.

The following widget commands are possible for scrolledlistbox widgets:

ASSOCIATED METHODS

activate	**bbox**	**curselection**	**delete**
get	**index**	**insert**	**nearest**
scan	**see**	**selection**	**size**
xview	**yview**		

See the "listbox" manual entry for details on the associated methods.

WIDGET-SPECIFIC METHODS

pathName **cget** *option*

Returns the current value of the configuration option given by *option*. *Option* may have any of the values accepted by the **scrolledlistbox** command.

pathName **clear**

Clears the listbox of all items.

pathName **configure** ?*option*? ?*value option value ...*?

Query or modify the configuration options of the widget. If no *option* is specified, returns a list describing all of the available options for *pathName* (see **Tk_ConfigureInfo** for information on the format of this list). If *option* is specified with no *value*, then the command returns a list describing the one named option (this list will be identical to the corresponding sublist of the value returned if no *option* is specified). If one or more *option−value* pairs are specified, then the command modifies the given widget option(s) to have the given value(s); in this case the command returns an empty string. *Option* may have any of the values accepted by the **scrolledlistbox** command.

pathName **getcurselection**

Returns the contents of the listbox element indicated by the current selection indexes. Short cut version of get and curselection command combination.

pathName **justify** *direction*

Justifies the list contents via teh scroll bars in one of four directions: **left**, **right**, **top**, or **bottom**.

pathName **selecteditemcount**

Returns the number of items currently selected in the list.

pathName **sort** *order*

Sort the current list in either **ascending** or **descending** order. The values **increasing** and **decreasing** are also accepted.

COMPONENTS

Name: **listbox**
Class: **listbox**

The listbox component is the listbox widget. See the "listbox" widget manual entry for details on the listbox component item.

Name: **horizsb**
Class: **Scrollbar**

The horizsb component is the horizontal scroll bar. See the "scrollbar" widget manual entry for details on the horizsb component item.

Name: **vertsb**
Class: **Scrollbar**

The vertsb component is the vertical scroll bar. See the "scrollbar" widget manual entry for details on the vertsb component item.

EXAMPLE

```
option add *textBackground white
proc selCmd {} {
  puts stdout "[.slb getcurselection]"
}
proc defCmd {} {
```

```
  puts stdout "Double Click"
  return [selCmd]
}
scrolledlistbox .slb -selection single \
  -vscrollmode static -hscrollmode dynamic -labeltext "List" \
  -selectioncommand selCmd -dblclickcommand defCmd
pack .slb -padx 10 -pady 10 -fill both -expand yes
.slb insert end {Hello {Out There} World}
```

AUTHOR

Mark L. Ulferts

KEYWORDS

scrolledlistbox, listbox, widget

NAME

scrolledtext − Create and manipulate a scrolled text widget

SYNOPSIS

scrolledtext *pathName* ?*options*?

INHERITANCE

itk::Widget <- Labeledwidget <- Scrolledwidget <- Scrolledtext

STANDARD OPTIONS

activeBackground	**background**	**borderWidth**	**cursor**
exportSelection	**foreground**	**highlightColor**	**highlightThickness**
insertBackground	**insertBorderWidth**	**insertOffTime**	**insertOnTime**
insertWidth	**padX**	**padY**	**relief**
selectBackground	**selectBorderWidth**	**selectForeground**	**setGrid**

See the "options" manual entry for details on the standard options.

ASSOCIATED OPTIONS

activeRelief	**elementBorderWidth**	**jump**	**troughColor**

See the "scrollbar" widget manual entry for details on the above associated options.

spacing1	**spacing2**	**spacing3**	**state**
wrap			

See the "text" widget manual entry for details on the above associated options.

INHERITED OPTIONS

disabledForeground	**labelBitmap**	**labelFont**	**labelImage**
labelMargin	**labelPos**	**labelText**	**labelVariable**
state			

See the "labeledwidget" class manual entry for details on the inherited options.

WIDGET-SPECIFIC OPTIONS

Name: **height**
Class: **Height**
Command-Line Switch: **-height**

Specifies the height of the scrolled text as an entire unit. The value may be specified in any of the forms acceptable to **Tk_GetPixels**. Any additional space needed to display the other components such as labels, margins, and scrollbars force the text to be compressed. A value of zero along with the same value for the width causes the value given for the visibleitems option to be applied which administers geometry constraints in a different manner. The default height is zero.

Name: **hscrollMode**
Class: **ScrollMode**
Command-Line Switch: **-hscrollmode**

Specifies the the display mode to be used for the horizontal scrollbar: **static, dynamic,** or **none**. In static mode, the scroll bar is displayed at all times. Dynamic mode displays the scroll bar as required, and none disables the scroll bar display. The default is static.

Name: **sbWidth**
Class: **Width**
Command-Line Switch: **-sbwidth**

Specifies the width of the scrollbar in any of the forms acceptable to **Tk_GetPixels**.

Name: **scrollMargin**
Class: **Margin**
Command-Line Switch: **-scrollmargin**

Specifies the distance between the text area and scrollbar in any of the forms acceptable to **Tk_GetPixels**. The default is 3 pixels.

Name: **textBackground**
Class: **Background**
Command-Line Switch: **-textbackground**

Specifies the background color for the text area in any of the forms acceptable to **Tk_GetColor**.

Name: **textFont**
Class: **Font**
Command-Line Switch: **-textfont**

Specifies the font to be used in the scrolled text area.

Name: **visibleitems**
Class: **VisibleItems**
Command-Line Switch: **-visibleitems**

Specifies the widthxheight in characters and lines for the text. This option is only administered if the width and height options are both set to zero, otherwise they take precedence. The default value is 80x24. With the visibleitems option engaged, geometry constraints are maintained only on the text. The size of the other components such as labels, margins, and scroll bars, are additive and independent, effecting the overall size of the scrolled text. In contrast, should the width and height options have non zero values, they are applied to the scrolled text as a whole. The text is compressed or expanded to maintain the geometry constraints.

Name: **vscrollMode**
Class: **ScrollMode**
Command-Line Switch: **-vscrollmode**

Specifies the the display mode to be used for the vertical scrollbar: **static, dynamic,** or **none**. In static mode, the scroll bar is displayed at all times. Dynamic mode displays the scroll bar as required, and none disables the scroll bar display. The default is static.

Name: **width**
Class: **Width**
Command-Line Switch: **-width**

Specifies the width of the scrolled text as an entire unit. The value may be specified in any of the forms acceptable to **Tk_GetPixels**. Any additional space needed to display the other components such as labels, margins, and scrollbars force the text to be compressed. A value of zero along with the same value for the height causes the value given for the visibleitems option to be applied which administers geometry constraints in a different manner. The default width is zero.

DESCRIPTION

The **scrolledtext** command creates a scrolled text widget with additional options to manage the scrollbars. This includes options to control the method in which the scrollbars are displayed, i.e. statically or dynamically. Options also exist for adding a label to the scrolled text area and controlling its position. Import/export of methods are provided for file I/O.

METHODS

The **scrolledtext** command creates a new Tcl command whose name is *pathName*. This command may be used to invoke various operations on the widget. It has the following general form:

pathName option ?*arg arg ...*?

Option and the *arg*s determine the exact behavior of the command. The following commands are possible for scrolledtext widgets:

ASSOCIATED METHODS

bbox	**compare**	**debug**	**delete**
dlineinfo	**get**	**index**	**insert**
mark	**scan**	**search**	**see**
tag	**window**	**xview**	**yview**

See the "text" manual entry for details on the standard methods.

WIDGET-SPECIFIC METHODS

pathName **cget** *option*

Returns the current value of the configuration option given by *option*. *Option* may have any of the values accepted by the **scrolledtext** command.

pathName **childsite**

Returns the child site widget path name.

pathName **clear**

Clear the text area of all characters.

pathName **configure** ?*option*? ?*value option value ...*?

Query or modify the configuration options of the widget. If no *option* is specified, returns a list describing all of the available options for *pathName* (see **Tk_ConfigureInfo** for information on the format of this list). If *option* is specified with no *value*, then the command returns a list describing the one named option (this list will be identical to the corresponding sublist of the value returned if no *option* is specified). If one or more *option−value* pairs are specified, then the command modifies the given widget option(s) to have the given value(s); in this case the command returns an empty string. *Option* may have any of the values accepted by the **scrolledtext** command.

pathName **import** *filename* ?*index*?

Load the text from a file into the text area at the *index*. The *filename* must exist.

pathName **export** *filename*

Write text to a file. If *filename* exists then contents are replaced with text widget contents.

COMPONENTS

Name: **text**
Class: **Text**

The text component is the text widget. See the "text" widget manual entry for details on the text component item.

Name: **horizsb**
Class: **Scrollbar**

The horizsb component is the horizontal scroll bar. See the "scrollbar" widget manual entry for details on the horizsb component item.

Name: **vertsb**
Class: **Scrollbar**

The vertsb component is the vertical scroll bar. See the "scrollbar" widget manual entry for details on the vertsb component item.

EXAMPLE

```
option add *textBackground white

scrolledtext .st -scrollmode dynamic -labeltext "Password File"

pack .st -padx 10 -pady 10 -fill both -expand yes

.st import /etc/passwd
```

AUTHOR

Mark L. Ulferts

KEYWORDS

scrolledtext, text, widget

NAME

selectionbox − Create and manipulate a selection box widget

SYNOPSIS

selectionbox *pathName* ?*options*?

INHERITANCE

itk::Widget <- selectionbox

STANDARD OPTIONS

activeBackground	**background**	**borderWidth**	**cursor**
exportSelection	**foreground**	**highlightColor**	**highlightThickness**
insertBackground	**insertBorderWidth**	**insertOffTime**	**insertOnTime**
insertWidth	**relief**	**repeatDelay**	**repeatInterval**
selectBackground	**selectBorderWidth**	**selectForeground**	

See the "options" manual entry for details on the standard options.

ASSOCIATED OPTIONS

textBackground **textFont**

See the "entryfield" widget class manual entry for details on the above associated options.

labelFont **labelMargin**

See the "labeledwidget" class manual entry for details on the above associated options.

activeRelief **elementBorderWidth** **jump** **troughColor**

See the "scrollbar" widget class manual entry for details on the above associated options.

dblClickCommand	**hscrollMode**	**sbWidth**	**scrollMargin**
textBackground	**textFont**	**vscrollMode**	

See the "scrolledlistbox" widget class manual entry for details on the above associated options.

WIDGET-SPECIFIC OPTIONS

Name: **childSitePos**
Class: **Position**
Command-Line Switch: **-childsitepos**

Specifies the position of the child site in the selection box: **n**, **s**, **e**, **w**, or . The default is center

Name: **height**
Class: **Height**
Command-Line Switch: **-height**

Specifies the height of the selection box. The value may be specified in any of the forms acceptable to Tk_GetPixels. The default is 320 pixels.

Name: **itemsCommand**
Class: **Command**
Command-Line Switch: **-itemscommand**

Specifies a command to be evaluated following selection of an item.

Name: **itemsLabel**
Class: **Text**
Command-Line Switch: **-itemslabel**

Specifies the text of the label for the items list. The default is "List".

Name: **itemsLabelPos**
Class: **Position**
Command-Line Switch: **-itemslabelpos**

Specifies the position of the label along the side of the items list: **n**, **ne**, **e**, **se**, **s**, **sw**, **w**, or **nw**. The default is nw.

Name: **itemsOn**
Class: **ItemsOn**
Command-Line Switch: **-itemson**

Specifies whether or not to display the items list in any of the forms acceptable to **Tcl_GetBoolean**. The default is true.

Name: **margin**
Class: **Margin**
Command-Line Switch: **-margin**

Specifies distance between the items list and selection entry in any of the forms acceptable to **Tk_GetPixels**. The default is 7 pixels.

Name: **selectionCommand**
Class: **Command**
Command-Line Switch: **-selectioncommand**

Specifies a Tcl procedure to be associated with a return key press event in the selection entry field.

Name: **selectionLabel**
Class: **Text**
Command-Line Switch: **-selectionlabel**

Specifies the text of the label for the selection entry field. The default is "Selection".

Name: **selectionLabelPos**
Class: **Position**
Command-Line Switch: **-selectionlabelpos**

Specifies the position of the label along the side of the selection: **n**, **ne**, **e**, **se**, **s**, **sw**, **w**, or **nw**. The default is nw.

Name: **selectionOn**
Class: **SelectionOn**
Command-Line Switch: **-selectionon**

Specifies whether or not to display the selection entry in any of the forms acceptable to **Tcl_GetBoolean**. The default is true.

Name: **width**
Class: **Width**
Command-Line Switch: **-width**

Specifies the width of the selection box. The value may be specified in any of the forms acceptable to Tk_GetPixels. The default is 260 pixels.

DESCRIPTION

The **selectionbox** command creates a scrolled list of items and a selection entry field. The user may choose any of the items displayed in the scrolled list of alternatives and the selection field will be filled with the choice. The user is also free to enter a new value in the selection entry field. Both the list and entry areas have labels. A child site is also provided in which the user may create other widgets to be used in

conjunction with the selection box.

METHODS

The **selectionbox** command creates a new Tcl command whose name is *pathName*. This command may be used to invoke various operations on the widget. It has the following general form:

pathName option ?*arg arg ...*?

Option and the *arg*s determine the exact behavior of the command.

ASSOCIATED METHODS

curselection	**delete**	**index**	**nearest**
scan	**selection**	**size**	

See the "listbox" widget class manual entry for details on the associated methods.

WIDGET-SPECIFIC METHODS

pathName **cget** *option*

Returns the current value of the configuration option given by *option*. *Option* may have any of the values accepted by the **selectionbox** command.

pathName **childsite**

Returns the child site widget path name.

pathName **clear** *component*

Delete the contents of either the selection entry widget or items list. The *component* argument may be either **items** or **selection**.

pathName **configure** ?*option*? ?*value option value ...*?

Query or modify the configuration options of the widget. If no *option* is specified, returns a list describing all of the available options for *pathName* (see **Tk_ConfigureInfo** for information on the format of this list). If *option* is specified with no *value*, then the command returns a list describing the one named option (this list will be identical to the corresponding sublist of the value returned if no *option* is specified). If one or more *option−value* pairs are specified, then the command modifies the given widget option(s) to have the given value(s); in this case the command returns an empty string. *Option* may have any of the values accepted by the **selectionbox** command.

pathName **get**

Returns the current value of the selection entry widget.

pathName **insert** *component args*

Insert element(s) into either the selection entry widget or items list. The *component* argument may be either **items** or **selection**. The *args* follow the rules of either an entry or list widget depending on the *component* value.

pathName **selectitem**

Replace the selection entry field contents with the currently selected items value.

COMPONENTS

Name:	**childsite**
Class:	**Frame**

The childsite component is the user child site for the selection box. See the "frame" widget manual entry for details on the childsite component item.

Name: **items**
Class: **Scrolledlistbox**

The items component provides the scrolled list box of items for the selection box. See the "scrolledlistbox" widget manual entry for details on the items component item.

Name: **selection**
Class: **Entryfield**

The selection component provides the entry field in the selection box for display of the selected item in the items component. See the "entryfield" widget manual entry for details on the selection component item.

EXAMPLE

```
option add *textBackground white

selectionbox .sb -items {Hello {Out There} World}
pack .sb -padx 10 -pady 10 -fill both -expand yes

set cs [label [.sb childsite].label -text "Child Site"]
pack $cs -fill x -padx 10 -pady 10

.sb insert items 2 {Cruel Cruel}

.sb selection set 1
```

AUTHOR

Mark L. Ulferts

KEYWORDS

selectionbox, widget

NAME

selectiondialog − Create and manipulate a selection dialog widget

SYNOPSIS

selectiondialog *pathName* ?*options*?

INHERITANCE

itk::Toplevel <- Shell <- Dialogshell <- Dialog <- Selectiondialog

STANDARD OPTIONS

activeBackground	**background**	**borderWidth**	**cursor**
exportSelection	**foreground**	**highlightColor**	**highlightThickness**
insertBackground	**insertBorderWidth**	**insertOffTime**	**insertOnTime**
insertWidth	**selectBackground**	**selectBorderWidth**	**selectForeground**

See the "options" manual entry for details on the standard options.

ASSOCIATED OPTIONS

textBackground **textFont**

See the "entryfield" widget manual entry for details on the above associated options.

labelFont

See the "labeledwidget" widget manual entry for details on the above associated options.

activeRelief	**elementBorderWidth**	**jump**	**troughColor**

See the "scrollbar" widget class manual entry for details on the above associated options.

textBackground **textFont**

See the "scrolledlistbox" widget class manual entry for details on the above associated options.

childsitepos	**itemsCommand**	**itemsLabel**	**itemsOn**
selectionCommand	**selectionLabel**	**selectionOn**	

See the "selectionbox" widget manual entry for details on the above associated options.

INHERITED OPTIONS

buttonBoxPadX	**buttonBoxPadY**	**buttonBoxPos**	**padX**
padY	**separator**	**thickness**	

See the "dialogshell" widget manual entry for details on the above inherited options.

height	**master**	**modality**	**width**

See the "shell" widget manual entry for details on the above inherited options.

title

See the "Toplevel" widget manual entry for details on the above inherited options.

DESCRIPTION

The **selectiondialog** command creates a selection box similar to the OSF/Motif standard selection dialog composite widget. The selectiondialog is derived from the Dialog class and is composed of a selectionbox with commands to manipulate the dialog buttons.

METHODS

The **selectiondialog** command creates a new Tcl command whose name is *pathName*. This command may be used to invoke various operations on the widget. It has the following general form:

pathName option ?*arg arg ...*?

Option and the *arg*s determine the exact behavior of the command. The following commands are possible for selectiondialog widgets:

ASSOCIATED METHODS

childsite **clear** **get** **insert**
selectitem

See the "selectionbox" widget manual entry for details on the above associated methods.

curselection **delete** **index** **nearest**
scan **selection** **size**

See the "listbox" widget manual entry for details on the above associated methods.

INHERITED METHODS

add **buttonconfigure** **default** **hide**
invoke **show**

See the "buttonbox" widget manual entry for details on the above inherited methods.

activate **center** **deactivate**

See the "shell" widget manual entry for details on the above inherited methods.

WIDGET-SPECIFIC METHODS

pathName **cget** *option*

Returns the current value of the configuration option given by *option*. *Option* may have any of the values accepted by the **selectiondialog** command.

pathName **configure** ?*option*? ?*value option value ...*?

Query or modify the configuration options of the widget. If no *option* is specified, returns a list describing all of the available options for *pathName* (see **Tk_ConfigureInfo** for information on the format of this list). If *option* is specified with no *value*, then the command returns a list describing the one named option (this list will be identical to the corresponding sublist of the value returned if no *option* is specified). If one or more *option−value* pairs are specified, then the command modifies the given widget option(s) to have the given value(s); in this case the command returns an empty string. *Option* may have any of the values accepted by the **selectiondialog** command.

COMPONENTS

Name: **selectionbox**
Class: **Selectionbox**

The selectionbox component is the selection box for the selection dialog. See the "selectionbox" widget manual entry for details on the selectionbox component item.

EXAMPLE

```
selectiondialog .sd
.sd activate
```

AUTHOR

Mark L. Ulferts

KEYWORDS

selectiondialog, selectionbox, dialog, dialogshell, shell, widget

NAME

shell − Create and manipulate a shell widget

SYNOPSIS

shell *pathName* ?*options*?

INHERITANCE

itk::Toplevel <- shell

STANDARD OPTIONS

background **cursor** **foreground**

See the "options" manual entry for details on the standard options.

INHERITED OPTIONS

title

See the "Toplevel" manual entry for details on the above inherited options.

WIDGET-SPECIFIC OPTIONS

Name: **height**
Class: **Height**
Command-Line Switch: **-height**

Specifies the height of the shell. The value may be specified in any of the forms acceptable to **Tk_GetPixels**. A value of zero causes the height to be adjusted to the required value based on the size requests of the components placed in the childsite. Otherwise, the height is fixed. The default is zero. NOTE: This may cause some amount of flickering on slower machines. To prevent it simply set the width and height to a appropriate value.

Name: **master**
Class: **Window**
Command-Line Switch: **-master**

Defines the shell as being a transient window with the master window given by the master option. The master window should be either another existing toplevel window or {} for no master. The default is {} for shells and "." for dialogs.

Name: **modality**
Class: **Modality**
Command-Line Switch: **-modality**

Allows the shell to grab control of the screen in one of three different ways: **application**, **system**, or **none**. Application modal prevents any other toplevel windows within the application which are direct children of '.' from gaining focus. System modal locks the screen and prevents all windows from gaining focus regardless of application. A modality of none performs no grabs at all. The default is none.

Name: **padX**
Class: **Pad**
Command-Line Switch: **-padx**

Specifies a padding distance for the childsite in the X-direction in any of the forms acceptable to **Tk_GetPixels**. The default is 10.

Name: **padY**
Class: **Pad**
Command-Line Switch: **-pady**

Specifies a padding distance for the childsite in the Y-direction in any of the forms acceptable to

Tk_GetPixels. The default is 10.

Name: **width**
Class: **Width**
Command-Line Switch: **-width**

Specifies the width of the shell. The value may be specified in any of the forms acceptable to **Tk_GetPixels**. A value of zero causes the width to be adjusted to the required value based on the size requests of the components placed in the childsite. Otherwise, the width is fixed. The default is zero. NOTE: This may cause some amount of flickering on slower machines. To prevent it simply set the width and height to a appropriate value.

DESCRIPTION

The **shell** command creates a shell which is a top level widget which supports modal operation.

METHODS

The **shell** command create a new Tcl command whose name is *pathName*. This command may be used to invoke various operations on the widget. It has the following general form:

pathName option ?*arg arg ...*?

Option and the *arg*s determine the exact behavior of the command. The following commands are possible for shell widgets:

WIDGET-SPECIFIC METHODS

pathName **activate**
Display the shell and wait based on the modality. For application and system modal activations, perform a grab operation, and wait for the result. The result may be returned via an argument to the **deactivate** method.

pathName **center** *?widget?*
Centers the shell with respect to another widget. The widget argument is optional. If provided, it should be the path of another widget with to center upon. If absent, then the shell will be centered on the screen as a whole.

pathName **cget** *option*
Returns the current value of the configuration option given by *option*. *Option* may have any of the values accepted by the **shell** command.

pathName **childsite**
Returns the pathname of the child site widget.

pathName **configure** ?*option*? ?*value option value ...*?
Query or modify the configuration options of the widget. If no *option* is specified, returns a list describing all of the available options for *pathName* (see **Tk_ConfigureInfo** for information on the format of this list). If *option* is specified with no *value*, then the command returns a list describing the one named option (this list will be identical to the corresponding sublist of the value returned if no *option* is specified). If one or more *option−value* pairs are specified, then the command modifies the given widget option(s) to have the given value(s); in this case the command returns an empty string. *Option* may have any of the values accepted by the **shell** command.

pathName **deactivate** ?*arg*?
Deactivate the display of the shell. The method takes an optional argument to be passed to the **activate** method which returns the value. The optional argument is only effective for application and system modal dialogs.

COMPONENTS

Name: **shellchildsite**
Class: **frame**

The shellchildsite component is the user child site for the shell. See the "frame" widget manual entry for details on the shellchildsite component item.

EXAMPLE

```
shell .sh -modality application -padx 20 -pady 20 -title Shell

pack [label [.sh childsite].l -text SHELL]

.sh center
.sh activate
```

AUTHOR

Mark L. Ulferts

Kris Raney

KEYWORDS

shell, widget

NAME

spindate − Create and manipulate time spinner widgets

SYNOPSIS

spindate *pathName* ?*options*?

INHERITANCE

itk::Widget <- Spindate

STANDARD OPTIONS

background **cursor** **foreground** **relief**

See the "options" manual entry for details on the standard options.

ASSOCIATED OPTIONS

textBackground **textFont**

See the "entryfield" manual entry for details on the above associated options.

labelFont **labelMargin**

See the "labeledwidget" manual entry for details on the above associated options.

arrowOrient **repeatDelay** **repeatInterval**

See the "spinner" manual entry for details on the above associated options.

WIDGET-SPECIFIC OPTIONS

Name: **dateMargin**
Class: **Margin**
Command-Line Switch: **-datemargin**

Specifies the margin space between the month, day, and year spinners is any of the forms accpetable to **Tcl_GetPixels**. The default is 1 pixel.

Name: **dayLabel**
Class: **Text**
Command-Line Switch: **-daylabel**

Specifies the text of the label for the day spinner. The default is "Day".

Name: **dayOn**
Class: **dayOn**
Command-Line Switch: **-dayon**

Specifies whether or not to display the day spinner in any of the forms acceptable to **Tcl_GetBoolean**. The default is true.

Name: **dayWidth**
Class: **Width**
Command-Line Switch: **-daywidth**

Specifies the width of the day spinner in any of the forms acceptable to **Tcl_GetPixels**. The default is 3 pixels.

Name: **labelPos**
Class: **Position**
Command-Line Switch: **-labelpos**

Specifies the position of the label along the sides of the various spinners: **n**, **e**, **s**, or **w**. The default is w.

Name: **monthFormat**
Class: **MonthFormat**
Command-Line Switch: **-monthformat**

Specifies the format of month display, **integer** (1-12) or **brief** strings (Jan - Dec), or **full** strings (January - December).

Name: **monthLabel**
Class: **Text**
Command-Line Switch: **-monthlabel**

Specifies the text of the label for the month spinner. The default is "Month".

Name: **monthOn**
Class: **monthOn**
Command-Line Switch: **-monthon**

Specifies whether or not to display the month spinner in any of the forms acceptable to **Tcl_GetBoolean**. The default is true.

Name: **monthWidth**
Class: **Width**
Command-Line Switch: **-monthwidth**

Specifies the width of the month spinner in any of the forms acceptable to **Tcl_GetPixels**. The default is 3 pixels.

Name: **orient**
Class: **Orient**
Command-Line Switch: **-orient**

Specifies the orientation of the month, day, and year spinners: **vertical** or **horizontal**. The default is horizontal.

Name: **yearDigits**
Class: **YearDigits**
Command-Line Switch: **-yeardigits**

Specifies the number of digits to be displayed as the value for the year spinner. The valid values are 2 and 4. The default is 2.

Name: **yearLabel**
Class: **Text**
Command-Line Switch: **-yearlabel**

Specifies the text of the label for the year spinner. The default is "Year"

Name: **yearOn**
Class: **yearOn**
Command-Line Switch: **-yearon**

Specifies whether or not to display the year spinner in any of the forms acceptable to **Tcl_GetBoolean**. The default is true.

Name: **yearWidth**
Class: **Width**
Command-Line Switch: **-yearwidth**

Specifies the width of the year spinner in any of the forms acceptable to **Tcl_GetPixels**. The default is 3 pixels.

DESCRIPTION

The **spindate** command creates a set of spinners for use in date value entry. The set includes an month, day, and year spinner widget.

METHODS

The **spindate** command creates a new Tcl command whose name is *pathName*. This command may be used to invoke various operations on the widget. It has the following general form:

pathName option ?*arg arg ...*?

Option and the *arg*s determine the exact behavior of the command. The following commands are possible for spindate widgets:

WIDGET-SPECIFIC METHODS

pathName **cget** *option*

Returns the current value of the configuration option given by *option*. *Option* may have any of the values accepted by the **spindate** command.

pathName **configure** ?*option*? ?*value option value ...*?

Query or modify the configuration options of the widget. If no *option* is specified, returns a list describing all of the available options for *pathName* (see **Tk_ConfigureInfo** for information on the format of this list). If *option* is specified with no *value*, then the command returns a list describing the one named option (this list will be identical to the corresponding sublist of the value returned if no *option* is specified). If one or more *option−value* pairs are specified, then the command modifies the given widget option(s) to have the given value(s); in this case the command returns an empty string. *Option* may have any of the values accepted by the **spindate** command.

pathName **get** ?**format**?

Returns the current contents of the spindate widget in a format of string or as an integer clock value using the **-string** and **-clicks** format options respectively. The default is by string. Reference the clock command for more information on obtaining dates and their formats.

pathName **show** *date*

Changes the currently displayed date to be that of the date argument. The date may be specified either as a string, an integer clock value or the keyword "now". Reference the clock command for more information on obtaining dates and their formats.

COMPONENTS

Name: **month**
Class: **Spinner**

The month spinner component is the month spinner of the date spinner. See the Spinner widget manual entry for details on the month component item.

Name: **day**
Class: **Spinint**

The day spinner component is the day spinner of the date spinner. See the SpinInt widget manual entry for details on the day component item.

Name: **year**
Class: **Spinint**

The year spinner component is the year spinner of the date spinner. See the SpinInt widget manual entry fordetails on the year component item.

EXAMPLE

```
spindate .sd
pack .sd -padx 10 -pady 10
```

AUTHOR

Sue Yockey

Mark L. Ulferts

KEYWORDS

spindate, spinint, spinner, entryfield, entry, widget

NAME

spinint − Create and manipulate a integer spinner widget

SYNOPSIS

spinint *pathName* ?*options*?

INHERITANCE

itk::Widget <- Labeledwidget <- Spinner <- Spinint

STANDARD OPTIONS

background	**borderWidth**	**cursor**	**exportSelection**
foreground	**highlightColor**	**highlightThickness**	**insertBackground**
insertBorderWidth	**insertOffTime**	**insertOnTime**	**insertWidth**
justify	**relief**	**selectBackground**	**selectBorderWidth**
selectForeground	**textVariable**		

See the "options" manual entry for details on the standard options.

ASSOCIATED OPTIONS

show	**state**

See the "entry" manual entry for details on the associated options.

INHERITED OPTIONS

command	**childSitePos**	**fixed**	**focusCommand**
invalid	**textBackground**	**textFont**	**validate**
width			

See the "entryfield" widget manual entry for details on the above inherited options.

disabledForeground	**labelBitmap**	**labelFont**	**labelImage**
labelMargin	**labelPos**	**labelText**	**labelVariable**
state			

See the "labeledwidget" widget manual entry for details on the above inherited options.

arroworient	**decrement**	**increment**	**repeatDelay**
repeatInterval			

See the "spinner" widget manual entry for details on the above inherited options.

WIDGET-SPECIFIC OPTIONS

Name: **range**
Class: **Range**
Command-Line Switch: **-range**

Specifies a two element list of minimum and maximum integer values. The default is no range, {{} {}}.

Name: **step**
Class: **Step**
Command-Line Switch: **-step**

Specifies the increment/decrement value. The default is 1.

Name: **wrap**
Class: **Wrap**
Command-Line Switch: **-wrap**

Specifies whether to wrap the spinner value upon reaching the minimum or maximum value in any of the forms acceptable to **Tcl_GetBoolean**. The default is true.

DESCRIPTION

The **spinint** command creates a spinint widget. The spinint allows "spinning" of integer values within a specified range with wrap support. The spinner arrows may be drawn horizontally or vertically.

METHODS

The **spinint** command creates a new Tcl command whose name is *pathName*. This command may be used to invoke various operations on the widget. It has the following general form:

pathName option ?*arg arg* ...?

Option and the *arg*s determine the exact behavior of the command. The following commands are possible for spinint widgets:

ASSOCIATED METHODS

delete	**get**	**icursor**	**index**
insert	**peek**	**scan**	**selection**
xview			

See the "entry" manual entry for details on the associated methods.

INHERITED METHODS

childsite	**clear**

See the "entryfield" manual entry for details on the associated methods.

WIDGET-SPECIFIC METHODS

pathName **cget** *option*

Returns the current value of the configuration option given by *option*. *Option* may have any of the values accepted by the **spinint** command.

pathName **configure** ?*option*? ?*value option value* ...?

Query or modify the configuration options of the widget. If no *option* is specified, returns a list describing all of the available options for *pathName* (see **Tk_ConfigureInfo** for information on the format of this list). If *option* is specified with no *value*, then the command returns a list describing the one named option (this list will be identical to the corresponding sublist of the value returned if no *option* is specified). If one or more *option−value* pairs are specified, then the command modifies the given widget option(s) to have the given value(s); in this case the command returns an empty string. *Option* may have any of the values accepted by the **spinint** command.

pathName **down**

Decrement the spinner value by the value given in the step option.

pathName **up**

Increment the spinner value by the value given in the step option.

COMPONENTS

See the "Spinner" widget manual entry for details on the integer spinner component items.

EXAMPLE

```
option add *textBackground white

spinint .si -labeltext "Temperature" -labelpos w \
```

```
-fixed yes -width 5 -range {32 212}

pack .si -pady 10
```

AUTHOR

Sue Yockey

KEYWORDS

spinint, widget

NAME

spinner − Create and manipulate a spinner widget

SYNOPSIS

spinner *pathName* ?*options*?

INHERITANCE

itk::Widget <- Labeledwidget <- Spinner

STANDARD OPTIONS

background	**borderWidth**	**cursor**	**exportSelection**
foreground	**highlightColor**	**highlightThickness**	**insertBackground**
insertBorderWidth	**insertOffTime**	**insertOnTime**	**insertWidth**
justify	**relief**	**selectBackground**	**selectBorderWidth**
selectForeground	**textVariable**		

See the "options" manual entry for details on the standard options.

ASSOCIATED OPTIONS

show	**state**

See the "entry" manual entry for details on the associated options.

INHERITED OPTIONS

childSitePos	**command**	**fixed**	**focusCommand**
invalid	**textBackground**	**textFont**	**validate**
width			

See the "entryfield" widget manual entry for details on the above inherited options.

disabledForeground	**labelBitmap**	**labelFont**	**labelImage**
labelMargin	**labelPos**	**labelText**	**labelVariable**
state			

See the "labeledwidget" widget manual entry for details on the above inherited options.

WIDGET-SPECIFIC OPTIONS

Name: **arrowOrient**
Class: **Orient**
Command-Line Switch: **-arroworient**

Specifies placement of arrow buttons: **horizontal** or **vertical**. The default is vertical.

Name: **decrement**
Class: **Command**
Command-Line Switch: **-decrement**

Tcl command to be executed when down arrow is pressed.

Name: **increment**
Class: **Command**
Command-Line Switch: **-increment**

Tcl command to be executed when up arrow is pressed.

Name: **repeatDelay**
Class: **RepeatDelay**
Command-Line Switch: **-repeatdelay**

Specifies the initial delay in milliseconds before the spinner repeat action on the arrow buttons engages. The default is 300 milliseconds.

Name: **repeatInterval**
Class: **RepeatInterval**
Command-Line Switch: **-repeatinterval**

Specifies the repeat delay in milliseconds between selections of the arrow buttons. A repeatinterval of 0 disables button repeat. The default is 100 milliseconds.

DESCRIPTION

The **spinner** command creates a spinner widget. The spinner is comprised of an entryfield plus up and down arrow buttons. Arrows may be drawn horizontally or vertically.

METHODS

The **spinner** command creates a new Tcl command whose name is *pathName*. This command may be used to invoke various operations on the widget. It has the following general form:

pathName option ?*arg arg ...*?

Option and the *arg*s determine the exact behavior of the command. The following commands are possible for spinner widgets:

ASSOCIATED METHODS

delete	**get**	**icursor**	**index**
insert	**scan**	**selection**	**xview**

See the "entry" manual entry for details on the associated methods.

INHERITED METHODS

childsite	**clear**	**peek**

See the "entryfield" manual entry for details on the associated methods.

WIDGET-SPECIFIC METHODS

pathName **cget** *option*

Returns the current value of the configuration option given by *option*. *Option* may have any of the values accepted by the **spinner** command.

pathName **configure** ?*option*? ?*value option value ...*?

Query or modify the configuration options of the widget. If no *option* is specified, returns a list describing all of the available options for *pathName* (see **Tk_ConfigureInfo** for information on the format of this list). If *option* is specified with no *value*, then the command returns a list describing the one named option (this list will be identical to the corresponding sublist of the value returned if no *option* is specified). If one or more *option−value* pairs are specified, then the command modifies the given widget option(s) to have the given value(s); in this case the command returns an empty string. *Option* may have any of the values accepted by the **spinner** command.

pathName **down**

Derived classes may overload this method to specialize functionality.

pathName **up**

Derived classes may overload this method to specialize functionality.

COMPONENTS

Name: **downarrow**
Class: **Canvas**

The downarrow component is the downward pointing button of the spinner. See the "canvas" widget manual entry for details on the downarrow component item.

Name: **uparrow**
Class: **Canvas**

The uparrow component is the upward pointing button of the spinner. See the "canvas" widget manual entry for details on the uparrow component item.

EXAMPLE

```
set months {January February March April May June July \
        August September October November December}

proc blockInput {char} {
   return 0
}

proc spinMonth {step} {
   global months

   set index [expr [lsearch $months [.sm get]] + $step]

   if {$index < 0} {set index 11}
   if {$index > 11} {set index 0}

   .sm delete 0 end
   .sm insert 0 [lindex $months $index]
}

spinner .sm -labeltext "Month : " -width 10 -fixed 10 -validate blockInput \
      -decrement {spinMonth -1} -increment {spinMonth 1}
.sm insert 0 January

pack .sm -padx 10 -pady 10
```

ACKNOWLEDGEMENTS:

Ken Copeland <ken@hilco.com>

10/18/95 - Added auto-repeat action to spinner arrow buttons.

AUTHOR

Sue Yockey

KEYWORDS

spinner, widget

NAME

spintime − Create and manipulate time spinner widgets

SYNOPSIS

spintime *pathName* ?*options*?

INHERITANCE

itk::Widget <- Spintime

STANDARD OPTIONS

background **cursor** **foreground** **relief**

See the "options" manual entry for details on the standard options.

ASSOCIATED OPTIONS

textBackground **textFont**

See the "entryfield" manual entry for details on the above associated options.

labelFont **labelMargin**

See the "labeledwidget" manual entry for details on the above associated options.

arrowOrient **repeatDelay** **repeatInterval**

See the "spinner" manual entry for details on the above associated options.

WIDGET-SPECIFIC OPTIONS

Name: **labelPos**
Class: **Position**
Command-Line Switch: **-labelpos**

Specifies the position of the label along the sides of the various spinners: **n**, **e**, **s**, or **w**. The default is w.

Name: **hourLabel**
Class: **Text**
Command-Line Switch: **-hourlabel**

Specifies the text of the label for the hour spinner. The default is "Hour".

Name: **hourOn**
Class: **hourOn**
Command-Line Switch: **-houron**

Specifies whether or not to display the hour spinner in any of the forms acceptable to **Tcl_GetBoolean**. The default is true.

Name: **hourWidth**
Class: **Width**
Command-Line Switch: **-hourwidth**

Specifies the width of the hour spinner in any of the forms acceptable to **Tcl_GetPixels**. The default is 3 pixels.

Name: **militaryOn**
Class: **militaryOn**
Command-Line Switch: **-militaryon**

Specifies use of a 24 hour clock for hour display in any of the forms acceptable to **Tcl_GetBoolean**. The default is true.

Name: **minuteLabel**
Class: **Text**
Command-Line Switch: **-minutelabel**

Specifies the text of the label for the minute spinner. The default is "Minute".

Name: **minuteOn**
Class: **minuteOn**
Command-Line Switch: **-minuteon**

Specifies whether or not to display the minute spinner in any of the forms acceptable to **Tcl_GetBoolean**. The default is true.

Name: **minuteWidth**
Class: **Width**
Command-Line Switch: **-minutewidth**

Specifies the width of the minute spinner in any of the forms acceptable to **Tcl_GetPixels**. The default is 3 pixels.

Name: **orient**
Class: **Orient**
Command-Line Switch: **-orient**

Specifies the orientation of the hour, minute, and second spinners: **vertical** or **horizontal**. The default is horizontal.

Name: **secondLabel**
Class: **Text**
Command-Line Switch: **-secondlabel**

Specifies the text of the label for the second spinner. The default is "Second"

Name: **secondOn**
Class: **secondOn**
Command-Line Switch: **-secondon**

Specifies whether or not to display the second spinner in any of the forms acceptable to **Tcl_GetBoolean**. The default is true.

Name: **secondWidth**
Class: **Width**
Command-Line Switch: **-secondwidth**

Specifies the width of the second spinner in any of the forms acceptable to **Tcl_GetPixels**. The default is 3 pixels.

Name: **timeMargin**
Class: **Margin**
Command-Line Switch: **-timemargin**

Specifies the margin space between the hour, minute, and second spinners is any of the forms accpetable to **Tcl_GetPixels**. The default is 1 pixel.

DESCRIPTION

The **spintime** command creates a set of spinners for use in time value entry. The set includes an hour, minute, and second spinner widget.

METHODS

The **spintime** command creates a new Tcl command whose name is *pathName*. This command may be used to invoke various operations on the widget. It has the following general form:

pathName option ?*arg arg ...*?

Option and the *arg*s determine the exact behavior of the command. The following commands are possible for spintime widgets:

WIDGET-SPECIFIC METHODS

pathName **cget** *option*

Returns the current value of the configuration option given by *option*. *Option* may have any of the values accepted by the **spintime** command.

pathName **configure** ?*option*? ?*value option value ...*?

Query or modify the configuration options of the widget. If no *option* is specified, returns a list describing all of the available options for *pathName* (see **Tk_ConfigureInfo** for information on the format of this list). If *option* is specified with no *value*, then the command returns a list describing the one named option (this list will be identical to the corresponding sublist of the value returned if no *option* is specified). If one or more *option−value* pairs are specified, then the command modifies the given widget option(s) to have the given value(s); in this case the command returns an empty string. *Option* may have any of the values accepted by the **spintime** command.

pathName **get** ?**format**?

Returns the current contents of the spintime widget in a format of string or as an integer clock value using the **-string** and **-clicks** format options respectively. The default is by string. Reference the clock command for more information on obtaining time and its formats.

pathName **show** *time*

Changes the currently displayed time to be that of the time argument. The time may be specified either as a string, an integer clock value or the keyword "now". Reference the clock command for more information on obtaining times and its format.

COMPONENTS

Name: **hour**
Class: **Spinint**

The hour component is the hour spinner of the time spinner. See the SpinInt widget manual entry for details on the hour component item.

Name: **minute**
Class: **Spinint**

The minute component is the minute spinner of the time spinner. See the SpinInt widget manual entry for details on the minute component item.

Name: **second**
Class: **Spinint**

The second component is the second spinner of the time spinner. See the SpinInt widget manual entry for details on the second component item.

EXAMPLE

```
spintime .st
pack .st -padx 10 -pady 10
```

AUTHOR

Sue Yockey

Mark L. Ulferts

KEYWORDS

spintime, spinint, spinner, entryfield, entry, widget

NAME

tabnotebook − create and manipulate tabnotebook widgets

SYNOPSIS

tabnotebook *pathName*? *options*?

INHERITANCE

itk::Widget <- tabnotebook

STANDARD OPTIONS

background	**disabledForeground**	**foreground**	**scrollCommand**
cursor	**font**	**height**	**width**

See the "options" manual entry for details on the standard options.

WIDGET-SPECIFIC OPTIONS

Name: **angle**
Class: **Angle**
Command-Line Switch: **-angle**

Specifes the angle of slope from the inner edge to the outer edge of the tab. An angle of 0 specifies square tabs. Valid ranges are 0 to 45 degrees inclusive. Default is 15 degrees. If **tabPos** is e or w, this option is ignored.

Name: **auto**
Class: **Auto**
Command-Line Switch: **-auto**

Specifies whether to use the automatic packing/unpacking algorithm of the notebook. A value of true indicates that page frames will be unpacked and packed acoording to the algorithm described in the select command. A value of **false** leaves the current page packed and subsequent **selects**, **next**, or **previous** commands do not switch pages automatically. In either case the page's associated command (see the **add** command's description of the command option) is invoked. The value may have any of the forms accepted by the **Tcl_GetBoolean**, such as **true**, **false**, **0**, **1**, **yes**, or **no**.

Name: **backdrop**
Class: **Backdrop**
Command-Line Switch: **-backdrop**

Specifies a background color to use when filling in the backdrop area behind the tabs.

Name: **background**
Class: **Background**
Command-Line Switch: **-background**

Specifies a background color to use for displaying a page and its associated tab. This can be thought of as the selected tab background color, since the tab associated with the selected page is the selected tab.

Name: **bevelAmount**
Class: **BevelAmount**
Command-Line Switch: **-bevelamount**

Specifes the size of tab corners. A value of 0 with **angle** set to 0 results in square tabs. A **bevelAmount** of 4, means that the tab will be drawn with angled corners that cut in 4 pixels from the edge of the tab. The default is 0.

Name: **borderWidth**
Class: **BorderWidth**
Command-Line Switch: **-borderwidth**

Specifies the width of shadowed border to place around the notebook area of the tabnotebook. The default value is 2.

Name: **disabledForeground**
Class: **DisabledForeground**
Command-Line Switch: **-disabledforeground**

Specifies a foreground color to use for displaying a tab's label when its **state** is disabled.

Name: **equalTabs**
Class: **EqualTabs**
Command-Line Switch: **-equaltabs**

Specifies whether to force tabs to be equal sized or not. A value of **true** means constrain tabs to be equal sized. A value of **false** allows each tab to size based on the text label size. The value may have any of the forms accepted by the **Tcl_GetBoolean**, such as **true**, **false**, **0**, **1**, **yes**, or **no**.

For horizontally positioned tabs (**tabpos** is either **s** or **n**), **true** forces all tabs to be equal width (the width being equal to the longest label plus any **padX** specified). Horizontal tabs are always equal in height.

For vertically positioned tabs (**tabpos** is either **w** or **e**), **true** forces all tabs to be equal height (the height being equal to the height of the label with the largest font). Vertically oriented tabs are always equal in width.

Name: **foreground**
Class: **Foreground**
Command-Line Switch: **-foreground**

Specifies a foreground color to use for displaying a page and its associated tab label. This can be thought of as the selected tab background color, since the tab associated with the selected page is the selected tab.

Name: **gap**
Class: **Gap**
Command-Line Switch: **-gap**

Specifies the amount of pixel space to place between each tab. Value may be any pixel offset value. In addition, a special keyword **overlap** can be used as the value to achieve a standard overlap of tabs. This value may have any of the forms acceptable to **Tk_GetPixels**.

Name: **margin**
Class: **Margin**
Command-Line Switch: **-Bmargin**

Specifies the amount of space to place between the outside edge of the tabnotebook and the outside edge of its tabs. If **tabPos** is **s**, this is the amount of space between the bottom edge of the tabnotebook and the bottom edge of the set of tabs. If **tabPos** is **n**, this is the amount of space between the top edge of the tabnotebook and the top edge of the set of tabs. If **tabPos** is **e**, this is the amount of space between the right edge of the tabnotebook and the right edge of the set of tabs. If **tabPos** is **w**, this is the amount of space between the left edge of the tabnotebook and the left edge of the set of tabs. This value may have any of the forms acceptable to **Tk_GetPixels**.

Name: **padX**
Class: **PadX**
Command-Line Switch: **-padx**

Specifies a non-negative value indicating how much extra space to request for a tab around its label in the X-direction. When computing how large a window it needs, the tab will add this amount to the width it would normally need The tab will end up with extra internal space to the

left and right of its text label. This value may have any of the forms acceptable to **Tk_GetPixels**.

Name: **padY**
Class: **PadY**
Command-Line Switch: **-pady**

Specifies a non-negative value indicating how much extra space to request for a tab around its label in the Y-direction. When computing how large a window it needs, the tab will add this amount to the height it would normally need The tab will end up with extra internal space to the top and bottom of its text label. This value may have any of the forms acceptable to **Tk_GetPixels**.

Name: **raiseSelect**
Class: **RaiseSelect**
Command-Line Switch: **-raiseselect**

Specifes whether to slightly raise the selected tab from the rest of the tabs. The selected tab is drawn 2 pixels closer to the outside of the tabnotebook than the unselected tabs. A value of **true** says to raise selected tabs, a value of **false** turns this feature off. The default is **false**. The value may have any of the forms accepted by the **Tcl_GetBoolean**, such as **true**, **false**, **0**, **1**, **yes**, or **no**.

Name: **start**
Class: **Start**
Command-Line Switch: **-start**

Specifies the amount of space to place between the left or top edge of the tabnotebook and the starting edge of its tabs. For horizontally positioned tabs, this is the amount of space between the left edge of the notebook and the left edge of the first tab. For vertically positioned tabs, this is the amount of space between the top of the notebook and the top of the first tab. This value may change if the user performs a MButton-2 scroll on the tabs. This value may have any of the forms acceptable to **Tk_GetPixels**.

Name: **state**
Class: **State**
Command-Line Switch: **-state**

Sets the active state of the tabnotebook. Specifying **normal** allows all pages to be selectable. Specifying **disabled** disables the notebook causing all page tabs to be drawn in the **disabledForeground** color.

Name: **tabBackground**
Class: **TabBackground**
Command-Line Switch: **-tabbackground**

Specifies a background color to use for displaying tab backgrounds when they are in their unselected state. This is the background associated with tabs on all pages other than the selected page.

Name: **tabBorders**
Class: **TabBorders**
Command-Line Switch: **-tabborders**

Specifies whether to draw the borders of tabs that are not selected. Specifying **true** (the default) draws these borders, specifying **false** draws only the border around the selected tab. The value may have any of the forms accepted by the **Tcl_GetBoolean**, such as **true**, **false**, **0**, **1**, **yes**, or **no**.

Name: **tabForeground**
Class: **TabForeground**
Command-Line Switch: **-tabforeground**

Specifies a foreground color to use for displaying tab labels when they are in their unselected state. This is the foreground associated with tabs on all pages other than the selected page.

Name: **tabPos**
Class: **TabPos**
Command-Line Switch: **-tabpos**

Specifies the location of the set of tabs in relation to the notebook area. Must be n, s, e, or w. Defaults to s.

DESCRIPTION

The **tabnotebook** command creates a new window (given by the pathName argument) and makes it into a **tabnotebook** widget. Additional options, described above may be specified on the command line or in the option database to configure aspects of the tabnotebook such as its colors, font, and text. The tabnotebook command returns its pathName argument. At the time this command is invoked, there must not exist a window named pathName, but pathName's parent must exist.

A **tabnotebook** is a widget that contains a set of tabbed pages. It displays one page from the set as the selected page. A Tab displays the label for the page to which it is attached and serves as a page selector. When a page's tab is selected, the page's contents are displayed in the page area. The selected tab has a three-dimensional effect to make it appear to float above the other tabs. The tabs are displayed as a group along either the left, top, right, or bottom edge. When first created a tabnotebook has no pages. Pages may be added or deleted using widget commands described below.

A special option may be provided to the tabnotebook. The **-auto** option specifies whether the tabnotebook will automatically handle the unpacking and packing of pages when pages are selected. A value of true signifies that the notebook will automatically manage it. This is the default value. A value of false signifies the notebook will not perform automatic switching of pages.

NOTEBOOK PAGES

A tabnotebook's pages area contains a single child site frame. When a new page is created it is a child of this frame. The page's child site frame serves as a geometry container for applications to pack widgets into. It is this frame that is automatically unpacked or packed when the auto option is true. This creates the effect of one page being visible at a time. When a new page is selected, the previously selected page's child site frame is automatically unpacked from the tabnotebook's child site frame and the newly selected page's child site is packed into the tabnotebook's child site frame.

However, sometimes it is desirable to handle page changes in a different manner. By specifying the **auto** option as **false**, child site packing can be disabled and done differently. For example, all widgets might be packed into the first page's child site **frame**. Then when a new page is selected, the application can reconfigure the widgets and give the appearance that the page was flipped.

In both cases the command option for a page specifies a Tcl Command to execute when the page is selected. In the case of **auto** being **true**, it is between the unpacking of the previously selected page and the packing of the newly selected page.

Notebook pages can also be controlled with scroll bars or other widgets that obey the **scrollcommand** protocol. By giving a scrollbar a **-command** to call the tabnotebook's **select** method, the tabnotebook can be controlled with a scrollbar.

The notebook area is implemented with the notebook mega widget.

TABS

Tabs appear along the edge of the notebook area. Tabs are drawn to appear attached to their associated page. When a tab is clicked on, the associated page is selected and the tab is drawn as raised above all other tabs and as a seamless part of its notebook page. Tabs can be controlled in their location along the edges, the angle tab sides are drawn with, gap between tabs, starting margin of tabs, internal padding around text labels in a tab, the font, and its label.

The Tab area is implemented with the **tabset** mega widget. See **tabset(1)**. Tabs may be oriented along either the north, south, east, or west sides with the **tabPos** option. North and south tabs may appear as angled, square, or bevelled. West and east tabs may appear as square or bevelled. By changing tab gaps, tab angles, bevelling, orientations, colors, fonts, start locations, and margins; tabs may appear in a wide variety of styles. For example, it is possible to implement Microsoft-style tabs, Borland property tab styles, or Borland Delphi style tabs all with the same tabnotebook.

WIDGET-SPECIFIC METHODS

The **tabnotebook** command creates a new Tcl command whose name is *pathName*. This command may be used to invoke various operations on the widget. It has the following general form:

pathName option ?*arg arg ...*?

option and the *arg*s determine the exact behavior of the command.

Many of the widget commands for a notebook take as one argument an indicator of which page of the notebook to operate on. These indicators are called indexes and may be specified in any of the following forms:

number Specifies the page numerically, where 0 corresponds to the first page in the notebook, 1 to the second, and so on.

select Specifies the currently selected page's index. If no page is currently selected, the value -1 is returned.

end Specifes the last page in the tabnotebook's index. If the notebook is empty this will return -1.

pattern If the index doesn't satisfy any of the above forms, then this form is used. Pattern is pattern-matched against the label of each page in the notebook, in order from the first to the last page, until a matching entry is found. The rules of Tcl_StringMatch are used. The following commands are possible for tabnotebook widgets:

pathName **add** ?*option value option value ...*?

Add a new page at the end of the tabnotebook. A new child site frame is created. Returns the child site pathName. If additional arguments are present, they specify any of the following options:

-angle *value*

Specifes the angle of slope from the inner edge to the outer edge of the tab. An angle of 0 specifies square tabs. Valid ranges are 0 to 45 degrees inclusive. Default is 15 degrees. If this option is specified as an empty string (the default), then the angle option for the overall tabnotebook is used. This is generally only set at the tabnotebook level. Tabs normally will want to share the same angle value.

-background *value*

Specifies a background color to use for displaying tabs when they are selected and for displaying the current page. If this option is specified as an empty string (the default), then the background option for the overall tabnotebook is used.

-bevelamount *value*

Specifes the size of tab corners. A value of 0 with angle set to 0 results in square tabs. A bevelAmount of 4, means that the tab will be drawn with angled corners that cut in 4 pixels from the edge of the tab. The default is 0. This is generally only set at the tabnotebook level. Tabs normally will want to share the same bevelAmount.

-bitmap *value*

If label is a non-empty string, specifies a bitmap to display in this page's tab. Bitmap may be of any of the forms accepted by Tk_GetPixmap.

-command *value*

Specifies a Tcl command to be executed when this page is selected. This allows the

programmer a hook to reconfigure this page's widgets or any other page's widgets.

If the tabnotebook has the auto option set to true, when a page is selected this command will be called immediately after the previously selected page is unpacked and immediately before this page is selected. The index value select is valid during this Tcl command. 'index select' will return this page's page number.

If the auto option is set to false, when a page is selected the unpack and pack calls are bypassed. This Tcl command is still called.

-disabledforeground *value*

Specifies a foreground color to use for displaying tab labels when tabs are in their disable state. If this option is specified as an empty string (the default), then the disabledforeground option for the overall tabnotebook is used.

-font *value*

Specifies the font to use when drawing a text label on a page tab. If this option is specified as an empty string then the font option for the overall tabnotebook is used..

-foreground *value*

Specifies a foreground color to use for displaying tab labels when they are selected. If this option is specified as an empty string (the default), then the foreground option for the overall tabnotebook is used.

-label *value*

Specifies a string to display as an identifying label for a notebook page. This label serves as an additional identifier used to reference the page. This label may be used for the index value in widget commands.

-tabbackground *value*

Specifies a background color to use for displaying a tab when it is not elected. If this option is specified as an empty string (the default), then the tabBackground option for the overall tabnotebook is used.

-tabforeground *value*

Specifies a foreground color to use for displaying the tab's text label when it is not selected. If this option is specified as an empty string (the default), then the tabForeground option for the overall tabnotebook is used.

-padx *value*

Specifies a non-negative value indicating how much extra space to request for a tab around its label in the X-direction. When computing how large a window it needs, the tab will add this amount to the width it would normally need The tab will end up with extra internal space to the left and right of its text label. This value may have any of the forms acceptable to Tk_GetPixels. If this option is specified as an empty string (the default), then the padX option for the overall tabnotebook is used

-pady *value*

Specifies a non-negative value indicating how much extra space to request for a tab around its label in the Y-direction. When computing how large a window it needs, the tab will add this amount to the height it would normally need The tab will end up with extra internal space to the top and bottom of its text label. This value may have any of the forms acceptable to Tk_GetPixels. If this option is specified as an empty string (the default), then the padY option for the overall tabnotebook is used

-state *value*

Specifies one of two states for the page: normal or disabled. In normal state unselected tabs are displayed using the tabforeground and tabbackground option from the

tabnotebook or the page. Selected tabs and pages are displayed using the foreground and background option from the tabnotebook or the page. The disabled state means that the page and its tab is insensitive: it doesn't respond to mouse button presses or releases. In this state the entry is displayed according to its disabledForeground option for the tabnotebook and the background/tabbackground option from the page or the tabnotebook.

pathName **childSite** ?*index*?

If passed no arguments, returns a list of pathNames for all the pages in the tabnotebook. If the tab notebook is empty, an empty list is returned

If *index* is passed, it returns the *pathName* for the page's child site **frame** specified by *index*. Widgets that are created with this *pathName* will be displayed when the associated page is selected. If *index* is not a valid index, an empty string is returned.

pathName **configure** ?*option*? ?*value option value* ...?

Query or modify the configuration options of the widget. If no *option* is specified, returns a list describing all of the available options for *pathName* (see **Tk_ConfigureInfo** for information on the format of this list). If option is specified with no value, then the command returns a list describing the one named option (this list will be identical to the corresponding sublist of the value returned if no option is specified). If one or more option-value pairs are specified, then the command modifies the given widget option(s) to have the given value(s); in this case the command returns an empty string. *Option* may have any of the values accepted by the tabnotebook command.

pathName **delete** *index1* ?*index2*?

Delete all of the pages between *index1* and *index2* inclusive. If *index2* is omitted then it defaults to *index1*. Returns an empty string.

pathName **index** *index*

Returns the numerical index corresponding to *index*.

pathName **insert** *index* ?*option value option value* ...?

Insert a new page in the tabnotebook before the page specified by *index*. A new child site **frame** is created. The additional arguments are the same as for the **add** command. Returns the child site *pathName*.

pathName **next**

Advances the selected page to the next page (order is determined by insertion order). If the currently selected page is the last page in the notebook, the selection wraps around to the first page in the notebook. It behaves as if the user selected the new page.

For notebooks with **auto** set to **true** the current page's child site is unpacked from the notebook's child site frame. Then the next page's child site is packed into the notebook's child site frame. The Tcl command given with the command option will be invoked between these two operations.

For notebooks with **auto** set to **false** the Tcl command given with the command option will be invoked.

pathName **pageconfigure** *index* ?*option*? ?*value option value* ...?

This command is similar to the **configure** command, except that it applies to the options for an individual page, whereas configure applies to the options for the tabnotebook as a whole. *Options* may have any of the values accepted by the add widget command. If options are specified, options are modified as indicated in the command and the command returns an empty string. If no options are specified, returns a list describing the current options for page index (see **Tk_ConfigureInfo** for information on the format of this list).

pathName **prev**

Moves the selected page to the previous page (order is determined by insertion order). If the

currently selected page is the first page in the notebook, the selection wraps around to the last page in the notebook. It behaves as if the user selected the new page.

For notebooks with **auto** set to **true** the current page's child site is unpacked from the notebook's child site **frame**. Then the previous page's child site is packed into the notebook's child site frame. The Tcl command given with the command option will be invoked between these two operations.

For notebooks with **auto** set to **false** the Tcl command given with the command option will be invoked.

pathName **select** *index*
Selects the page specified by *index* as the currently selected page. It behaves as if the user selected the new page.

For notebooks with **auto** set to **true** the current page's child site is unpacked from the notebook's child site frame. Then the *index* page's child site is packed into the notebook's child site frame. The Tcl command given with the command option will be invoked between these two operations.

For notebooks with **auto** set to **false** the Tcl command given with the command option will be invoked.

pathName **view**
Returns the currently selected page. This command is for compatibility with the **scrollbar** widget.

pathName **view** *index*
Selects the page specified by *index* as the currently selected page. This command is for compatibility with the **scrollbar** widget.

pathName **view moveto** *fraction*
Uses the *fraction* value to determine the corresponding page to move to. This command is for compatibility with the **scrollbar** widget.

pathName **view scroll** *num what*
Uses the *num* value to determine how many pages to move forward or backward (*num* can be negative or positive). The *what* argument is ignored. This command is for compatibility with the **scrollbar** widget.

COMPONENTS

Generally all behavior of the internal components, **tabset** and **notebook** are controlled via the **pageconfigure** method. The following section documents these two components.

Name: **tabset**
Class: **Tabset**

This is the tabset component. It implements the tabs that are associated with the notebook component.

See the "**Tabset**" widget manual entry for details on the **tabset** component item.

Name: **notebook**
Class: **Notebook**

This is the notebook component. It implements the notebook that contains the pages of the tabnotebook.

See the "**Notebook**" widget manual entry for details on the **notebook** component item.

EXAMPLE

Following is an example that creates a tabnotebook with two pages.

```
# Create the tabnotebook widget and pack it.
  tabnotebook .tn -width 100 -height 100
```

```
pack .tn \
        -anchor nw \
        -fill both \
        -expand yes \
        -side left \
        -padx 10 \
        -pady 10

# Add two pages to the tabnotebook,
# labelled "Page One" and "Page Two"
.tn add -label "Page One"
.tn add -label "Page Two"

# Get the child site frames of these two pages.
set page1CS [.tn childsite 0]
set page2CS [.tn childsite "Page Two"]

# Create buttons on each page of the tabnotebook.
button $page1CS.b -text "Button One"
pack $page1CS.b
button $page2CS.b -text "Button Two"
pack $page2CS.b

# Select the first page of the tabnotebook.
.tn select 0
```

AUTHOR

Bill W. Scott

KEYWORDS

tab tabset notebook tabnotebook page

NAME

tabset − create and manipulate tabs as as set

SYNOPSIS

tabset *pathName* ?*options*?

INHERITANCE

itk::Widget <- tabset

STANDARD OPTIONS

background	**font**	**selectBackground**	**cursor**
foreground	**selectForeground**	**disabledForeground**	**height**
width			

See the "options" manual entry for details on the standard options.

WIDGET-SPECIFIC OPTIONS

Name: **angle**
Class: **Angle**
Command-Line Switch: **-angle**

Specifes the angle of slope from the inner edge to the outer edge of the tab. An angle of 0 specifies square tabs. Valid ranges are 0 to 45 degrees inclusive. Default is 15 degrees. If tabPos is e or w, this option is ignored.

Name: **backdrop**
Class: **Backdrop**
Command-Line Switch: **-backdrop**

Specifies a background color to use when filling in the area behind the tabs.

Name: **bevelAmount**
Class: **BevelAmount**
Command-Line Switch: **-bevelamount**

Specifes the size of tab corners. A value of 0 with angle set to 0 results in square tabs. A **bevelAmount** of 4, means that the tab will be drawn with angled corners that cut in 4 pixels from the edge of the tab. The default is 0.

Name: **command**
Class: **Command**
Command-Line Switch: **-command**

Specifes the prefix of a Tcl command to invoke to change the view in the widget associated with the tabset. When a user selects a tab, a Tcl command is invoked. The actual command consists of this option followed by a space and a number. The number is the numerical index of the tab that has been selected.

Name: **equalTabs**
Class: **EqualTabs**
Command-Line Switch: **-equaltabs**

Specifies whether to force tabs to be equal sized or not. A value of **true** means constrain tabs to be equal sized. A value of **false** allows each tab to size based on the text label size. The value may have any of the forms accepted by the **Tcl_GetBoolean**, such as **true**, **false**, **0**, **1**, **yes**, or **no**.

For horizontally positioned tabs (**tabPos** is either **s** or **n**), **true** forces all tabs to be equal width (the width being equal to the longest label plus any padX specified). Horizontal tabs are always equal in height.

For vertically positioned tabs (**tabPos** is either **w** or **e**), **true** forces all tabs to be equal height (the

height being equal to the height of the label with the largest font). Vertically oriented tabs are always equal in width.

Name: **gap**
Class: **Gap**
Command-Line Switch: **-gap**

Specifies the amount of pixel space to place between each tab. Value may be any pixel offset value. In addition, a special keyword **overlap** can be used as the value to achieve a standard overlap of tabs. This value may have any of the forms acceptable to **Tk_GetPixels**.

Name: **margin**
Class: **Margin**
Command-Line Switch: **-margin**

Specifies the amount of space to place between the outside edge of the tabset and the outside edge of its tabs. If **tabPos** is **s**, this is the amount of space between the bottom edge of the tabset and the bottom edge of the set of tabs. If **tabPos** is **n**, this is the amount of space between the top edge of the tabset and the top edge of the set of tabs. If **tabPos** is **e**, this is the amount of space between the right edge of the tabset and the right edge of the set of tabs. If **tabPos** is **w**, this is the amount of space between the left edge of the tabset and the left edge of the set of tabs. This value may have any of the forms acceptable to **Tk_GetPixels**.

Name: **padX**
Class: **PadX**
Command-Line Switch: **-padx**

Specifies a non-negative value indicating how much extra space to request for a tab around its label in the X-direction. When computing how large a window it needs, the tab will add this amount to the width it would normally need The tab will end up with extra internal space to the left and right of its text label. This value may have any of the forms acceptable to **Tk_GetPixels**.

Name: **padY**
Class: **PadY**
Command-Line Switch: **-pady**

Specifies a non-negative value indicating how much extra space to request for a tab around its label in the Y-direction. When computing how large a window it needs, the tab will add this amount to the height it would normally need The tab will end up with extra internal space to the top and bottom of its text label. This value may have any of the forms acceptable to **Tk_GetPixels**.

Name: **raiseSelect**
Class: **RaiseSelect**
Command-Line Switch: **-raiseselect**

Specifes whether to slightly raise the selected tab from the rest of the tabs. The selected tab is drawn 2 pixels closer to the outside edge of the tabset than the unselected tabs. A value of true says to raise selected tabs, a value of false turns this off. The default is false. The value may have any of the forms accepted by the **Tcl_GetBoolean**, such as **true**, **false**, **0**, **1**, **yes**, or **no**.

Name: **start**
Class: **Start**
Command-Line Switch: **-start**

Specifies the amount of space to place between the left or top edge of the tabset and the starting edge of its tabs. For horizontally positioned tabs, this is the amount of space between the left edge of the tabset and the left edge of the first tab. For vertically positioned tabs, this is the amount of space between the top of the tabset and the top of the first tab. This value may change if the user performs a MButton-2 scroll on the tabs. This value may have any of the forms acceptable to

Tk_GetPixels.

Name: **state**
Class: **State**
Command-Line Switch: **-state**

Sets the active state of the tabset. Specifying **normal** allows all tabs to be selectable. Specifying **disabled** disables the tabset causing all tabs to be drawn in the disabledForeground color.

Name: **tabBorders**
Class: **TabBorders**
Command-Line Switch: **-tabborders**

Specifies whether to draw the borders of tabs that are not selected. Specifying true (the default) draws these borders, specifying false draws only the border around the selected tab. The value may have any of the forms accepted by the **Tcl_GetBoolean**, such as **true**, **false, 0**, **1**, **yes**, or **no**.

Name: **tabPos**
Class: **TabPos**
Command-Line Switch: **-tabpos**

Specifies the location of the set of tabs in relation to another widget. Must be **n**, **s**, **e**, or **w**. Defaults to **s**. North tabs open downward, South tabs open upward. West tabs open to the right, east tabs open to the left.

DESCRIPTION

The **tabset** command creates a new window (given by the pathName argument) and makes it into a **tabset** widget. Additional *options*, described above may be specified on the command line or in the option database to configure aspects of the tabset such as its colors, font, and text. The **tabset** command returns its *pathName* argument. At the time this command is invoked, there must not exist a window named *pathName*, but pathName's parent must exist.

A **tabset** is a widget that contains a set of Tab buttons. It displays these tabs in a row or column depending on it tabpos. When a tab is clicked on, it becomes the only tab in the tab set that is selected. All other tabs are deselected. The Tcl command prefix associated with this tab (through the command tab configure option) is invoked with the tab index number appended to its argument list. This allows the tabset to control another widget such as a Notebook.

TABS

Tabs are drawn to appear attached to another widget. The tabset draws an edge boundary along one of its edges. This edge is known as the attachment edge. This edge location is dependent on the value of **tabPos**. For example, if **tabPos** is **s**, the attachment edge wil be on the top side of the tabset (in order to attach to the bottom or south side of its attached widget). The selected tab is draw with a 3d relief to appear above the other tabs. This selected tab "opens" toward attachment edge.

Tabs can be controlled in their location along the edges, the angle that tab sides are drawn with, gap between tabs, starting margin of tabs, internal padding around labels in a tab, the font, and its text or bitmap.

WIDGET-SPECIFIC METHODS

The **tabset** command creates a new Tcl command whose name is *pathName*. This command may be used to invoke various operations on the widget. It has the following general form:

pathName option ?*arg arg* ...?

option and the *arg*s determine the exact behavior of the command.

Many of the widget commands for a tabset take as one argument an indicator of which tab of the tabset to operate on. These indicators are called indexes and may be specified in any of the following forms:

number Specifies the tab numerically, where 0 corresponds to the first tab in the tab set, 1 to the second, and so on.

select Specifies the currently selected tab's index. If no tab is currently selected, the value -1 is returned.

end Specifes the last tab in the tabset's index. If the tabset is empty this will return -1.

pattern If the index doesn't satisfy any of the above forms, then this form is used. Pattern is pattern-matched against the label of each tab in the tabset, in order from the first to the last tab, until a matching entry is found. The rules of Tcl_StringMatch are used.

The following commands are possible for tabset widgets:

pathName **add** ?*option value option value* ...?

Add a new tab at the end of the tabset. Returns the child site *pathName*. If additional arguments are present, they specify any of the following options:

-angle *value*

Specifes the angle of slope from the inner edge to the outer edge of the tab. An angle of 0 specifies square tabs. Valid ranges are 0 to 45 degrees inclusive. Default is 15 degrees. If this option is specified as an empty string (the default), then the angle option for the overall tabset is used.

-background *value*

Specifies a background color to use for displaying tabs when they are in their normal state (unselected). If this option is specified as an empty string (the default), then the background option for the overall tabset is used.

-bevelamount *value*

Specifes the size of tab corners. A value of 0 with angle set to 0 results in square tabs. A bevelAmount of 4, means that the tab will be drawn with angled corners that cut in 4 pixels from the edge of the tab. The default is 0. This is generally only set at the tabset configuration level. Tabs normally will want to share the same bevelAmount.

-bitmap *value*

If label is a non-empty string, specifies a bitmap to display in the tab. Bitmap may be of any of the forms accepted by Tk_GetBitmap.

-disabledforeground *value*

Specifies a foreground color to use for displaying tab labels when tabs are in their disable state. If this option is specified as an empty string (the default), then the disabledforeground option for the overall tabset is used.

-font *value*

Specifies the font to use when drawing the label on a tab. If this option is specified as an empty string then the font option for the overall tabset is used.

-foreground *value*

Specifies a foreground color to use for displaying tab labels when tabs are in their normal unselected state. If this option is specified as an empty string (the default), then the foreground option for the overall tabset is used.

-image *value*

If label is a non-empty string, specifies an image to display in the tab. Image must have been created with the image create command. Typically, if the image option is specified then it overrides other options that specify a bitmap or textual

value to display in the widget; the image option may be reset to an empty string to re-enable a bitmap or text display.

-label *value*
Specifies a text string to be placed in the tabs label. If this value is set, the bitmap option is overridden and this option is used instead. This label serves as an additional identifier used to reference the tab. This label may be used for the index value in widget commands.

-selectbackground *value*
Specifies a background color to use for displaying the selected tab. If this option is specified as an empty string (the default), then the selectBackground option for the overall tabset is used.

-selectforeground *value*
Specifies a foreground color to use for displaying the selected tab. If this option is specified as an empty string (the default), then the selectForeground option for the overall tabset is used.

-padx *value*
Specifies a non-negative value indicating how much extra space to request for a tab around its label in the X-direction. When computing how large a window it needs, the tab will add this amount to the width it would normally need The tab will end up with extra internal space to the left and right of its text label. This value may have any of the forms acceptable to Tk_GetPixels. If this option is specified as an empty string (the default), then the padX option for the overall tabset is used

-pady *value*
Specifies a non-negative value indicating how much extra space to request for a tab around its label in the Y-direction. When computing how large a window it needs, the tab will add this amount to the height it would normally need The tab will end up with extra internal space to the top and bottom of its text label. This value may have any of the forms acceptable to Tk_GetPixels. If this option is specified as an empty string (the default), then the padY option for the overall tabset is used

-state *value*
Sets the state of the tab. Specifying normal allows this tab to be selectable. Specifying disabled disables the this tab causing its tab label to be drawn in the disabledForeground color. The tab will not respond to events until the state is set back to normal.

pathName **configure** ?*option*? ?*value option value ...*?
Query or modify the configuration options of the widget. If no *option* is specified, returns a list describing all of the available options for *pathName* (see **Tk_ConfigureInfo** for information on the format of this list). If option is specified with no value, then the command returns a list describing the one named option (this list will be identical to the corresponding sublist of the value returned if no option is specified). If one or more option-value pairs are specified, then the command modifies the given widget option(s) to have the given value(s); in this case the command returns an empty string. *Option* may have any of the values accepted by the tabset command.

pathName **delete** *index1* ?*index2*?
Delete all of the tabs between *index1* and *index2* inclusive. If *index2* is omitted then it defaults to *index1*. Returns an empty string.

pathName **index** *index*
Returns the numerical index corresponding to *index*.

pathName **insert** *index* ?*option value option value ...*?
Insert a new tab in the tabset before the tab specified by *index*. The additional arguments are the same as for the **add** command. Returns the tab's *pathName*.

pathName **next**
Advances the selected tab to the next tab (order is determined by insertion order). If the currently selected tab is the last tab in the tabset, the selection wraps around to the first tab. It behaves as if the user selected the next tab.

pathName **tabconfigure** *index* ?*option*? ?*value*?
This command is similar to the **configure** command, except that it applies to the options for an individual tab, whereas configure applies to the options for the tabset as a whole. Options may have any of the values accepted by the **add** widget command. If options are specified, options are modified as indicated in the command and the command returns an empty string. If no options are specified, returns a list describing the current options for tab index (see **Tk_ConfigureInfo** for information on the format of this list).

pathName **prev**
Moves the selected tab to the previous tab (order is determined by insertion order). If the currently selected tab is the first tab in the tabset, the selection wraps around to the last tab in the tabset. It behaves as if the user selected the previous tab.

pathName **select** *index*
Selects the tab specified by *index* as the currently selected tab. It behaves as if the user selected the new tab.

EXAMPLE

Following is an example that creates a tabset with two tabs and a list box that the tabset controls. In addition selecting an item from the list also selects the corresponding tab.

```
# Define a proc that knows how to select an item
# from a list given an index from the tabset -command callback.
  proc selectItem { item } {
    .l selection clear [.l curselection]
    .l selection set $item
    .l see $item
}

# Define a proc that knows how to select a tab
# given a y pixel coordinate from the list..
  proc selectTab { y } {
    set whichItem [.l nearest $y]
    .ts select $whichItem
}

# Create a listbox with two items (one and two)
# and bind button 1 press to the selectTab procedure.
  listbox .l -selectmode single -exportselection false
  .l insert end one
  .l insert end two
  .l selection set 0
  pack .l
  bind .l <ButtonPress-1> { selectTab %y }
```

```
# Create a tabset, set its -command to call selectItem
# Add two labels to the tabset (one and two).
  tabset .ts -command selectItem
  .ts add -label 1
  .ts add -label 2
  .ts select 0
  pack .ts -fill x -expand no
```

AUTHOR

Bill W. Scott

KEYWORDS

tab tabset notebook tabnotebook

NAME

timeentry − Create and manipulate a timeentry widget

SYNOPSIS

timeentry *pathName* ?*options*?

INHERITANCE

itk::Widget <- LabeledWidget <- Timefield <- Timeentry

STANDARD OPTIONS

background	**borderWidth**	**cursor**	**exportSelection**
foreground	**highlightColor**	**highlightThickness**	**insertBackground**
justify	**relief**		

See the "options" manual entry for details on the standard options.

INHERITED OPTIONS

disabledForeground	**labelBitmap**	**labelFont**	**labelImage**
labelMargin	**labelPos**	**labelText**	**labelVariable**
state			

See the "labeledwidget" class manual entry for details on these inherited options.

command	**format**	**seconds**	**textBackground**
textFont			

See the "timefield" class manual entry for details on these inherited options.

ASSOCIATED OPTIONS

hourRadius	**hourColor**	**minuteRadius**	**minuteColor**
pivotRadius	**pivotColor**	**secondRadius**	**secondColor**
clockColor	**clockStipple**	**tickColor**	**watchHeight**
watchWidth			

See the "watch" manual entry for details on the associated options.

WIDGET-SPECIFIC OPTIONS

Name: **closeText**
Class: **Text**
Command-Line Switch: **-closetext**

Specifies the text to be displayed on the close button of the watch popup. The default is Close.

Name: **grab**
Class: **Grab**
Command-Line Switch: **-grab**

Specifies the grab level, **local** or **global**, to be obtained before bringing up the popup watch. The default is global. For more information concerning grab levels, consult the documentation for Tk's **grab** command.

Name: **icon**
Class: **Icon**
Command-Line Switch: **-icon**

Specifies the watch icon image to be used in the timeentry. This image must have been created previously with the **image create** command. Should one not be provided, then one will be generated, pixmap if possible, bitmap otherwise.

Name: **state**
Class: **State**
Command-Line Switch: **-state**

Specifies the state of the widget which may be **disabled** or **normal**. A disabled state prevents selection of the timefield or time icon button.

DESCRIPTION

The **timeentry** command creates a time entry field with a popup watch by combining the timefield and watch widgets together. This allows a user to enter the time via the keyboard or by using the mouse and selecting the watch icon which brings up a popup watch.

METHODS

The **timeentry** command creates a new Tcl command whose name is *pathName*. This command may be used to invoke various operations on the widget. It has the following general form:

pathName option ?*arg arg ...*?

Option and the *arg*s determine the exact behavior of the command. The following commands are possible for timeentry widgets:

INHERITED METHODS

get **isvalid** **show**

See the "timefield" manual entry for details on the associated methods.

WIDGET-SPECIFIC METHODS

pathName **cget** *option*

Returns the current value of the configuration option given by *option*. *Option* may have any of the values accepted by the **timeentry** command.

pathName **configure** ?*option*? ?*value option value ...*?

Query or modify the configuration options of the widget. If no *option* is specified, returns a list describing all of the available options for *pathName* (see **Tk_ConfigureInfo** for information on the format of this list). If *option* is specified with no *value*, then the command returns a list describing the one named option (this list will be identical to the corresponding sublist of the value returned if no *option* is specified). If one or more *option−value* pairs are specified, then the command modifies the given widget option(s) to have the given value(s); in this case the command returns an empty string. *Option* may have any of the values accepted by the **timeentry** command.

COMPONENTS

Name: **label**
Class: **Label**

The label component provides a label component to used to identify the time. See the "label" widget manual entry for details on the label component item.

Name: **iconbutton**
Class: **Label**

The iconbutton component provides a labelbutton component to act as a lightweight button displaying the watch icon. Upon pressing the labelbutton, the watch appears. See the "label" widget manual entry for details on the labelbutton component item.

Name: **time**
Class: **Entry**

The time component provides the entry field for time input and display. See the "entry" widget manual entry for details on the time component item.

EXAMPLE

```
timeentry .te
pack .te
```

AUTHOR

Mark L. Ulferts

KEYWORDS

timeentry, widget

NAME

timefield − Create and manipulate a time field widget

SYNOPSIS

timefield *pathName* ?*options*?

INHERITANCE

itk::Widget <- LabeledWidget <- timefield

STANDARD OPTIONS

background	**borderWidth**	**cursor**	**exportSelection**
foreground	**highlightColor**	**highlightThickness**	**insertBackground**
justify	**relief**		

See the "options" manual entry for details on the standard options.

INHERITED OPTIONS

disabledForeground	**labelBitmap**	**labelFont**	**labelImage**
labelMargin	**labelPos**	**labelText**	**labelVariable**
state			

See the "labeledwidget" class manual entry for details on the inherited options.

WIDGET-SPECIFIC OPTIONS

Name: **childSitePos**
Class: **Position**
Command-Line Switch: **-childsitepos**

Specifies the position of the child site in the time field: **n**, **s**, **e**, or **w**. The default is e.

Name: **command**
Class: **Command**
Command-Line Switch: **-command**

Specifies a Tcl command to be executed upon detection of a Return key press event.

Name: **state**
Class: **State**
Command-Line Switch: **-state**

Specifies one of two states for the timefield: **normal** or **disabled**. If the timefield is disabled then input is not accepted. The default is normal.

Name: **textBackground**
Class: **Background**
Command-Line Switch: **-textbackground**

Background color for inside textual portion of the entry field. The value may be given in any of the forms acceptable to **Tk_GetColor**.

Name: **textFont**
Class: **Font**
Command-Line Switch: **-textfont**

Name of font to use for display of text in timefield. The value may be given in any of the forms acceptable to **Tk_GetFont**.

DESCRIPTION

The **timefield** command creates an enhanced text entry widget for the purpose of time entry with various degrees of built-in intelligence.

METHODS

The **timefield** command creates a new Tcl command whose name is *pathName*. This command may be used to invoke various operations on the widget. It has the following general form:

pathName option ?*arg arg* ...?

Option and the *arg*s determine the exact behavior of the command. The following commands are possible for timefield widgets:

WIDGET-SPECIFIC METHODS

pathName **cget** *option*

Returns the current value of the configuration option given by *option*. *Option* may have any of the values accepted by the **timefield** command.

pathName **configure** ?*option*? ?*value option value* ...?

Query or modify the configuration options of the widget. If no *option* is specified, returns a list describing all of the available options for *pathName* (see **Tk_ConfigureInfo** for information on the format of this list). If *option* is specified with no *value*, then the command returns a list describing the one named option (this list will be identical to the corresponding sublist of the value returned if no *option* is specified). If one or more *option−value* pairs are specified, then the command modifies the given widget option(s) to have the given value(s); in this case the command returns an empty string. *Option* may have any of the values accepted by the **timefield** command.

pathName **get** ?**format**?

Returns the current contents of the timefield in a format of string or as an integer clock value using the **-string** and **-clicks** format options respectively. The default is by string. Reference the clock command for more information on obtaining times and their formats.

pathName **isvalid**

Returns a boolean indication of the validity of the currently displayed time value. For example, 12:59:59 is valid whereas 25:59:59 is invalid.

pathName **show** *time*

Changes the currently displayed time to be that of the time argument. The time may be specified either as a string, an integer clock value or the keyword "now" (the default). Reference the clock command for more information on obtaining times and their formats.

COMPONENTS

Name: **time**
Class: **Entry**

The time component provides the entry field for time input and display. See the "entry" widget manual entry for details on the time component item.

EXAMPLE

```
proc returnCmd {} {
  puts [.tf get]
}

timefield .tf -command returnCmd
```

```
pack .tf -fill x -expand yes -padx 10 -pady 10
```

AUTHOR

John A. Tucker

Mark L. Ulferts

KEYWORDS

timefield, widget

NAME

toolbar − Create and manipulate a tool bar

SYNOPSIS

toolbar *pathName* ?*options*?

STANDARD OPTIONS

activeBackground	**font**	**insertForeground**	**selectForeground**
activeForeground	**foreground**	**orient**	**state**
background	**highlightBackground**	**relief**	**troughColor**
borderWidth	**highlightColor**	**selectBackground**	**cursor**
highlightThickness	**selectBorderWidth**	**disabledForeground**	**insertBackground**
selectColor			

See the "options" manual entry for details on the standard options. For widgets added to the toolbar, these options will be propogated if the widget supports the option. For example, all widgets that support a font option will be changed if the the toolbar's font option is configured.

WIDGET-SPECIFIC OPTIONS

Name: **balloonBackground**
Class: **BalloonBackground**
Command-Line Switch: **-ballooonbackground**

Specifies the background color of the balloon help displayed at the bottom center of a widget on the toolbar that has a non empty string for its balloonStr option. The default color is yellow.

Name: **balloonDelay1**
Class: **BalloonDelay1**
Command-Line Switch: **-balloondelay1**

Specifies the length of time (in milliseconds) to wait before initially posting a balloon help hint window. This delay is in effect whenever 1) the mouse leaves the toolbar, or 2) a toolbar item is selected with the mouse button.

Name: **balloonDelay2**
Class: **BalloonDelay2**
Command-Line Switch: **-balloondelay2**

Specifies the length of time (in milliseconds) to wait before continuing to post balloon help hint windows. This delay is in effect after the first time a balloon hint window is activated. It remains in effect until 1) the mouse leaves the toolbar, or 2) a toolbar item is selected with the mouse button.

Name: **balloonFont**
Class: **BalloonFont**
Command-Line Switch: **-balloonfont**

Specifies the font of the balloon help text displayed at the bottom center of a widget on the toolbar that has a non empty string for its balloonStr option. The default font is 6x10.

Name: **balloonForeground**
Class: **BalloonForeground**
Command-Line Switch: **-ballooonforeground**

Specifies the foreground color of the balloon help displayed at the bottom center of a widget on the toolbar that has a non empty string for its balloonStr option. The default color is black.

Name: **helpVariable**
Class: **HelpVariable**
Command-Line Switch: **-helpvariable**

Specifies the global variable to update whenever the mouse is in motion over a toolbar widget.

This global variable is updated with the current value of the active widget's helpStr. Other widgets can "watch" this variable with the trace command, or as is the case with entry or label widgets, they can set their textVariable to the same global variable. This allows for a simple implementation of a help status bar. Whenever the mouse leaves a menu entry, the helpVariable is set to the empty string {}.

Name: **orient**
Class: **Orient**
Command-Line Switch: **-orient**

Specifies the orientation of the toolbar. Must be either horizontal or vertical.

DESCRIPTION

The **toolbar** command creates a new window (given by the pathName argument) and makes it into a **toolbar** widget. Additional options, described above may be specified on the command line or in the option database to configure aspects of the toolbar such as its colors, font, and orientation. The **toolbar** command returns its pathName argument. At the time this command is invoked, there must not exist a window named pathName, but pathName's parent must exist.

A **toolbar** is a widget that displays a collection of widgets arranged either in a row or a column (depending on the value of the -orient option). This collection of widgets is usually for user convenience to give access to a set of commands or settings. Any widget may be placed on a toolbar. However, command or value-oriented widgets (such as button, radiobutton, etc.) are usually the most useful kind of widgets to appear on a toolbar.

In addition, the toolbar adds two new options to all widgets that are added to it. These are the **helpStr** and **balloonStr** options. See the discussion for the widget command add below.

WIDGET-SPECIFIC METHODS

The toolbar command creates a new Tcl command whose name is pathName. This command may be used to invoke various operations on the widget. It has the following general form:

pathName option ?arg arg ...?

Option and args determine the exact behavior of the command.

Many of the widget commands for a toolbar take as one argument an indicator of which widget item of the toolbar to operate on. The indicator is called an **index** and may be specified in any of the following forms:

number Specifies the widget numerically, where 0 corresponds to the first widget in the notebook, 1 to the second, and so on. (For horizontal, 0 is the leftmost; for vertical, 0 is the topmost).

end Specifes the last widget in the toolbar's index. If the toolbar is empty this will return -1.

last Same as end.

pattern If the index doesn't satisfy any of the above forms, then this form is used. Pattern is pattern-matched against the widgetName of each widget in the toolbar, in order from the first to the last widget, until a matching entry is found. An exact match must occur.

The following commands are possible for toolbar widgets:

pathName **add** *widgetCommand widgetName ?option value?*

Adds a widget with the command widgetCommand whose name is widgetName to the toolbar. If widgetCommand is radiobutton or checkbutton, its packing is slightly padded to match the geometry of button widgets. In addition, the indicatorOn option is false by default and the selectColor is that of the toolbar background by default. This allows Radiobutton and Checkbutton widgets to be

added as icons by simply setting their bitmap or image options. If additional arguments are present, they are the set of available options that the widget type of *widgetCommand* supports. In addition they may also be one of the following options:

-helpstr *value*
Specifes the help string to associate with the widget. When the mouse moves over the widget, the variable denoted by **helpVariable** is set to **helpStr**. Another widget can bind to the helpVariable and thus track status help.

-balloonstr *value*
Specifes the string to display in a balloon window for this widget. A balloon window is a small popup window centered at the bottom of the widget. Usually the **balloonStr** value is the name of the item on the toolbar. It is sometimes known as a hint window.

When the mouse moves into an item on the toolbar, a timer is set based on the value of **balloonDelay1**. If the mouse stays inside the item for **balloonDelay1**, the balloon window will pop up displaying the **balloonStr** value. Once the balloon window is posted, a new timer based on **balloonDelay2** is set. This is typically a shorter timer. If the mouse is moved to another item, the window is unposted and a new window will be posted over the item if the shorter delay time is satisfied.

While the balloon window is posted, it can also be unposted if the item is selected. In this case the timer is reset to **balloonDelay1**. Whenever the mouse leaves the toolbar, the timer is also reset to **balloonDelay1**.

This window posting/unposting model is the same model used in the Windows95 environment.

pathName **cget** *option*
Returns the current value of the configuration option given by *option*.

pathName **configure** ?*option value*?
Query or modify the configuration options of the widget. If no *option* is specified, returns a list describing all of the available options for pathName (see Tk_ConfigureInfo for information on the format of this list). If *option* is specified with no value, then the command returns a list describing the one named option (this list will be identical to the corresponding sublist of the value returned if no option is specified). If one or more option-value pairs are specified, then the command modifies the given widget option(s) to have the given value(s); in this case the command returns an empty string.

pathName **delete** *index* ?*index2*?
This command deletes all items between *index* and *index2* inclusive. If *index2* is omitted then it defaults to *index*. Returns an empty string.

pathName **index** *index*
Returns the widget's numerical index for the entry corresponding to *index*. If *index* is not found, -1 is returned.

pathName **insert** *beforeIndex widgetCommand widgetName ?option value?*

Insert a new item named *widgetName* with the
command *widgetCommand* before the item specified by *beforeIndex*. If *widgetCommand* is **radiobutton** or **checkbutton**, its packing is slightly padded to match the geometry of button widgets. In addition, the **indicatorOn** option is **false** by default and the **selectColor** is that of the toolbar background by default. This allows **Radiobutton** and **Checkbutton** widgets to be added as icons by simply setting their **bitmap** or **image** options. The set of available options is the same as specified in the **ad** command.

pathName **itemcget** *index option*

Returns the current value of the configuration option given by *option* for index. The item type of *index* determines the valid available options.

pathName **itemconfigure** *index ?option value?*

Query or modify the configuration options of the widget of the toolbar specified by *index*. If no option is specified, returns a list describing all of the available options for *index* (see **Tk_ConfigureInfo** for information on the format of this list). If *option* is specified with no value, then the command returns a list describing the one named option (this list will be identical to the corresponding sublist of the value returned if no option is specified). If one or more option-value pairs are specified, then the command modifies the given widget option(s) to have the given value(s); in this case the command returns an empty string. The item type of *index* determines the valid available options. The set of available options is the same as specified in the **ad** command.

EXAMPLE

```
toolbar .tb -helpvariable statusVar

.tb add button item1 \
  -helpstr "Save It" -bitmap @./icons/Tool_32_box.xbm \
  -balloonstr "Save" -command {puts 1}
.tb add button item2 \
  -helpstr "Save It" -bitmap @./icons/Tool_32_brush.xbm \
  -balloonstr "Save" -command {puts 1}
.tb add button item3 \
  -helpstr "Save It" -bitmap @./icons/Tool_32_cut.xbm \
  -balloonstr "Save" -command {puts 1}
.tb add button item4 \
  -helpstr "Save It" -bitmap @./icons/Tool_32_draw.xbm \
  -balloonstr "Save" -command {puts 1}
.tb add button item5 \
  -bitmap @./icons/Tool_32_erase.xbm -helpstr "Play It" \
  -command {puts 2}
.tb add frame filler \
  -borderwidth 1 -width 10 -height 10
.tb add radiobutton item6 \
  -bitmap @./icons/Tool_32_oval.xbm -command {puts 4} \
  -variable result -value OPEN -helpstr "Radio Button # 1" \
  -balloonstr "Radio"
.tb add radiobutton item7 \
  -bitmap @./icons/Tool_32_line.xbm -command {puts 5} \
  -variable result -value CLOSED
.tb add checkbutton item8 \
  -bitmap @./icons/Tool_32_text.xbm -command {puts 6} \
  -variable checkit -onvalue yes -offvalue no
.tb add checkbutton check2 \
  -bitmap @./icons/Tool_32_points.xbm -command {puts 7} \
   -variable checkit2 -onvalue yes -offvalue no

pack .tb -side top -anchor nw
```

AUTHOR

Bill Scott

KEYWORDS

toolbar, button, radiobutton, checkbutton, iwidgets, widget

NAME

watch − Create and manipulate time with a watch widgets

SYNOPSIS

watch *pathName* ?*options*?

INHERITANCE

itk::Widget <- Watch

STANDARD OPTIONS

background **cursor** **foreground** **relief**

See the "options" manual entry for details on the standard options.

ASSOCIATED OPTIONS

See the "Canvas" manual entry for details on the above associated options.

WIDGET-SPECIFIC OPTIONS

Name: **clockColor**
Class: **ColorfR**
Command-Line Switch: **-clockcolor**

Fill color for the main oval encapsulating the watch, in any of the forms acceptable to **Tk_GetColor**. The default is "White".

Name: **clockStipple**
Class: **BitmapfR**
Command-Line Switch: **-clockstipple**

Bitmap for the main oval encapsulating the watch, in any of the forms acceptable to **Tk_GetBitmap**. The default is "".

Name: **height**
Class: **Height**
Command-Line Switch: **-height**

Specifies the height of the watch widget in any of the forms acceptable to **Tk_GetPixels**. The default height is 175 pixels.

Name: **hourColor**
Class: **ColorfR**
Command-Line Switch: **-hourcolor**

Fill color for the hour hand, in any of the forms acceptable to **Tk_GetColor**. The default is "Red".

Name: **hourRadius**
Class: **Radius**
Command-Line Switch: **-hourradius**

Specifies the radius of the hour hand as a percentage of the radius from the center to the out perimeter of the clock. The value must be a fraction <= 1. The default is ".5".

Name: **minuteColor**
Class: **ColorfR**
Command-Line Switch: **-minutecolor**

Fill color for the minute hand, in any of the forms acceptable to **Tk_GetColor**. The default is "Yellow".

Name: **minuteRadius**
Class: **Radius**
Command-Line Switch: **-minuteradius**

Specifies the radius of the minute hand as a percentage of the radius from the center to the out perimeter of the clock. The value must be a fraction <= 1. The default is ".8".

Name: **pivotColor**
Class: **ColorfR**
Command-Line Switch: **-pivotcolor**

Fill color for the circle in which the watch hands rotate in any of the forms acceptable to **Tk_GetColor**. The default is "White".

Name: **pivotRadius**
Class: **Radius**
Command-Line Switch: **-pivotradius**

Specifies the radius of the circle in which the watch hands rotate as a percentage of the radius. The value must be a fraction <= 1. The default is ".1".

Name: **secondColor**
Class: **ColorfR**
Command-Line Switch: **-secondcolor**

Fill color for the second hand, in any of the forms acceptable to **Tk_GetColor**. The default is "Black".

Name: **secondRadius**
Class: **Radius**
Command-Line Switch: **-secondradius**

Specifies the radius of the second hand as a percentage of the radius from the center to the out perimeter of the clock. The value must be a fraction <= 1. The default is ".9".

Name: **showAmPm**
Class: **ShosAmPm**
Command-Line Switch: **-showampm**

Specifies whether the AM/PM radiobuttons should be displayed, in any of the forms acceptable to **Tcl_GetBoolean**. The default is yes.

Name: **state**
Class: **State**
Command-Line Switch: **-state**

Specifies the editable state for the hands on the watch. In a normal state, the user can select and move the hands via mouse button 1. The valid values are **normal**, and **disabled**. The defult is normal.

Name: **tickColor**
Class: **ColorfR**
Command-Line Switch: **-tickcolor**

Fill color for the 60 ticks around the perimeter of the watch, in any of the forms acceptable to **Tk_GetColor**. The default is "Black".

Name: **width**
Class: **Width**
Command-Line Switch: **-width**

Specifies the width of the watch widget in any of the forms acceptable to **Tk_GetPixels**. The

default height is 155 pixels.

DESCRIPTION

The **watch** command creates a watch with hour, minute, and second hands modifying the time value.

METHODS

The **watch** command creates a new Tcl command whose name is *pathName*. This command may be used to invoke various operations on the widget. It has the following general form:

pathName option ?*arg arg ...*?

Option and the *arg*s determine the exact behavior of the command. The following commands are possible for watch widgets:

WIDGET-SPECIFIC METHODS

pathName **cget** *option*

Returns the current value of the configuration option given by *option*. *Option* may have any of the values accepted by the **watch** command.

pathName **configure** ?*option*? ?*value option value ...*?

Query or modify the configuration options of the widget. If no *option* is specified, returns a list describing all of the available options for *pathName* (see **Tk_ConfigureInfo** for information on the format of this list). If *option* is specified with no *value*, then the command returns a list describing the one named option (this list will be identical to the corresponding sublist of the value returned if no *option* is specified). If one or more *option−value* pairs are specified, then the command modifies the given widget option(s) to have the given value(s); in this case the command returns an empty string. *Option* may have any of the values accepted by the **watch** command.

pathName **get** ?**format**?

Returns the current time of the watch in a format of string or as an integer clock value using the **-string** and **-clicks** format options respectively. The default is by string. Reference the clock command for more information on obtaining time and its formats.

pathName **show** *time*

Changes the currently displayed time to be that of the time argument. The time may be specified either as a string, an integer clock value or the keyword "now". Reference the clock command for more information on obtaining time and its format.

pathName **watch** ?**args**?

Evaluates the specifieds **args** against the canvas component.

COMPONENTS

Name: **canvas**
Class: **Canvas**

The canvas component is the where the clock is drawn. See the Canvas widget manual entry for details.

Name: **frame**
Class: **Frame**

The frame component is the where the "AM" and "PM" radiobuttons are displayed. See the Frame widget manual entry for details.

Name: **am**
Class: **Radiobutton**

The am component indicates whether on not the time is relative to "AM". See the Radiobutton widget manual entry for details.

Name: **pm**
Class: **Radiobutton**

The pm component indicates whether on not the time is relative to "PM". See the Radiobutton widget manual entry for details.

EXAMPLE

```
watch .w -state disabled -showampm no -width 155 -height 155
pack .w -padx 10 -pady 10 -fill both -expand yes

while {1} {
  after 1000
  .w show
  update
}
```

AUTHOR

John Tucker

Mark L. Ulferts

KEYWORDS

watch, hand, ticks, pivot, widget

NAME

Tcl_AddObjErrorInfo, Tcl_AddErrorInfo, Tcl_SetErrorCode, Tcl_PosixError − record information about errors

SYNOPSIS

#include <tcl.h>

Tcl_AddObjErrorInfo(*interp, message, length*)

Tcl_AddErrorInfo(*interp, message*)

Tcl_SetObjErrorCode(*interp, errorObjPtr*)

Tcl_SetErrorCode(*interp, element, element, ...* **(char** ∗**) NULL**)

char ∗
Tcl_PosixError(*interp*)

ARGUMENTS

Tcl_Interp	∗*interp*	(in)	Interpreter in which to record information.
char	∗*message*	(in)	For **Tcl_AddObjErrorInfo**, this points to the first byte of an array of bytes containing a string to record in the **errorInfo** variable. This byte array may contain embedded null bytes unless *length* is negative. For **Tcl_AddErrorInfo**, this is a conventional C string to record in the **errorInfo** variable.
int	*length*	(in)	The number of bytes to copy from *message* when setting the **errorInfo** variable. If negative, all bytes up to the first null byte are used.
Tcl_Obj	∗*errorObjPtr*	(in)	This variable **errorCode** will be set to this value.
char	∗*element*	(in)	String to record as one element of **errorCode** variable. Last *element* argument must be NULL.

DESCRIPTION

These procedures are used to manipulate two Tcl global variables that hold information about errors. The variable **errorInfo** holds a stack trace of the operations that were in progress when an error occurred, and is intended to be human-readable. The variable **errorCode** holds a list of items that are intended to be machine-readable. The first item in **errorCode** identifies the class of error that occurred (e.g. POSIX means an error occurred in a POSIX system call) and additional elements in **errorCode** hold additional pieces of information that depend on the class. See the Tcl overview manual entry for details on the various formats for **errorCode**.

The **errorInfo** variable is gradually built up as an error unwinds through the nested operations. Each time an error code is returned to **Tcl_EvalObj** (or **Tcl_Eval**, which calls **Tcl_EvalObj**) it calls the procedure **Tcl_AddObjErrorInfo** to add additional text to **errorInfo** describing the command that was being executed when the error occurred. By the time the error has been passed all the way back to the application, it will contain a complete trace of the activity in progress when the error occurred.

It is sometimes useful to add additional information to **errorInfo** beyond what can be supplied automatically by **Tcl_EvalObj**. **Tcl_AddObjErrorInfo** may be used for this purpose: its *message* and *length* arguments describe an additional string to be appended to **errorInfo**. For example, the **source** command calls **Tcl_AddObjErrorInfo** to record the name of the file being processed and the line number on which the error occurred; for Tcl procedures, the procedure name and line number within the procedure are recorded,

and so on. The best time to call **Tcl_AddObjErrorInfo** is just after **Tcl_EvalObj** has returned **TCL_ERROR**. In calling **Tcl_AddObjErrorInfo**, you may find it useful to use the **errorLine** field of the interpreter (see the **Tcl_Interp** manual entry for details).

Tcl_AddErrorInfo resembles **Tcl_AddObjErrorInfo** but differs in initializing **errorInfo** from the string value of the interpreter's result if the error is just starting to be logged. It does not use the result as a Tcl object so any embedded null characters in the result will cause information to be lost. It also takes a conventional C string in *message* instead of **Tcl_AddObjErrorInfo**'s counted string.

The procedure **Tcl_SetObjErrorCode** is used to set the **errorCode** variable. *errorObjPtr* contains a list object built up by the caller. **errorCode** is set to this value. **Tcl_SetObjErrorCode** is typically invoked just before returning an error in an object command. If an error is returned without calling **Tcl_SetObjErrorCode** or **Tcl_SetErrorCode** the Tcl interpreter automatically sets **errorCode** to **NONE**.

The procedure **Tcl_SetErrorCode** is also used to set the **errorCode** variable. However, it takes one or more strings to record instead of an object. Otherwise, it is similar to **Tcl_SetObjErrorCode** in behavior.

Tcl_PosixError sets the **errorCode** variable after an error in a POSIX kernel call. It reads the value of the **errno** C variable and calls **Tcl_SetErrorCode** to set **errorCode** in the **POSIX** format. The caller must previously have called **Tcl_SetErrno** to set **errno**; this is necessary on some platforms (e.g. Windows) where Tcl is linked into an application as a shared library, or when the error occurs in a dynamically loaded extension. See the manual entry for **Tcl_SetErrno** for more information.

Tcl_PosixError returns a human-readable diagnostic message for the error (this is the same value that will appear as the third element in **errorCode**). It may be convenient to include this string as part of the error message returned to the application in the interpreter's result.

It is important to call the procedures described here rather than setting **errorInfo** or **errorCode** directly with **Tcl_ObjSetVar2**. The reason for this is that the Tcl interpreter keeps information about whether these procedures have been called. For example, the first time **Tcl_AddObjErrorInfo** is called for an error, it clears the existing value of **errorInfo** and adds the error message in the interpreter's result to the variable before appending *message*; in subsequent calls, it just appends the new *message*. When **Tcl_SetErrorCode** is called, it sets a flag indicating that **errorCode** has been set; this allows the Tcl interpreter to set **errorCode** to **NONE** if it receives an error return when **Tcl_SetErrorCode** hasn't been called.

If the procedure **Tcl_ResetResult** is called, it clears all of the state associated with **errorInfo** and **errorCode** (but it doesn't actually modify the variables). If an error had occurred, this will clear the error state to make it appear as if no error had occurred after all.

SEE ALSO

Tcl_DecrRefCount, Tcl_IncrRefCount, Tcl_Interp, Tcl_ResetResult, Tcl_SetErrno

KEYWORDS

error, object, object result, stack, trace, variable

NAME

Tcl_Alloc, Tcl_Free, Tcl_Realloc − allocate or free heap memory

SYNOPSIS

#include <tcl.h>

char *
Tcl_Alloc(*size*)

Tcl_Free(*ptr*)

char *
Tcl_Realloc(*ptr*, *size*)

ARGUMENTS

int	*size*	(in)	Size in bytes of the memory block to allocate.
char	**ptr*	(in)	Pointer to memory block to free or realloc.

DESCRIPTION

These procedures provide a platform and compiler independent interface for memory allocation. Programs that need to transfer ownership of memory blocks between Tcl and other modules should use these routines rather than the native **malloc()** and **free()** routines provided by the C run-time library.

Tcl_Alloc returns a pointer to a block of at least *size* bytes suitably aligned for any use.

Tcl_Free makes the space referred to by *ptr* available for further allocation.

Tcl_Realloc changes the size of the block pointed to by *ptr* to *size* bytes and returns a pointer to the new block. The contents will be unchanged up to the lesser of the new and old sizes. The returned location may be different from *ptr*.

KEYWORDS

alloc, allocation, free, malloc, memory, realloc

NAME

Tcl_AllowExceptions − allow all exceptions in next script evaluation

SYNOPSIS

#include <tcl.h>

Tcl_AllowExceptions(*interp*)

ARGUMENTS

Tcl_Interp	∗*interp*	(in)	Interpreter in which script will be evaluated.

DESCRIPTION

If a script is evaluated at top-level (i.e. no other scripts are pending evaluation when the script is invoked), and if the script terminates with a completion code other than TCL_OK, TCL_CONTINUE or TCL_RETURN, then Tcl normally converts this into a TCL_ERROR return with an appropriate message.

However, if **Tcl_AllowExceptions** is invoked immediately before calling a procedure such as **Tcl_Eval**, then arbitrary completion codes are permitted from the script, and they are returned without modification. This is useful in cases where the caller can deal with exceptions such as TCL_BREAK or TCL_CONTINUE in a meaningful way.

KEYWORDS

continue, break, exception, interpreter

NAME

Tcl_AppInit − perform application-specific initialization

SYNOPSIS

#include <tcl.h>

int
Tcl_AppInit(*interp*)

ARGUMENTS

Tcl_Interp	∗*interp*	(in)	Interpreter for the application.

DESCRIPTION

Tcl_AppInit is a ‘‘hook’’ procedure that is invoked by the main programs for Tcl applications such as **tclsh** and **wish**. Its purpose is to allow new Tcl applications to be created without modifying the main programs provided as part of Tcl and Tk. To create a new application you write a new version of **Tcl_AppInit** to replace the default version provided by Tcl, then link your new **Tcl_AppInit** with the Tcl library.

Tcl_AppInit is invoked after by **Tcl_Main** and **Tk_Main** after their own initialization and before entering the main loop to process commands. Here are some examples of things that **Tcl_AppInit** might do:

[1] Call initialization procedures for various packages used by the application. Each initialization procedure adds new commands to *interp* for its package and performs other package-specific initialization.

[2] Process command-line arguments, which can be accessed from the Tcl variables **argv** and **argv0** in *interp*.

[3] Invoke a startup script to initialize the application.

Tcl_AppInit returns TCL_OK or TCL_ERROR. If it returns TCL_ERROR then it must leave an error message in *interp->result*; otherwise the result is ignored.

In addition to **Tcl_AppInit**, your application should also contain a procedure **main** that calls **Tcl_Main** as follows:

```
Tcl_Main(argc, argv, Tcl_AppInit);
```

The third argument to **Tcl_Main** gives the address of the application-specific initialization procedure to invoke. This means that you don’t have to use the name **Tcl_AppInit** for the procedure, but in practice the name is nearly always **Tcl_AppInit** (in versions before Tcl 7.4 the name **Tcl_AppInit** was implicit; there was no way to specify the procedure explicitly). The best way to get started is to make a copy of the file **tclAppInit.c** from the Tcl library or source directory. It already contains a **main** procedure and a template for **Tcl_AppInit** that you can modify for your application.

KEYWORDS

application, argument, command, initialization, interpreter

NAME

Tcl_GetAssocData, Tcl_SetAssocData, Tcl_DeleteAssocData − manage associations of string keys and user specified data with Tcl interpreters.

SYNOPSIS

#include <tcl.h>

ClientData
Tcl_GetAssocData(*interp, key, delProcPtr*)

Tcl_SetAssocData(*interp, key, delProc, clientData*)

Tcl_DeleteAssocData(*interp, key*)

ARGUMENTS

Tcl_Interp	∗*interp*	(in)	Interpreter in which to execute the specified command.
char	∗*key*	(in)	Key for association with which to store data or from which to delete or retrieve data. Typically the module prefix for a package.
Tcl_InterpDeleteProc	∗*delProc*	(in)	Procedure to call when *interp* is deleted.
Tcl_InterpDeleteProc	∗∗*delProcPtr*	(in)	Pointer to location in which to store address of current deletion procedure for association. Ignored if NULL.
ClientData	*clientData*	(in)	Arbitrary one-word value associated with the given key in this interpreter. This data is owned by the caller.

DESCRIPTION

These procedures allow extensions to associate their own data with a Tcl interpreter. An association consists of a string key, typically the name of the extension, and a one-word value, which is typically a pointer to a data structure holding data specific to the extension. Tcl makes no interpretation of either the key or the value for an association.

Storage management is facilitated by storing with each association a procedure to call when the interpreter is deleted. This procedure can dispose of the storage occupied by the client's data in any way it sees fit.

Tcl_SetAssocData creates an association between a string key and a user specified datum in the given interpreter. If there is already an association with the given *key*, **Tcl_SetAssocData** overwrites it with the new information. It is up to callers to organize their use of names to avoid conflicts, for example, by using package names as the keys. If the *deleteProc* argument is non-NULL it specifies the address of a procedure to invoke if the interpreter is deleted before the association is deleted. *DeleteProc* should have arguments and result that match the type **Tcl_InterpDeleteProc**:

```
typedef void Tcl_InterpDeleteProc(
    ClientData clientData,
    Tcl_Interp *interp);
```

When *deleteProc* is invoked the *clientData* and *interp* arguments will be the same as the corresponding arguments passed to **Tcl_SetAssocData**. The deletion procedure will *not* be invoked if the association is deleted before the interpreter is deleted.

Tcl_GetAssocData returns the datum stored in the association with the specified key in the given interpreter, and if the *delProcPtr* field is non-**NULL**, the address indicated by it gets the address of the delete procedure stored with this association. If no association with the specified key exists in the given interpreter **Tcl_GetAssocData** returns **NULL**.

Tcl_DeleteAssocData deletes an association with a specified key in the given interpreter. It does not call the deletion procedure.

KEYWORDS

association, data, deletion procedure, interpreter, key

NAME

Tcl_AsyncCreate, Tcl_AsyncMark, Tcl_AsyncInvoke, Tcl_AsyncDelete − handle asynchronous events

SYNOPSIS

```
#include <tcl.h>

Tcl_AsyncHandler
Tcl_AsyncCreate(proc, clientData)

Tcl_AsyncMark(async)

int
Tcl_AsyncInvoke(interp, code)

Tcl_AsyncDelete(async)

int
Tcl_AsyncReady()
```

ARGUMENTS

Tcl_AsyncProc	*∗proc*	(in)	Procedure to invoke to handle an asynchronous event.
ClientData	*clientData*	(in)	One-word value to pass to *proc*.
Tcl_AsyncHandler	*async*	(in)	Token for asynchronous event handler.
Tcl_Interp	*∗interp*	(in)	Tcl interpreter in which command was being evaluated when handler was invoked, or NULL if handler was invoked when there was no interpreter active.
int	*code*	(in)	Completion code from command that just completed in *interp*, or 0 if *interp* is NULL.

DESCRIPTION

These procedures provide a safe mechanism for dealing with asynchronous events such as signals. If an event such as a signal occurs while a Tcl script is being evaluated then it isn't safe to take any substantive action to process the event. For example, it isn't safe to evaluate a Tcl script since the interpreter may already be in the middle of evaluating a script; it may not even be safe to allocate memory, since a memory allocation could have been in progress when the event occurred. The only safe approach is to set a flag indicating that the event occurred, then handle the event later when the world has returned to a clean state, such as after the current Tcl command completes.

Tcl_AsyncCreate creates an asynchronous handler and returns a token for it. The asynchronous handler must be created before any occurrences of the asynchronous event that it is intended to handle (it is not safe to create a handler at the time of an event). When an asynchronous event occurs the code that detects the event (such as a signal handler) should call **Tcl_AsyncMark** with the token for the handler. **Tcl_AsyncMark** will mark the handler as ready to execute, but it will not invoke the handler immediately. Tcl will call the *proc* associated with the handler later, when the world is in a safe state, and *proc* can then carry out the actions associated with the asynchronous event. *Proc* should have arguments and result that match the type **Tcl_AsyncProc**:

```
typedef int Tcl_AsyncProc(
    ClientData clientData,
    Tcl_Interp *interp,
    int code);
```

The *clientData* will be the same as the *clientData* argument passed to **Tcl_AsyncCreate** when the handler was created. If *proc* is invoked just after a command has completed execution in an interpreter, then *interp* will identify the interpreter in which the command was evaluated and *code* will be the completion code returned by that command. The command's result will be present in *interp->result*. When *proc* returns, whatever it leaves in *interp->result* will be returned as the result of the command and the integer value returned by *proc* will be used as the new completion code for the command.

It is also possible for *proc* to be invoked when no interpreter is active. This can happen, for example, if an asynchronous event occurs while the application is waiting for interactive input or an X event. In this case *interp* will be NULL and *code* will be 0, and the return value from *proc* will be ignored.

The procedure **Tcl_AsyncInvoke** is called to invoke all of the handlers that are ready. The procedure **Tcl_AsyncReady** will return non-zero whenever any asynchronous handlers are ready; it can be checked to avoid calls to **Tcl_AsyncInvoke** when there are no ready handlers. Tcl calls **Tcl_AsyncReady** after each command is evaluated and calls **Tcl_AsyncInvoke** if needed. Applications may also call **Tcl_AsyncInvoke** at interesting times for that application. For example, Tcl's event handler calls **Tcl_AsyncReady** after each event and calls **Tcl_AsyncInvoke** if needed. The *interp* and *code* arguments to **Tcl_AsyncInvoke** have the same meaning as for *proc*: they identify the active interpreter, if any, and the completion code from the command that just completed.

Tcl_AsyncDelete removes an asynchronous handler so that its *proc* will never be invoked again. A handler can be deleted even when ready, and it will still not be invoked.

If multiple handlers become active at the same time, the handlers are invoked in the order they were created (oldest handler first). The *code* and *interp->result* for later handlers reflect the values returned by earlier handlers, so that the most recently created handler has last say about the interpreter's result and completion code. If new handlers become ready while handlers are executing, **Tcl_AsyncInvoke** will invoke them all; at each point it invokes the highest-priority (oldest) ready handler, repeating this over and over until there are no longer any ready handlers.

WARNING

It is almost always a bad idea for an asynchronous event handler to modify *interp->result* or return a code different from its *code* argument. This sort of behavior can disrupt the execution of scripts in subtle ways and result in bugs that are extremely difficult to track down. If an asynchronous event handler needs to evaluate Tcl scripts then it should first save *interp->result* plus the values of the variables **errorInfo** and **errorCode** (this can be done, for example, by storing them in dynamic strings). When the asynchronous handler is finished it should restore *interp->result*, **errorInfo**, and **errorCode**, and return the *code* argument.

KEYWORDS

asynchronous event, handler, signal

NAME

Tcl_BackgroundError − report Tcl error that occurred in background processing

SYNOPSIS

#include <tcl.h>

Tcl_BackgroundError(*interp*)

ARGUMENTS

Tcl_Interp	∗*interp*	(in)	Interpreter in which the error occurred.

DESCRIPTION

This procedure is typically invoked when a Tcl error occurs during ‘‘background processing’’ such as executing an event handler. When such an error occurs, the error condition is reported to Tcl or to a widget or some other C code, and there is not usually any obvious way for that code to report the error to the user. In these cases the code calls **Tcl_BackgroundError** with an *interp* argument identifying the interpreter in which the error occurred. At the time **Tcl_BackgroundError** is invoked, *interp->result* is expected to contain an error message. **Tcl_BackgroundError** will invoke the **bgerror** Tcl command to report the error in an application-specific fashion. If no **bgerror** command exists, or if it returns with an error condition, then **Tcl_BackgroundError** reports the error itself by printing a message on the standard error file.

Tcl_BackgroundError does not invoke **bgerror** immediately because this could potentially interfere with scripts that are in process at the time the error occurred. Instead, it invokes **bgerror** later as an idle callback. **Tcl_BackgroundError** saves the values of the **errorInfo** and **errorCode** variables and restores these values just before invoking **bgerror**.

It is possible for many background errors to accumulate before **bgerror** is invoked. When this happens, each of the errors is processed in order. However, if **bgerror** returns a break exception, then all remaining error reports for the interpreter are skipped.

KEYWORDS

background, bgerror, error

NAME

Tcl_Backslash − parse a backslash sequence

SYNOPSIS

#include <tcl.h>

char
Tcl_Backslash(*src, countPtr*)

ARGUMENTS

char	∗*src*	(in)	Pointer to a string starting with a backslash.
int	∗*countPtr*	(out)	If *countPtr* isn't NULL, ∗*countPtr* gets filled in with number of characters in the backslash sequence, including the backslash character.

DESCRIPTION

This is a utility procedure used by several of the Tcl commands. It parses a backslash sequence and returns the single character corresponding to the sequence. **Tcl_Backslash** modifies ∗*countPtr* to contain the number of characters in the backslash sequence.

See the Tcl manual entry for information on the valid backslash sequences. All of the sequences described in the Tcl manual entry are supported by **Tcl_Backslash**.

KEYWORDS

backslash, parse

NAME

Tcl_NewBooleanObj, Tcl_SetBooleanObj, Tcl_GetBooleanFromObj − manipulate Tcl objects as boolean values

SYNOPSIS

```
#include <tcl.h>

Tcl_Obj *
Tcl_NewBooleanObj(boolValue)

Tcl_SetBooleanObj(objPtr, boolValue)

int
Tcl_GetBooleanFromObj(interp, objPtr, boolPtr)
```

ARGUMENTS

int	*boolValue*	(in)	Integer value used to initialize or set a boolean object. If the integer is nonzero, the boolean object is set to 1; otherwise the boolean object is set to 0.
Tcl_Obj	∗*objPtr*	(in/out)	For **Tcl_SetBooleanObj**, this points to the object to be converted to boolean type. For **Tcl_GetBooleanFromObj**, this refers to the object from which to get a boolean value; if *objPtr* does not already point to a boolean object, an attempt will be made to convert it to one.
Tcl_Interp	∗*interp*	(in/out)	If an error occurs during conversion, an error message is left in the interpreter's result object unless *interp* is NULL.
int	∗*boolPtr*	(out)	Points to place where **Tcl_GetBooleanFromObj** stores the boolean value (0 or 1) obtained from *objPtr*.

DESCRIPTION

These procedures are used to create, modify, and read boolean Tcl objects from C code. **Tcl_NewBooleanObj** and **Tcl_SetBooleanObj** will create a new object of boolean type or modify an existing object to have boolean type. Both of these procedures set the object to have the boolean value (0 or 1) specified by *boolValue*; if *boolValue* is nonzero, the object is set to 1, otherwise to 0. **Tcl_NewBooleanObj** returns a pointer to a newly created object with reference count zero. Both procedures set the object's type to be boolean and assign the boolean value to the object's internal representation *longValue* member. **Tcl_SetBooleanObj** invalidates any old string representation and, if the object is not already a boolean object, frees any old internal representation.

Tcl_GetBooleanFromObj attempts to return a boolean value from the Tcl object *objPtr*. If the object is not already a boolean object, it will attempt to convert it to one. If an error occurs during conversion, it returns **TCL_ERROR** and leaves an error message in the interpreter's result object unless *interp* is NULL. Otherwise, **Tcl_GetBooleanFromObj** returns **TCL_OK** and stores the boolean value in the address given by *boolPtr*. If the object is not already a boolean object, the conversion will free any old internal representation.

SEE ALSO

Tcl_NewObj, Tcl_DecrRefCount, Tcl_IncrRefCount, Tcl_GetObjResult

KEYWORDS

boolean, boolean object, boolean type, internal representation, object, object type, string representation

NAME

Tcl_CallWhenDeleted, Tcl_DontCallWhenDeleted − Arrange for callback when interpreter is deleted

SYNOPSIS

#include <tcl.h>

Tcl_CallWhenDeleted(*interp*, *proc*, *clientData*)

Tcl_DontCallWhenDeleted(*interp*, *proc*, *clientData*)

ARGUMENTS

Tcl_Interp	∗*interp*	(in)	Interpreter with which to associated callback.
Tcl_InterpDeleteProc	∗*proc*	(in)	Procedure to call when *interp* is deleted.
ClientData	*clientData*	(in)	Arbitrary one-word value to pass to *proc*.

DESCRIPTION

Tcl_CallWhenDeleted arranges for *proc* to be called by **Tcl_DeleteInterp** if/when *interp* is deleted at some future time. *Proc* will be invoked just before the interpreter is deleted, but the interpreter will still be valid at the time of the call. *Proc* should have arguments and result that match the type **Tcl_InterpDeleteProc**:

```
typedef void Tcl_InterpDeleteProc(
    ClientData clientData,
    Tcl_Interp ∗interp);
```

The *clientData* and *interp* parameters are copies of the *clientData* and *interp* arguments given to **Tcl_CallWhenDeleted**. Typically, *clientData* points to an application-specific data structure that *proc* uses to perform cleanup when an interpreter is about to go away. *Proc* does not return a value.

Tcl_DontCallWhenDeleted cancels a previous call to **Tcl_CallWhenDeleted** with the same arguments, so that *proc* won't be called after all when *interp* is deleted. If there is no deletion callback that matches *interp*, *proc*, and *clientData* then the call to **Tcl_DontCallWhenDeleted** has no effect.

KEYWORDS

callback, delete, interpreter

NAME

Tcl_CommandComplete − Check for unmatched braces in a Tcl command

SYNOPSIS

#include <tcl.h>

int
Tcl_CommandComplete(*cmd*)

ARGUMENTS

char	∗*cmd*	(in)	Command string to test for completeness.

DESCRIPTION

Tcl_CommandComplete takes a Tcl command string as argument and determines whether it contains one or more complete commands (i.e. there are no unclosed quotes, braces, brackets, or variable references). If the command string is complete then it returns 1; otherwise it returns 0.

KEYWORDS

complete command, partial command

NAME

Tcl_Concat − concatenate a collection of strings

SYNOPSIS

#include <tcl.h>

char *
Tcl_Concat(*argc, argv*)

ARGUMENTS

int	*argc*	(in)	Number of strings.
char	**argv[]*	(in)	Array of strings to concatenate. Must have *argc* entries.

DESCRIPTION

Tcl_Concat is a utility procedure used by several of the Tcl commands. Given a collection of strings, it concatenates them together into a single string, with the original strings separated by spaces. This procedure behaves differently than **Tcl_Merge**, in that the arguments are simply concatenated: no effort is made to ensure proper list structure. However, in most common usage the arguments will all be proper lists themselves; if this is true, then the result will also have proper list structure.

Tcl_Concat eliminates leading and trailing white space as it copies strings from **argv** to the result. If an element of **argv** consists of nothing but white space, then that string is ignored entirely. This white-space removal was added to make the output of the **concat** command cleaner-looking.

The result string is dynamically allocated using **Tcl_Alloc**; the caller must eventually release the space by calling **Tcl_Free**.

SEE ALSO

Tcl_ConcatObj

KEYWORDS

concatenate, strings

NAME

Tcl_CreateChannel, Tcl_GetChannelInstanceData, Tcl_GetChannelType, Tcl_GetChannelName, Tcl_GetChannelHandle, Tcl_GetChannelMode, Tcl_GetChannelBufferSize, Tcl_SetDefaultTranslation, Tcl_SetChannelBufferSize, Tcl_NotifyChannel, Tcl_BadChannelOption − procedures for creating and manipulating channels

SYNOPSIS

#include <tcl.h>

Tcl_Channel
Tcl_CreateChannel(*typePtr, channelName, instanceData, mask*)

ClientData
Tcl_GetChannelInstanceData(*channel*)

Tcl_ChannelType *
Tcl_GetChannelType(*channel*)

char *
Tcl_GetChannelName(*channel*)

int
Tcl_GetChannelHandle(*channel, direction, handlePtr*)

int
Tcl_GetChannelFlags(*channel*)

Tcl_SetDefaultTranslation(*channel, transMode*)

int
Tcl_GetChannelBufferSize(*channel*)

Tcl_SetChannelBufferSize(*channel, size*)

Tcl_NotifyChannel(*channel, mask*)

int
Tcl_BadChannelOption(*interp, optionName, optionList*)

ARGUMENTS

Tcl_ChannelType	**typePtr*	(in)	Points to a structure containing the addresses of procedures that can be called to perform I/O and other functions on the channel.
char	**channelName*	(in)	The name of this channel, such as **file3**; must not be in use by any other channel. Can be NULL, in which case the channel is created without a name.
ClientData	*instanceData*	(in)	Arbitrary one-word value to be associated with this channel. This value is passed to procedures in *typePtr* when they are invoked.
int	*mask*	(in)	OR-ed combination of **TCL_READABLE** and **TCL_WRITABLE** to indicate whether a channel is

			readable and writable.
Tcl_Channel	*channel*	(in)	The channel to operate on.
int	*direction*	(in)	**TCL_READABLE** means the input handle is wanted; **TCL_WRITABLE** means the output handle is wanted.
ClientData	∗*handlePtr*	(out)	Points to the location where the desired OS-specific handle should be stored.
Tcl_EolTranslation	*transMode*	(in)	The translation mode; one of the constants **TCL_TRANSLATE_AUTO**, **TCL_TRANSLATE_CR**, **TCL_TRANSLATE_LF** and **TCL_TRANSLATE_CRLF**.
int	*size*	(in)	The size, in bytes, of buffers to allocate in this channel.
int	*mask*	(in)	An OR-ed combination of **TCL_READABLE**, **TCL_WRITABLE** and **TCL_EXCEPTION** that indicates events that have occurred on this channel.
Tcl_Interp	∗*interp*	(in)	Current interpreter. (can be NULL)
char	∗*optionName*	(in)	Name of the invalid option.
char	∗*optionList*	(in)	Specific options list (space separated words, without "-") to append to the standard generic options list. Can be NULL for generic options error message only.

DESCRIPTION

Tcl uses a two-layered channel architecture. It provides a generic upper layer to enable C and Tcl programs to perform input and output using the same APIs for a variety of files, devices, sockets etc. The generic C APIs are described in the manual entry for **Tcl_OpenFileChannel**.

The lower layer provides type-specific channel drivers for each type of device supported on each platform. This manual entry describes the C APIs used to communicate between the generic layer and the type-specific channel drivers. It also explains how new types of channels can be added by providing new channel drivers.

Channel drivers consist of a number of components: First, each channel driver provides a **Tcl_ChannelType** structure containing pointers to functions implementing the various operations used by the generic layer to communicate with the channel driver. The **Tcl_ChannelType** structure and the functions referenced by it are described in the section TCL_CHANNELTYPE, below.

Second, channel drivers usually provide a Tcl command to create instances of that type of channel. For example, the Tcl **open** command creates channels that use the file and command channel drivers, and the Tcl **socket** command creates channels that use TCP sockets for network communication.

Third, a channel driver optionally provides a C function to open channel instances of that type. For example, **Tcl_OpenFileChannel** opens a channel that uses the file channel driver, and **Tcl_OpenTcpClient** opens a channel that uses the TCP network protocol. These creation functions typically use **Tcl_CreateChannel** internally to open the channel.

To add a new type of channel you must implement a C API or a Tcl command that opens a channel by invoking **Tcl_CreateChannel**. When your driver calls **Tcl_CreateChannel** it passes in a **Tcl_ChannelType** structure describing the driver's I/O procedures. The generic layer will then invoke the functions referenced in that structure to perform operations on the channel.

Tcl_CreateChannel opens a new channel and associates the supplied *typePtr* and *instanceData* with it. The channel is opened in the mode indicated by *mask*. For a discussion of channel drivers, their operations and the **Tcl_ChannelType** structure, see the section TCL_CHANNELTYPE, below.

Tcl_GetChannelInstanceData returns the instance data associated with the channel in *channel*. This is the same as the *instanceData* argument in the call to **Tcl_CreateChannel** that created this channel.

Tcl_GetChannelType returns a pointer to the **Tcl_ChannelType** structure used by the channel in the *channel* argument. This is the same as the *typePtr* argument in the call to **Tcl_CreateChannel** that created this channel.

Tcl_GetChannelName returns a string containing the name associated with the channel, or NULL if the *channelName* argument to **Tcl_CreateChannel** was NULL.

Tcl_GetChannelHandle places the OS-specific device handle associated with *channel* for the given *direction* in the location specified by *handlePtr* and returns **TCL_OK**. If the channel does not have a device handle for the specified direction, then **TCL_ERROR** is returned instead. Different channel drivers will return different types of handle. Refer to the manual entries for each driver to determine what type of handle is returned.

Tcl_GetChannelMode returns an OR-ed combination of **TCL_READABLE** and **TCL_WRITABLE**, indicating whether the channel is open for input and output.

Tcl_SetDefaultTranslation sets the default end of line translation mode. This mode will be installed as the translation mode for the channel if an attempt is made to output on the channel while it is still in **TCL_TRANSLATE_AUTO** mode. For a description of end of line translation modes, see the manual entry for **fconfigure**.

Tcl_GetChannelBufferSize returns the size, in bytes, of buffers allocated to store input or output in *chan*. If the value was not set by a previous call to **Tcl_SetChannelBufferSize**, described below, then the default value of 4096 is returned.

Tcl_SetChannelBufferSize sets the size, in bytes, of buffers that will be allocated in subsequent operations on the channel to store input or output. The *size* argument should be between ten and one million, allowing buffers of ten bytes to one million bytes. If *size* is outside this range, **Tcl_SetChannelBufferSize** sets the buffer size to 4096.

Tcl_NotifyChannel is called by a channel driver to indicate to the generic layer that the events specified by *mask* have occurred on the channel. Channel drivers are responsible for invoking this function whenever the channel handlers need to be called for the channel. See **WATCHPROC** below for more details.

Tcl_BadChannelOption is called from driver specific set or get option procs to generate a complete error message.

TCL_CHANNELTYPE

A channel driver provides a **Tcl_ChannelType** structure that contains pointers to functions that implement the various operations on a channel; these operations are invoked as needed by the generic layer. The **Tcl_ChannelType** structure contains the following fields:

```
typedef struct Tcl_ChannelType {
    char *typeName;
    Tcl_DriverBlockModeProc *blockModeProc;
    Tcl_DriverCloseProc *closeProc;
    Tcl_DriverInputProc *inputProc;
    Tcl_DriverOutputProc *outputProc;
    Tcl_DriverSeekProc *seekProc;
    Tcl_DriverSetOptionProc *setOptionProc;
    Tcl_DriverGetOptionProc *getOptionProc;
```

```
    Tcl_DriverWatchProc *watchProc;
    Tcl_DriverGetHandleProc *getHandleProc;
} Tcl_ChannelType;
```

The driver must provide implementations for all functions except *blockModeProc*, *seekProc*, *setOptionProc*, and *getOptionProc*, which may be specified as NULL to indicate that the channel does not support seeking. Other functions that can not be implemented for this type of device should return **EINVAL** when invoked to indicate that they are not implemented.

TYPENAME

The *typeName* field contains a null-terminated string that identifies the type of the device implemented by this driver, e.g. **file** or **socket**.

BLOCKMODEPROC

The *blockModeProc* field contains the address of a function called by the generic layer to set blocking and nonblocking mode on the device. *BlockModeProc* should match the following prototype:

```
typedef int Tcl_DriverBlockModeProc(
    ClientData instanceData,
    int mode);
```

The *instanceData* is the same as the value passed to **Tcl_CreateChannel** when this channel was created. The *mode* argument is either **TCL_MODE_BLOCKING** or **TCL_MODE_NONBLOCKING** to set the device into blocking or nonblocking mode. The function should return zero if the operation was successful, or a nonzero POSIX error code if the operation failed.

If the operation is successful, the function can modify the supplied *instanceData* to record that the channel entered blocking or nonblocking mode and to implement the blocking or nonblocking behavior. For some device types, the blocking and nonblocking behavior can be implemented by the underlying operating system; for other device types, the behavior must be emulated in the channel driver.

CLOSEPROC

The *closeProc* field contains the address of a function called by the generic layer to clean up driver-related information when the channel is closed. *CloseProc* must match the following prototype:

```
typedef int Tcl_DriverCloseProc(
    ClientData instanceData,
    Tcl_Interp *interp);
```

The *instanceData* argument is the same as the value provided to **Tcl_CreateChannel** when the channel was created. The function should release any storage maintained by the channel driver for this channel, and close the input and output devices encapsulated by this channel. All queued output will have been flushed to the device before this function is called, and no further driver operations will be invoked on this instance after calling the *closeProc*. If the close operation is successful, the procedure should return zero; otherwise it should return a nonzero POSIX error code. In addition, if an error occurs and *interp* is not NULL, the procedure should store an error message in *interp->result*.

INPUTPROC

The *inputProc* field contains the address of a function called by the generic layer to read data from the file or device and store it in an internal buffer. *InputProc* must match the following prototype:

```
typedef int Tcl_DriverInputProc(
    ClientData instanceData,
    char *buf,
```

```
    int bufSize,
    int *errorCodePtr);
```

InstanceData is the same as the value passed to **Tcl_CreateChannel** when the channel was created. The *buf* argument points to an array of bytes in which to store input from the device, and the *bufSize* argument indicates how many bytes are available at *buf*.

The *errorCodePtr* argument points to an integer variable provided by the generic layer. If an error occurs, the function should set the variable to a POSIX error code that identifies the error that occurred.

The function should read data from the input device encapsulated by the channel and store it at *buf*. On success, the function should return a nonnegative integer indicating how many bytes were read from the input device and stored at *buf*. On error, the function should return -1. If an error occurs after some data has been read from the device, that data is lost.

If *inputProc* can determine that the input device has some data available but less than requested by the *bufSize* argument, the function should only attempt to read as much data as is available and return without blocking. If the input device has no data available whatsoever and the channel is in nonblocking mode, the function should return an **EAGAIN** error. If the input device has no data available whatsoever and the channel is in blocking mode, the function should block for the shortest possible time until at least one byte of data can be read from the device; then, it should return as much data as it can read without blocking.

OUTPUTPROC

The *outputProc* field contains the address of a function called by the generic layer to transfer data from an internal buffer to the output device. *OutputProc* must match the following prototype:

```
typedef int Tcl_DriverOutputProc(
    ClientData instanceData,
    char *buf,
    int toWrite,
    int *errorCodePtr);
```

InstanceData is the same as the value passed to **Tcl_CreateChannel** when the channel was created. The *buf* argument contains an array of bytes to be written to the device, and the *toWrite* argument indicates how many bytes are to be written from the *buf* argument.

The *errorCodePtr* argument points to an integer variable provided by the generic layer. If an error occurs, the function should set this variable to a POSIX error code that identifies the error.

The function should write the data at *buf* to the output device encapsulated by the channel. On success, the function should return a nonnegative integer indicating how many bytes were written to the output device. The return value is normally the same as *toWrite*, but may be less in some cases such as if the output operation is interrupted by a signal. If an error occurs the function should return -1. In case of error, some data may have been written to the device.

If the channel is nonblocking and the output device is unable to absorb any data whatsoever, the function should return -1 with an **EAGAIN** error without writing any data.

SEEKPROC

The *seekProc* field contains the address of a function called by the generic layer to move the access point at which subsequent input or output operations will be applied. *SeekProc* must match the following prototype:

```
typedef int Tcl_DriverSeekProc(
    ClientData instanceData,
    long offset,
    int seekMode,
    int *errorCodePtr);
```

The *instanceData* argument is the same as the value given to **Tcl_CreateChannel** when this channel was created. *Offset* and *seekMode* have the same meaning as for the **Tcl_SeekChannel** procedure (described in the manual entry for **Tcl_OpenFileChannel**).

The *errorCodePtr* argument points to an integer variable provided by the generic layer for returning **errno** values from the function. The function should set this variable to a POSIX error code if an error occurs. The function should store an **EINVAL** error code if the channel type does not implement seeking.

The return value is the new access point or -1 in case of error. If an error occurred, the function should not move the access point.

SETOPTIONPROC

The *setOptionProc* field contains the address of a function called by the generic layer to set a channel type specific option on a channel. *setOptionProc* must match the following prototype:

```
typedef int Tcl_DriverSetOptionProc(
    ClientData instanceData,
    Tcl_Interp *interp,
    char *optionName,
    char *optionValue);
```

optionName is the name of an option to set, and *optionValue* is the new value for that option, as a string. The *instanceData* is the same as the value given to **Tcl_CreateChannel** when this channel was created. The function should do whatever channel type specific action is required to implement the new value of the option.

Some options are handled by the generic code and this function is never called to set them, e.g. **-blockmode**. Other options are specific to each channel type and the *setOptionProc* procedure of the channel driver will get called to implement them. The *setOptionProc* field can be NULL, which indicates that this channel type supports no type specific options.

If the option value is successfully modified to the new value, the function returns **TCL_OK**. It should call **Tcl_BadChannelOption** which itself returns **TCL_ERROR** if the *optionName* is unrecognized. If *optionValue* specifies a value for the option that is not supported or if a system call error occurs, the function should leave an error message in the *result* field of *interp* if *interp* is not NULL. The function should also call **Tcl_SetErrno** to store an appropriate POSIX error code.

GETOPTIONPROC

The *getOptionProc* field contains the address of a function called by the generic layer to get the value of a channel type specific option on a channel. *getOptionProc* must match the following prototype:

```
typedef int Tcl_DriverGetOptionProc(
    ClientData instanceData,
    Tcl_Interp *interp,
    char *optionName,
    Tcl_DString *dsPtr);
```

OptionName is the name of an option supported by this type of channel. If the option name is not NULL, the function stores its current value, as a string, in the Tcl dynamic string *dsPtr*. If *optionName* is NULL, the function stores in *dsPtr* an alternating list of all supported options and their current values. On success, the function returns **TCL_OK**. It should call **Tcl_BadChannelOption** which itself returns **TCL_ERROR** if the *optionName* is unrecognized. If a system call error occurs, the function should leave an error message in the *result* field of *interp* if *interp* is not NULL. The function should also call **Tcl_SetErrno** to store an appropriate POSIX error code.

Some options are handled by the generic code and this function is never called to retrieve their value, e.g. **-blockmode**. Other options are specific to each channel type and the *getOptionProc* procedure of the channel driver will get called to implement them. The *getOptionProc* field can be NULL, which indicates that this channel type supports no type specific options.

WATCHPROC

The *watchProc* field contains the address of a function called by the generic layer to initialize the event notification mechanism to notice events of interest on this channel. *WatchProc* should match the following prototype:

```
typedef void Tcl_DriverWatchProc(
    ClientData instanceData,
    int mask);
```

The *instanceData* is the same as the value passed to **Tcl_CreateChannel** when this channel was created. The *mask* argument is an OR-ed combination of **TCL_READABLE**, **TCL_WRITABLE** and **TCL_EXCEPTION**; it indicates events the caller is interested in noticing on this channel.

The function should initialize device type specific mechanisms to notice when an event of interest is present on the channel. When one or more of the designated events occurs on the channel, the channel driver is responsible for calling **Tcl_NotifyChannel** to inform the generic channel module. The driver should take care not to starve other channel drivers or sources of callbacks by invoking Tcl_NotifyChannel too frequently. Fairness can be insured by using the Tcl event queue to allow the channel event to be scheduled in sequence with other events. See the description of **Tcl_QueueEvent** for details on how to queue an event.

GETHANDLEPROC

The *getHandleProc* field contains the address of a function called by the generic layer to retrieve a device-specific handle from the channel. *GetHandleProc* should match the following prototype:

```
typedef int Tcl_DriverGetHandleProc(
    ClientData instanceData,
    int direction,
    ClientData *handlePtr);
```

InstanceData is the same as the value passed to **Tcl_CreateChannel** when this channel was created. The *direction* argument is either **TCL_READABLE** to retrieve the handle used for input, or **TCL_WRITABLE** to retrieve the handle used for output.

If the channel implementation has device-specific handles, the function should retrieve the appropriate handle associated with the channel, according the *direction* argument. The handle should be stored in the location referred to by *handlePtr*, and **TCL_OK** should be returned. If the channel is not open for the specified direction, or if the channel implementation does not use device handles, the function should return **TCL_ERROR**.

TCL_BADCHANNELOPTION

This procedure generates a "bad option" error message in an (optional) interpreter. It is used by channel drivers when a invalid Set/Get option is requested. Its purpose is to concatenate the generic options list to the specific ones and factorize the generic options error message string.

It always return **TCL_ERROR**

An error message is generated in interp's result object to indicate that a command was invoked with the a bad option The message has the form

```
bad option "blah": should be one of
<...generic options...>+<...specific options...>
```

so you get for instance:

bad option "-blah": should be one of -blocking,
-buffering, -buffersize, -eofchar, -translation,
-peername, or -sockname

when called with optionList="peername sockname"

"blah" is the optionName argument and "<specific options>" is a space separated list of specific option words. The function takes good care of inserting minus signs before each option, commas after, and an "or" before the last option.

SEE ALSO

Tcl_Close(3), Tcl_OpenFileChannel(3), Tcl_SetErrno(3), Tcl_QueueEvent(3)

KEYWORDS

blocking, channel driver, channel registration, channel type, nonblocking

NAME

Tcl_CreateChannelHandler, Tcl_DeleteChannelHandler − call a procedure when a channel becomes readable or writable

SYNOPSIS

#include <tcl.h>

void
Tcl_CreateChannelHandler(*channel, mask, proc, clientData*)

void
Tcl_DeleteChannelHandler(*channel, proc, clientData*)

ARGUMENTS

Tcl_Channel	*channel*	(in)	Tcl channel such as returned by **Tcl_CreateChannel**.
int	*mask*	(in)	Conditions under which *proc* should be called: OR-ed combination of **TCL_READABLE**, **TCL_WRITABLE** and **TCL_EXCEPTION**. Specify a zero value to temporarily disable an existing handler.
Tcl_FileProc	*∗proc*	(in)	Procedure to invoke whenever the channel indicated by *channel* meets the conditions specified by *mask*.
ClientData	*clientData*	(in)	Arbitrary one-word value to pass to *proc*.

DESCRIPTION

Tcl_CreateChannelHandler arranges for *proc* to be called in the future whenever input or output becomes possible on the channel identified by *channel*, or whenever an exceptional condition exists for *channel*. The conditions of interest under which *proc* will be invoked are specified by the *mask* argument. See the manual entry for **fileevent** for a precise description of what it means for a channel to be readable or writable. *Proc* must conform to the following prototype:

```
typedef void Tcl_ChannelProc(
    ClientData clientData,
    int mask);
```

The *clientData* argument is the same as the value passed to **Tcl_CreateChannelHandler** when the handler was created. Typically, *clientData* points to a data structure containing application-specific information about the channel. *Mask* is an integer mask indicating which of the requested conditions actually exists for the channel; it will contain a subset of the bits from the *mask* argument to **Tcl_CreateChannelHandler** when the handler was created.

Each channel handler is identified by a unique combination of *channel*, *proc* and *clientData*. There may be many handlers for a given channel as long as they don't have the same *channel*, *proc*, and *clientData*. If **Tcl_CreateChannelHandler** is invoked when there is already a handler for *channel*, *proc*, and *clientData*, then no new handler is created; instead, the *mask* is changed for the existing handler.

Tcl_DeleteChannelHandler deletes a channel handler identified by *channel*, *proc* and *clientData*; if no such handler exists, the call has no effect.

Channel handlers are invoked via the Tcl event mechanism, so they are only useful in applications that are event-driven. Note also that the conditions specified in the *mask* argument to *proc* may no longer exist when *proc* is invoked: for example, if there are two handlers for **TCL_READABLE** on the same channel, the first handler could consume all of the available input so that the channel is no longer readable when the second handler is invoked. For this reason it may be useful to use nonblocking I/O on channels for which

there are event handlers.

SEE ALSO

Notifier(3), Tcl_CreateChannel(3), Tcl_OpenFileChannel(3), vwait(n).

KEYWORDS

blocking, callback, channel, events, handler, nonblocking.

NAME

Tcl_CreateCloseHandler, Tcl_DeleteCloseHandler − arrange for callbacks when channels are closed

SYNOPSIS

```
#include <tcl.h>

void
Tcl_CreateCloseHandler(channel, proc, clientData)

void
Tcl_DeleteCloseHandler(channel, proc, clientData)
```

ARGUMENTS

Tcl_Channel	*channel*	(in)	The channel for which to create or delete a close callback.
Tcl_CloseProc	**proc*	(in)	The procedure to call as the callback.
ClientData	*clientData*	(in)	Arbitrary one-word value to pass to *proc*.

DESCRIPTION

Tcl_CreateCloseHandler arranges for *proc* to be called when *channel* is closed with **Tcl_Close** or **Tcl_UnregisterChannel**, or using the Tcl **close** command. *Proc* should match the following prototype:

```
typedef void Tcl_CloseProc(
    ClientData clientData);
```

The *clientData* is the same as the value provided in the call to **Tcl_CreateCloseHandler**.

Tcl_DeleteCloseHandler removes a close callback for *channel*. The *proc* and *clientData* identify which close callback to remove; **Tcl_DeleteCloseHandler** does nothing if its *proc* and *clientData* arguments do not match the *proc* and *clientData* for a close handler for *channel*.

SEE ALSO

close(n), Tcl_Close(3), Tcl_UnregisterChannel(3)

KEYWORDS

callback, channel closing

NAME

Tcl_CreateCommand − implement new commands in C

SYNOPSIS

#include <tcl.h>

Tcl_Command
Tcl_CreateCommand(*interp, cmdName, proc, clientData, deleteProc*)

ARGUMENTS

Tcl_Interp	∗*interp*	(in)	Interpreter in which to create new command.
char	∗*cmdName*	(in)	Name of command.
Tcl_CmdProc	∗*proc*	(in)	Implementation of new command: *proc* will be called whenever *cmdName* is invoked as a command.
ClientData	*clientData*	(in)	Arbitrary one-word value to pass to *proc* and *deleteProc*.
Tcl_CmdDeleteProc	∗*deleteProc*	(in)	Procedure to call before *cmdName* is deleted from the interpreter; allows for command-specific cleanup. If NULL, then no procedure is called before the command is deleted.

DESCRIPTION

Tcl_CreateCommand defines a new command in *interp* and associates it with procedure *proc* such that whenever *cmdName* is invoked as a Tcl command (via a call to **Tcl_Eval**) the Tcl interpreter will call *proc* to process the command. It differs from **Tcl_CreateObjCommand** in that a new string-based command is defined; that is, a command procedure is defined that takes an array of argument strings instead of objects. The object-based command procedures registered by **Tcl_CreateObjCommand** can execute significantly faster than the string-based command procedures defined by **Tcl_CreateCommand**. This is because they take Tcl objects as arguments and those objects can retain an internal representation that can be manipulated more efficiently. Also, Tcl's interpreter now uses objects internally. In order to invoke a string-based command procedure registered by **Tcl_CreateCommand**, it must generate and fetch a string representation from each argument object before the call and create a new Tcl object to hold the string result returned by the string-based command procedure. New commands should be defined using **Tcl_CreateObjCommand**. We support **Tcl_CreateCommand** for backwards compatibility.

The procedures **Tcl_DeleteCommand**, **Tcl_GetCommandInfo**, and **Tcl_SetCommandInfo** are used in conjunction with **Tcl_CreateCommand**.

Tcl_CreateCommand will delete an existing command *cmdName*, if one is already associated with the interpreter. It returns a token that may be used to refer to the command in subsequent calls to **Tcl_GetCommandName**. If *cmdName* contains any **::** namespace qualifiers, then the command is added to the specified namespace; otherwise the command is added to the global namespace. If **Tcl_CreateCommand** is called for an interpreter that is in the process of being deleted, then it does not create a new command and it returns NULL. *Proc* should have arguments and result that match the type **Tcl_CmdProc**:

```
typedef int Tcl_CmdProc(
    ClientData clientData,
    Tcl_Interp *interp,
    int argc,
    char *argv[]);
```

When *proc* is invoked the *clientData* and *interp* parameters will be copies of the *clientData* and *interp* arguments given to **Tcl_CreateCommand**. Typically, *clientData* points to an application-specific data structure

that describes what to do when the command procedure is invoked. *Argc* and *argv* describe the arguments to the command, *argc* giving the number of arguments (including the command name) and *argv* giving the values of the arguments as strings. The *argv* array will contain *argc*+1 values; the first *argc* values point to the argument strings, and the last value is NULL.

Proc must return an integer code that is either **TCL_OK**, **TCL_ERROR**, **TCL_RETURN**, **TCL_BREAK**, or **TCL_CONTINUE**. See the Tcl overview man page for details on what these codes mean. Most normal commands will only return **TCL_OK** or **TCL_ERROR**. In addition, *proc* must set the interpreter result to point to a string value; in the case of a **TCL_OK** return code this gives the result of the command, and in the case of **TCL_ERROR** it gives an error message. The **Tcl_SetResult** procedure provides an easy interface for setting the return value; for complete details on how the the interpreter result field is managed, see the **Tcl_Interp** man page. Before invoking a command procedure, **Tcl_Eval** sets the interpreter result to point to an empty string, so simple commands can return an empty result by doing nothing at all.

The contents of the *argv* array belong to Tcl and are not guaranteed to persist once *proc* returns: *proc* should not modify them, nor should it set the interpreter result to point anywhere within the *argv* values. Call **Tcl_SetResult** with status **TCL_VOLATILE** if you want to return something from the *argv* array.

DeleteProc will be invoked when (if) *cmdName* is deleted. This can occur through a call to **Tcl_DeleteCommand** or **Tcl_DeleteInterp**, or by replacing *cmdName* in another call to **Tcl_CreateCommand**. *DeleteProc* is invoked before the command is deleted, and gives the application an opportunity to release any structures associated with the command. *DeleteProc* should have arguments and result that match the type **Tcl_CmdDeleteProc**:

```
typedef void Tcl_CmdDeleteProc(ClientData clientData);
```

The *clientData* argument will be the same as the *clientData* argument passed to **Tcl_CreateCommand**.

SEE ALSO

Tcl_CreateObjCommand, Tcl_DeleteCommand, Tcl_GetCommandInfo, Tcl_SetCommandInfo, Tcl_GetCommandName, Tcl_SetObjResult

KEYWORDS

bind, command, create, delete, interpreter, namespace

NAME

Tcl_CreateFileHandler, Tcl_DeleteFileHandler − associate procedure callbacks with files or devices (Unix only)

SYNOPSIS

#include <tcl.h>

Tcl_CreateFileHandler(*fd*, *mask*, *proc*, *clientData*)

Tcl_DeleteFileHandler(*fd*)

ARGUMENTS

int	*fd*	(in)	Unix file descriptor for an open file or device.
int	*mask*	(in)	Conditions under which *proc* should be called: OR-ed combination of **TCL_READABLE**, **TCL_WRITABLE**, and **TCL_EXCEPTION**. May be set to 0 to temporarily disable a handler.
Tcl_FileProc	∗*proc*	(in)	Procedure to invoke whenever the file or device indicated by *file* meets the conditions specified by *mask*.
ClientData	*clientData*	(in)	Arbitrary one-word value to pass to *proc*.

DESCRIPTION

Tcl_CreateFileHandler arranges for *proc* to be invoked in the future whenever I/O becomes possible on a file or an exceptional condition exists for the file. The file is indicated by *fd*, and the conditions of interest are indicated by *mask*. For example, if *mask* is **TCL_READABLE**, *proc* will be called when the file is readable. The callback to *proc* is made by **Tcl_DoOneEvent**, so **Tcl_CreateFileHandler** is only useful in programs that dispatch events through **Tcl_DoOneEvent** or through Tcl commands such as **vwait**.

Proc should have arguments and result that match the type **Tcl_FileProc**:

```
typedef void Tcl_FileProc(
    ClientData clientData,
    int mask);
```

The *clientData* parameter to *proc* is a copy of the *clientData* argument given to **Tcl_CreateFileHandler** when the callback was created. Typically, *clientData* points to a data structure containing application-specific information about the file. *Mask* is an integer mask indicating which of the requested conditions actually exists for the file; it will contain a subset of the bits in the *mask* argument to **Tcl_CreateFileHandler**.

There may exist only one handler for a given file at a given time. If **Tcl_CreateFileHandler** is called when a handler already exists for *fd*, then the new callback replaces the information that was previously recorded.

Tcl_DeleteFileHandler may be called to delete the file handler for *fd*; if no handler exists for the file given by *fd* then the procedure has no effect.

The purpose of file handlers is to enable an application to respond to events while waiting for files to become ready for I/O. For this to work correctly, the application may need to use non-blocking I/O operations on the files for which handlers are declared. Otherwise the application may block if it reads or writes too much data; while waiting for the I/O to complete the application won't be able to service other events. Use **Tcl_SetChannelOption** with **−blocking** to set the channel into blocking or nonblocking mode as required.

Note that these interfaces are only supported by the Unix implementation of the Tcl notifier.

KEYWORDS

callback, file, handler

NAME

Tcl_CreateInterp, Tcl_DeleteInterp, Tcl_InterpDeleted − create and delete Tcl command interpreters

SYNOPSIS

#include <tcl.h>

Tcl_Interp ∗
Tcl_CreateInterp()

Tcl_DeleteInterp(*interp*)

int
Tcl_InterpDeleted(*interp*)

ARGUMENTS

Tcl_Interp	∗*interp*	(in)	Token for interpreter to be destroyed.

DESCRIPTION

Tcl_CreateInterp creates a new interpreter structure and returns a token for it. The token is required in calls to most other Tcl procedures, such as **Tcl_CreateCommand**, **Tcl_Eval**, and **Tcl_DeleteInterp**. Clients are only allowed to access a few of the fields of Tcl_Interp structures; see the Tcl_Interp and **Tcl_CreateCommand** man pages for details. The new interpreter is initialized with no defined variables and only the built-in Tcl commands. To bind in additional commands, call **Tcl_CreateCommand**.

Tcl_DeleteInterp marks an interpreter as deleted; the interpreter will eventually be deleted when all calls to **Tcl_Preserve** for it have been matched by calls to **Tcl_Release**. At that time, all of the resources associated with it, including variables, procedures, and application-specific command bindings, will be deleted. After **Tcl_DeleteInterp** returns any attempt to use **Tcl_Eval** on the interpreter will fail and return **TCL_ERROR**. After the call to **Tcl_DeleteInterp** it is safe to examine *interp->result*, query or set the values of variables, define, undefine or retrieve procedures, and examine the runtime evaluation stack. See below, in the section **INTERPRETERS AND MEMORY MANAGEMENT** for details.

Tcl_InterpDeleted returns nonzero if **Tcl_DeleteInterp** was called with *interp* as its argument; this indicates that the interpreter will eventually be deleted, when the last call to **Tcl_Preserve** for it is matched by a call to **Tcl_Release**. If nonzero is returned, further calls to **Tcl_Eval** in this interpreter will return **TCL_ERROR**.

Tcl_InterpDeleted is useful in deletion callbacks to distinguish between when only the memory the callback is responsible for is being deleted and when the whole interpreter is being deleted. In the former case the callback may recreate the data being deleted, but this would lead to an infinite loop if the interpreter were being deleted.

INTERPRETERS AND MEMORY MANAGEMENT

Tcl_DeleteInterp can be called at any time on an interpreter that may be used by nested evaluations and C code in various extensions. Tcl implements a simple mechanism that allows callers to use interpreters without worrying about the interpreter being deleted in a nested call, and without requiring special code to protect the interpreter, in most cases. This mechanism ensures that nested uses of an interpreter can safely continue using it even after **Tcl_DeleteInterp** is called.

The mechanism relies on matching up calls to **Tcl_Preserve** with calls to **Tcl_Release**. If **Tcl_DeleteInterp** has been called, only when the last call to **Tcl_Preserve** is matched by a call to **Tcl_Release**, will the interpreter be freed. See the manual entry for **Tcl_Preserve** for a description of these functions.

The rules for when the user of an interpreter must call **Tcl_Preserve** and **Tcl_Release** are simple:

Interpreters Passed As Arguments
: Functions that are passed an interpreter as an argument can safely use the interpreter without any special protection. Thus, when you write an extension consisting of new Tcl commands, no special code is needed to protect interpreters received as arguments. This covers the majority of all uses.

Interpreter Creation And Deletion
: When a new interpreter is created and used in a call to **Tcl_Eval**, **Tcl_VarEval**, **Tcl_GlobalEval**, **Tcl_SetVar**, or **Tcl_GetVar**, a pair of calls to **Tcl_Preserve** and **Tcl_Release** should be wrapped around all uses of the interpreter. Remember that it is unsafe to use the interpreter once **Tcl_Release** has been called. To ensure that the interpreter is properly deleted when it is no longer needed, call **Tcl_InterpDeleted** to test if some other code already called **Tcl_DeleteInterp**; if not, call **Tcl_DeleteInterp** before calling **Tcl_Release** in your own code. Do not call **Tcl_DeleteInterp** on an interpreter for which **Tcl_InterpDeleted** returns nonzero.

Retrieving An Interpreter From A Data Structure
: When an interpreter is retrieved from a data structure (e.g. the client data of a callback) for use in **Tcl_Eval**, **Tcl_VarEval**, **Tcl_GlobalEval**, **Tcl_SetVar**, or **Tcl_GetVar**, a pair of calls to **Tcl_Preserve** and **Tcl_Release** should be wrapped around all uses of the interpreter; it is unsafe to reuse the interpreter once **Tcl_Release** has been called. If an interpreter is stored inside a callback data structure, an appropriate deletion cleanup mechanism should be set up by the code that creates the data structure so that the interpreter is removed from the data structure (e.g. by setting the field to NULL) when the interpreter is deleted. Otherwise, you may be using an interpreter that has been freed and whose memory may already have been reused.

All uses of interpreters in Tcl and Tk have already been protected. Extension writers should ensure that their code also properly protects any additional interpreters used, as described above.

KEYWORDS

command, create, delete, interpreter

SEE ALSO

Tcl_Preserve(3), Tcl_Release(3)

NAME

Tcl_CreateMathFunc − Define a new math function for expressions

SYNOPSIS

#include <tcl.h>

Tcl_CreateMathFunc(*interp, name, numArgs, argTypes, proc, clientData*)

ARGUMENTS

Tcl_Interp	∗*interp*	(in)	Interpreter in which new function will be defined.
char	∗*name*	(in)	Name for new function.
int	*numArgs*	(in)	Number of arguments to new function; also gives size of *argTypes* array.
Tcl_ValueType	∗*argTypes*	(in)	Points to an array giving the permissible types for each argument to function.
Tcl_MathProc	∗*proc*	(in)	Procedure that implements the function.
ClientData	*clientData*	(in)	Arbitrary one-word value to pass to *proc* when it is invoked.

DESCRIPTION

Tcl allows a number of mathematical functions to be used in expressions, such as **sin**, **cos**, and **hypot**. **Tcl_CreateMathFunc** allows applications to add additional functions to those already provided by Tcl or to replace existing functions. *Name* is the name of the function as it will appear in expressions. If *name* doesn't already exist as a function then a new function is created. If it does exist, then the existing function is replaced. *NumArgs* and *argTypes* describe the arguments to the function. Each entry in the *argTypes* array must be either TCL_INT, TCL_DOUBLE, or TCL_EITHER to indicate whether the corresponding argument must be an integer, a double-precision floating value, or either, respectively.

Whenever the function is invoked in an expression Tcl will invoke *proc*. *Proc* should have arguments and result that match the type **Tcl_MathProc**:

```
typedef int Tcl_MathProc(
    ClientData clientData,
    Tcl_Interp *interp,
    Tcl_Value *args,
    Tcl_Value *resultPtr);
```

When *proc* is invoked the *clientData* and *interp* arguments will be the same as those passed to **Tcl_CreateMathFunc**. *Args* will point to an array of *numArgs* Tcl_Value structures, which describe the actual arguments to the function:

```
typedef struct Tcl_Value {
    Tcl_ValueType type;
    long intValue;
    double doubleValue;
} Tcl_Value;
```

The *type* field indicates the type of the argument and is either TCL_INT or TCL_DOUBLE. It will match the *argTypes* value specified for the function unless the *argTypes* value was TCL_EITHER. Tcl converts the argument supplied in the expression to the type requested in *argTypes*, if that is necessary. Depending on the value of the *type* field, the *intValue* or *doubleValue* field will contain the actual value of the argument.

Proc should compute its result and store it either as an integer in *resultPtr->intValue* or as a floating value in *resultPtr->doubleValue*. It should set also *resultPtr->type* to either TCL_INT or TCL_DOUBLE to indicate which value was set. Under normal circumstances *proc* should return TCL_OK. If an error occurs

while executing the function, *proc* should return TCL_ERROR and leave an error message in *interp->result*.

KEYWORDS

expression, mathematical function

NAME

Tcl_CreateObjCommand, Tcl_DeleteCommand, Tcl_DeleteCommandFromToken, Tcl_GetCommandInfo, Tcl_SetCommandInfo, Tcl_GetCommandName − implement new commands in C

SYNOPSIS

#include <tcl.h>

Tcl_Command
Tcl_CreateObjCommand(*interp, cmdName, proc, clientData, deleteProc*)

int
Tcl_DeleteCommand(*interp, cmdName*)

int
Tcl_DeleteCommandFromToken(*interp, token*)

int
Tcl_GetCommandInfo(*interp, cmdName, infoPtr*)

int
Tcl_SetCommandInfo(*interp, cmdName, infoPtr*)

char ∗
Tcl_GetCommandName(*interp, token*)

ARGUMENTS

Tcl_Interp	∗*interp*	(in)	Interpreter in which to create a new command or that contains a command.
char	∗*cmdName*	(in)	Name of command.
Tcl_ObjCmdProc	∗*proc*	(in)	Implementation of the new command: *proc* will be called whenever *cmdName* is invoked as a command.
ClientData	*clientData*	(in)	Arbitrary one-word value to pass to *proc* and *deleteProc*.
Tcl_CmdDeleteProc	∗*deleteProc*	(in)	Procedure to call before *cmdName* is deleted from the interpreter; allows for command-specific cleanup. If NULL, then no procedure is called before the command is deleted.
Tcl_Command	*token*	(in)	Token for command, returned by previous call to **Tcl_CreateObjCommand**. The command must not have been deleted.
Tcl_CmdInfo	∗*infoPtr*	(in/out)	Pointer to structure containing various information about a Tcl command.

DESCRIPTION

Tcl_CreateObjCommand defines a new command in *interp* and associates it with procedure *proc* such that whenever *name* is invoked as a Tcl command (e.g., via a call to **Tcl_EvalObj**) the Tcl interpreter will call *proc* to process the command.

Tcl_CreateObjCommand will delete any command *name* already associated with the interpreter. It returns a token that may be used to refer to the command in subsequent calls to **Tcl_GetCommandName**. If *name* contains any **::** namespace qualifiers, then the command is added to the specified namespace; otherwise the command is added to the global namespace. If **Tcl_CreateObjCommand** is called for an interpreter that is in the process of being deleted, then it does not create a new command and it returns NULL.

proc should have arguments and result that match the type **Tcl_ObjCmdProc**:

```
typedef int Tcl_ObjCmdProc(
    ClientData clientData,
    Tcl_Interp *interp,
    int objc,
    Tcl_Obj *CONST objv[]);
```

When *proc* is invoked, the *clientData* and *interp* parameters will be copies of the *clientData* and *interp* arguments given to **Tcl_CreateObjCommand**. Typically, *clientData* points to an application-specific data structure that describes what to do when the command procedure is invoked. *Objc* and *objv* describe the arguments to the command, *objc* giving the number of argument objects (including the command name) and *objv* giving the values of the arguments. The *objv* array will contain *objc* values, pointing to the argument objects. Unlike *argv*[*argv*] used in a string-based command procedure, *objv*[*objc*] will not contain NULL.

Additionally, when *proc* is invoked, it must not modify the contents of the *objv* array by assigning new pointer values to any element of the array (for example, *objv*[**2**] = **NULL**) because this will cause memory to be lost and the runtime stack to be corrupted. The **CONST** in the declaration of *objv* will cause ANSI-compliant compilers to report any such attempted assignment as an error. However, it is acceptable to modify the internal representation of any individual object argument. For instance, the user may call **Tcl_GetIntFromObject** on *objv*[**2**] to obtain the integer representation of that object; that call may change the type of the object that *objv*[**2**] points at, but will not change where *objv*[**2**] points.

proc must return an integer code that is either **TCL_OK**, **TCL_ERROR**, **TCL_RETURN**, **TCL_BREAK**, or **TCL_CONTINUE**. See the Tcl overview man page for details on what these codes mean. Most normal commands will only return **TCL_OK** or **TCL_ERROR**. In addition, if *proc* needs to return a non-empty result, it can call **Tcl_SetObjResult** to set the interpreter's result. In the case of a **TCL_OK** return code this gives the result of the command, and in the case of **TCL_ERROR** this gives an error message. Before invoking a command procedure, **Tcl_EvalObj** sets interpreter's result to point to an object representing an empty string, so simple commands can return an empty result by doing nothing at all.

The contents of the *objv* array belong to Tcl and are not guaranteed to persist once *proc* returns: *proc* should not modify them. Call **Tcl_SetObjResult** if you want to return something from the *objv* array.

DeleteProc will be invoked when (if) *name* is deleted. This can occur through a call to **Tcl_DeleteCommand**, **Tcl_DeleteCommandFromToken**, or **Tcl_DeleteInterp**, or by replacing *name* in another call to **Tcl_CreateObjCommand**. *DeleteProc* is invoked before the command is deleted, and gives the application an opportunity to release any structures associated with the command. *DeleteProc* should have arguments and result that match the type **Tcl_CmdDeleteProc**:

```
typedef void Tcl_CmdDeleteProc(ClientData clientData);
```

The *clientData* argument will be the same as the *clientData* argument passed to **Tcl_CreateObjCommand**.

Tcl_DeleteCommand deletes a command from a command interpreter. Once the call completes, attempts to invoke *cmdName* in *interp* will result in errors. If *cmdName* isn't bound as a command in *interp* then **Tcl_DeleteCommand** does nothing and returns -1; otherwise it returns 0. There are no restrictions on *cmdName*: it may refer to a built-in command, an application-specific command, or a Tcl procedure. If *name* contains any **::** namespace qualifiers, the command is deleted from the specified namespace.

Given a token returned by **Tcl_CreateObjCommand**, **Tcl_DeleteCommandFromToken** deletes the command from a command interpreter. It will delete a command even if that command has been renamed. Once the call completes, attempts to invoke the command in *interp* will result in errors. If the command corresponding to *token* has already been deleted from *interp* then **Tcl_DeleteCommand** does nothing and returns -1; otherwise it returns 0.

Tcl_GetCommandInfo checks to see whether its *cmdName* argument exists as a command in *interp*. *cmdName* may include **::** namespace qualifiers to identify a command in a particular namespace. If the command is not found, then it returns 0. Otherwise it places information about the command in the **Tcl_CmdInfo** structure pointed to by *infoPtr* and returns 1. A **Tcl_CmdInfo** structure has the following fields:

```
typedef struct Tcl_CmdInfo {
    int isNativeObjectProc;
    Tcl_ObjCmdProc *objProc;
    ClientData objClientData;
    Tcl_CmdProc *proc;
    ClientData clientData;
    Tcl_CmdDeleteProc *deleteProc;
    ClientData deleteData;
    Tcl_Namespace *namespacePtr;
} Tcl_CmdInfo;
```

The *isNativeObjectProc* field has the value 1 if **Tcl_CreateObjCommand** was called to register the command; it is 0 if only **Tcl_CreateCommand** was called. It allows a program to determine whether it is faster to call *objProc* or *proc*: *objProc* is normally faster if *isNativeObjectProc* has the value 1. The fields *objProc* and *objClientData* have the same meaning as the *proc* and *clientData* arguments to **Tcl_CreateObjCommand**; they hold information about the object-based command procedure that the Tcl interpreter calls to implement the command. The fields *proc* and *clientData* hold information about the string-based command procedure that implements the command. If **Tcl_CreateCommand** was called for this command, this is the procedure passed to it; otherwise, this is a compatibility procedure registered by **Tcl_CreateObjCommand** that simply calls the command's object-based procedure after converting its string arguments to Tcl objects. The field *deleteData* is the ClientData value to pass to *deleteProc*; it is normally the same as *clientData* but may be set independently using the **Tcl_SetCommandInfo** procedure. The field *namespacePtr* holds a pointer to the Tcl_Namespace that contains the command.

Tcl_SetCommandInfo is used to modify the procedures and ClientData values associated with a command. Its *cmdName* argument is the name of a command in *interp*. *cmdName* may include **::** namespace qualifiers to identify a command in a particular namespace. If this command does not exist then **Tcl_SetCommandInfo** returns 0. Otherwise, it copies the information from **infoPtr* to Tcl's internal structure for the command and returns 1. Note that this procedure allows the ClientData for a command's deletion procedure to be given a different value than the ClientData for its command procedure. Note that **Tcl_SetCmdInfo** will not change a command's namespace; you must use **Tcl_RenameCommand** to do that.

Tcl_GetCommandName provides a mechanism for tracking commands that have been renamed. Given a token returned by **Tcl_CreateObjCommand** when the command was created, **Tcl_GetCommandName** returns the string name of the command. If the command has been renamed since it was created, then **Tcl_GetCommandName** returns the current name. This name does not include any **::** namespace qualifiers. The command corresponding to *token* must not have been deleted. The string returned by **Tcl_GetCommandName** is in dynamic memory owned by Tcl and is only guaranteed to retain its value as long as the command isn't deleted or renamed; callers should copy the string if they need to keep it for a long time.

SEE ALSO

Tcl_CreateCommand, Tcl_ResetResult, Tcl_SetObjResult

KEYWORDS

bind, command, create, delete, namespace, object

NAME

Tcl_IsSafe, Tcl_MakeSafe, Tcl_CreateSlave, Tcl_GetSlave, Tcl_GetMaster, Tcl_GetInterpPath, Tcl_CreateAlias, Tcl_CreateAliasObj, Tcl_GetAlias, Tcl_GetAliasObj, Tcl_ExposeCommand, Tcl_HideCommand − manage multiple Tcl interpreters, aliases and hidden commands.

SYNOPSIS

#include <tcl.h>

int
Tcl_IsSafe(*interp*)

int
Tcl_MakeSafe(*interp*)

Tcl_Interp ∗
Tcl_CreateSlave(*interp, slaveName, isSafe*)

Tcl_Interp ∗
Tcl_GetSlave(*interp, slaveName*)

Tcl_Interp ∗
Tcl_GetMaster(*interp*)

int
Tcl_GetInterpPath(*askingInterp, slaveInterp*)

int
Tcl_CreateAlias(*slaveInterp, srcCmd, targetInterp, targetCmd, argc, argv*)

int
Tcl_CreateAliasObj(*slaveInterp, srcCmd, targetInterp, targetCmd, objc, objv*)

int
Tcl_GetAlias(*interp, srcCmd, targetInterpPtr, targetCmdPtr, argcPtr, argvPtr*)

int
Tcl_GetAliasObj(*interp, srcCmd, targetInterpPtr, targetCmdPtr, objcPtr, objvPtr*)

int
Tcl_ExposeCommand(*interp, hiddenCmdName, cmdName*)

int
Tcl_HideCommand(*interp, cmdName, hiddenCmdName*)

ARGUMENTS

Tcl_Interp	∗*interp*	(in)	Interpreter in which to execute the specified command.
char	∗*slaveName*	(in)	Name of slave interpreter to create or manipulate.
int	*isSafe*	(in)	If non-zero, a ‘‘safe’’ slave that is suitable for running untrusted code is created, otherwise a trusted slave is created.

Tcl_Interp	∗*slaveInterp*	(in)	Interpreter to use for creating the source command for an alias (see below).
char	∗*srcCmd*	(in)	Name of source command for alias.
Tcl_Interp	∗*targetInterp*	(in)	Interpreter that contains the target command for an alias.
char	∗*targetCmd*	(in)	Name of target command for alias in *targetInterp*.
int	*argc*	(in)	Count of additional arguments to pass to the alias command.
char	∗∗*argv*	(in)	Vector of strings, the additional arguments to pass to the alias command. This storage is owned by the caller.
int	*objc*	(in)	Count of additional object arguments to pass to the alias object command.
Tcl_Object	∗∗*objv*	(in)	Vector of Tcl_Obj structures, the additional object argumenst to pass to the alias object command. This storage is owned by the caller.
Tcl_Interp	∗∗*targetInterpPtr*	(in)	Pointer to location to store the address of the interpreter where a target command is defined for an alias.
char	∗∗*targetCmdPtr*	(out)	Pointer to location to store the address of the name of the target command for an alias.
int	∗*argcPtr*	(out)	Pointer to location to store count of additional arguments to be passed to the alias. The location is in storage owned by the caller.
char	∗∗∗*argvPtr*	(out)	Pointer to location to store a vector of strings, the additional arguments to pass to an alias. The location is in storage owned by the caller, the vector of strings is owned by the called function.
int	∗*objcPtr*	(out)	Pointer to location to store count of additional object arguments to be passed to the alias. The location is in storage owned by the caller.
Tcl_Obj	∗∗∗*objvPtr*	(out)	Pointer to location to store a vector of Tcl_Obj structures, the additional arguments to pass to an object alias command. The location is in storage owned by the caller, the vector of Tcl_Obj structures is owned by the called function.
char	∗*cmdName*	(in)	Name of an exposed command to hide or create.
char	∗*hiddenCmdName*	(in)	Name under which a hidden command is stored and with which it can be exposed or invoked.

DESCRIPTION

These procedures are intended for access to the multiple interpreter facility from inside C programs. They enable managing multiple interpreters in a hierarchical relationship, and the management of aliases, commands that when invoked in one interpreter execute a command in another interpreter. The return value for those procedures that return an **int** is either **TCL_OK** or **TCL_ERROR**. If **TCL_ERROR** is returned then the **result** field of the interpreter contains an error message.

Tcl_CreateSlave creates a new interpreter as a slave of *interp*. It also creates a slave command named *slaveName* in *interp* which allows *interp* to manipulate the new slave. If *isSafe* is zero, the command creates a trusted slave in which Tcl code has access to all the Tcl commands. If it is **1**, the command creates a ''safe'' slave in which Tcl code has access only to set of Tcl commands defined as ''Safe Tcl''; see the manual entry for the Tcl **interp** command for details. If the creation of the new slave interpreter failed, **NULL** is returned.

Tcl_IsSafe returns **1** if *interp* is ''safe'' (was created with the **TCL_SAFE_INTERPRETER** flag specified), **0** otherwise.

Tcl_MakeSafe makes *interp* ''safe'' by removing all non-core and core unsafe functionality. Note that if you call this after adding some extension to an interpreter, all traces of that extension will be removed from the interpreter.

Tcl_GetSlave returns a pointer to a slave interpreter of *interp*. The slave interpreter is identified by *slaveName*. If no such slave interpreter exists, **NULL** is returned.

Tcl_GetMaster returns a pointer to the master interpreter of *interp*. If *interp* has no master (it is a top-level interpreter) then **NULL** is returned.

Tcl_GetInterpPath sets the *result* field in *askingInterp* to the relative path between *askingInterp* and *slaveInterp*; *slaveInterp* must be a slave of *askingInterp*. If the computation of the relative path succeeds, **TCL_OK** is returned, else **TCL_ERROR** is returned and the *result* field in *askingInterp* contains the error message.

Tcl_CreateAlias creates an object command named *srcCmd* in *slaveInterp* that when invoked, will cause the command *targetCmd* to be invoked in *targetInterp*. The arguments specified by the strings contained in *argv* are always prepended to any arguments supplied in the invocation of *srcCmd* and passed to *targetCmd*. This operation returns **TCL_OK** if it succeeds, or **TCL_ERROR** if it fails; in that case, an error message is left in the object result of *slaveInterp*. Note that there are no restrictions on the ancestry relationship (as created by **Tcl_CreateSlave**) between *slaveInterp* and *targetInterp*. Any two interpreters can be used, without any restrictions on how they are related.

Tcl_CreateAliasObj is similar to **Tcl_CreateAliasObj** except that it takes a vector of objects to pass as additional arguments instead of a vector of strings.

Tcl_GetAlias returns information about an alias *aliasName* in *interp*. Any of the result fields can be **NULL**, in which case the corresponding datum is not returned. If a result field is non−**NULL**, the address indicated is set to the corresponding datum. For example, if *targetNamePtr* is non−**NULL** it is set to a pointer to the string containing the name of the target command.

Tcl_GetAliasObj is similar to **Tcl_GetAlias** except that it returns a pointer to a vector of Tcl_Obj structures instead of a vector of strings.

Tcl_ExposeCommand moves the command named *hiddenCmdName* from the set of hidden commands to the set of exposed commands, putting it under the name *cmdName*. *HiddenCmdName* must be the name of an existing hidden command, or the operation will return **TCL_ERROR** and leave an error message in the *result* field in *interp*. If an exposed command named *cmdName* already exists, the operation returns **TCL_ERROR** and leaves an error message in the object result of *interp*. If the operation succeeds, it returns **TCL_OK**. After executing this command, attempts to use *cmdName* in a call to **Tcl_Eval** or with the Tcl **eval** command will again succeed.

Tcl_HideCommand moves the command named *cmdName* from the set of exposed commands to the set of hidden commands, under the name *hiddenCmdName*. *CmdName* must be the name of an existing exposed command, or the operation will return **TCL_ERROR** and leave an error message in the object result of *interp*. Currently both *cmdName* and *hiddenCmdName* must not contain namespace qualifiers, or the operation will return **TCL_ERROR** and leave an error message in the object result of *interp*. The *CmdName* will be looked up in the global namespace, and not relative to the current namespace, even if the current namespace is not the global one. If a hidden command whose name is *hiddenCmdName* already exists, the operation also returns **TCL_ERROR** and the *result* field in *interp* contains an error message. If the operation succeeds, it returns **TCL_OK**. After executing this command, attempts to use *cmdName* in a call to **Tcl_Eval** or with the Tcl **eval** command will fail.

SEE ALSO

For a description of the Tcl interface to multiple interpreters, see *interp(n)*.

KEYWORDS

alias, command, exposed commands, hidden commands, interpreter, invoke, master, slave,

NAME

Tcl_CreateTimerHandler, Tcl_DeleteTimerHandler − call a procedure at a given time

SYNOPSIS

#include <tcl.h>

Tcl_TimerToken
Tcl_CreateTimerHandler(*milliseconds, proc, clientData*)

Tcl_DeleteTimerHandler(*token*)

ARGUMENTS

int	*milliseconds*	(in)	How many milliseconds to wait before invoking *proc*.
Tcl_TimerProc	∗*proc*	(in)	Procedure to invoke after *milliseconds* have elapsed.
ClientData	*clientData*	(in)	Arbitrary one-word value to pass to *proc*.
Tcl_TimerToken	*token*	(in)	Token for previously-created timer handler (the return value from some previous call to **Tcl_CreateTimerHandler**).

DESCRIPTION

Tcl_CreateTimerHandler arranges for *proc* to be invoked at a time *milliseconds* milliseconds in the future. The callback to *proc* will be made by **Tcl_DoOneEvent**, so **Tcl_CreateTimerHandler** is only useful in programs that dispatch events through **Tcl_DoOneEvent** or through Tcl commands such as **vwait**. The call to *proc* may not be made at the exact time given by *milliseconds*: it will be made at the next opportunity after that time. For example, if **Tcl_DoOneEvent** isn't called until long after the time has elapsed, or if there are other pending events to process before the call to *proc*, then the call to *proc* will be delayed.

Proc should have arguments and return value that match the type **Tcl_TimerProc**:

```
typedef void Tcl_TimerProc(ClientData clientData);
```

The *clientData* parameter to *proc* is a copy of the *clientData* argument given to **Tcl_CreateTimerHandler** when the callback was created. Typically, *clientData* points to a data structure containing application-specific information about what to do in *proc*.

Tcl_DeleteTimerHandler may be called to delete a previously-created timer handler. It deletes the handler indicated by *token* so that no call to *proc* will be made; if that handler no longer exists (e.g. because the time period has already elapsed and *proc* has been invoked then **Tcl_DeleteTimerHandler** does nothing. The tokens returned by **Tcl_CreateTimerHandler** never have a value of NULL, so if NULL is passed to **Tcl_DeleteTimerHandler** then the procedure does nothing.

KEYWORDS

callback, clock, handler, timer

NAME

Tcl_CreateTrace, Tcl_DeleteTrace − arrange for command execution to be traced

SYNOPSIS

#include <tcl.h>

Tcl_Trace
Tcl_CreateTrace(*interp, level, proc, clientData*)

Tcl_DeleteTrace(*interp, trace*)

ARGUMENTS

Tcl_Interp	∗*interp*	(in)	Interpreter containing command to be traced or untraced.
int	*level*	(in)	Only commands at or below this nesting level will be traced. 1 means top-level commands only, 2 means top-level commands or those that are invoked as immediate consequences of executing top-level commands (procedure bodies, bracketed commands, etc.) and so on.
Tcl_CmdTraceProc	∗*proc*	(in)	Procedure to call for each command that's executed. See below for details on the calling sequence.
ClientData	*clientData*	(in)	Arbitrary one-word value to pass to *proc*.
Tcl_Trace	*trace*	(in)	Token for trace to be removed (return value from previous call to **Tcl_CreateTrace**).

DESCRIPTION

Tcl_CreateTrace arranges for command tracing. From now on, *proc* will be invoked before Tcl calls command procedures to process commands in *interp*. The return value from **Tcl_CreateTrace** is a token for the trace, which may be passed to **Tcl_DeleteTrace** to remove the trace. There may be many traces in effect simultaneously for the same command interpreter.

Proc should have arguments and result that match the type **Tcl_CmdTraceProc**:

```
typedef void Tcl_CmdTraceProc(
    ClientData clientData,
    Tcl_Interp *interp,
    int level,
    char *command,
    Tcl_CmdProc *cmdProc,
    ClientData cmdClientData,
    int argc,
    char *argv[]);
```

The *clientData* and *interp* parameters are copies of the corresponding arguments given to **Tcl_CreateTrace**. *ClientData* typically points to an application-specific data structure that describes what to do when *proc* is invoked. *Level* gives the nesting level of the command (1 for top-level commands passed to **Tcl_Eval** by the application, 2 for the next-level commands passed to **Tcl_Eval** as part of parsing or interpreting level-1 commands, and so on). *Command* points to a string containing the text of the command, before any argument substitution. *CmdProc* contains the address of the command procedure that will be called to process the command (i.e. the *proc* argument of some previous call to **Tcl_CreateCommand**) and *cmdClientData* contains the associated client data for *cmdProc* (the *clientData* value passed to **Tcl_CreateCommand**). *Argc* and *argv* give the final argument information that will be passed to *cmdProc*, after command, variable, and backslash substitution. *Proc* must not modify the *command* or *argv* strings.

Tracing will only occur for commands at nesting level less than or equal to the *level* parameter (i.e. the *level* parameter to *proc* will always be less than or equal to the *level* parameter to **Tcl_CreateTrace**).

Calls to *proc* will be made by the Tcl parser immediately before it calls the command procedure for the command (*cmdProc*). This occurs after argument parsing and substitution, so tracing for substituted commands occurs before tracing of the commands containing the substitutions. If there is a syntax error in a command, or if there is no command procedure associated with a command name, then no tracing will occur for that command. If a string passed to Tcl_Eval contains multiple commands (bracketed, or on different lines) then multiple calls to *proc* will occur, one for each command. The *command* string for each of these trace calls will reflect only a single command, not the entire string passed to Tcl_Eval.

Tcl_DeleteTrace removes a trace, so that no future calls will be made to the procedure associated with the trace. After **Tcl_DeleteTrace** returns, the caller should never again use the *trace* token.

KEYWORDS

command, create, delete, interpreter, trace

NAME

Tcl_DStringInit, Tcl_DStringAppend, Tcl_DStringAppendElement, Tcl_DStringStartSublist, Tcl_DStringEndSublist, Tcl_DStringLength, Tcl_DStringValue, Tcl_DStringSetLength, Tcl_DStringFree, Tcl_DStringResult, Tcl_DStringGetResult − manipulate dynamic strings

SYNOPSIS

#include <tcl.h>

Tcl_DStringInit(*dsPtr*)

char ∗
Tcl_DStringAppend(*dsPtr, string, length*)

char ∗
Tcl_DStringAppendElement(*dsPtr, string*)

Tcl_DStringStartSublist(*dsPtr*)

Tcl_DStringEndSublist(*dsPtr*)

int
Tcl_DStringLength(*dsPtr*)

char ∗
Tcl_DStringValue(*dsPtr*)

Tcl_DStringSetLength(*dsPtr, newLength*)

Tcl_DStringFree(*dsPtr*)

Tcl_DStringResult(*interp, dsPtr*)

Tcl_DStringGetResult(*interp, dsPtr*)

ARGUMENTS

Tcl_DString	∗*dsPtr*	(in/out)	Pointer to structure that is used to manage a dynamic string.
char	∗*string*	(in)	Pointer to characters to add to dynamic string.
int	*length*	(in)	Number of characters from string to add to dynamic string. If -1, add all characters up to null terminating character.
int	*newLength*	(in)	New length for dynamic string, not including null terminating character.
Tcl_Interp	∗*interp*	(in/out)	Interpreter whose result is to be set from or moved to the dynamic string.

DESCRIPTION

Dynamic strings provide a mechanism for building up arbitrarily long strings by gradually appending information. If the dynamic string is short then there will be no memory allocation overhead; as the string gets larger, additional space will be allocated as needed.

Tcl_DStringInit initializes a dynamic string to zero length. The Tcl_DString structure must have been allocated by the caller. No assumptions are made about the current state of the structure; anything already in it is discarded. If the structure has been used previously, **Tcl_DStringFree** should be called first to free up any memory allocated for the old string.

Tcl_DStringAppend adds new information to a dynamic string, allocating more memory for the string if needed. If *length* is less than zero then everything in *string* is appended to the dynamic string; otherwise *length* specifies the number of bytes to append. **Tcl_DStringAppend** returns a pointer to the characters of the new string. The string can also be retrieved from the *string* field of the Tcl_DString structure.

Tcl_DStringAppendElement is similar to **Tcl_DStringAppend** except that it doesn't take a *length* argument (it appends all of *string*) and it converts the string to a proper list element before appending. **Tcl_DStringAppendElement** adds a separator space before the new list element unless the new list element is the first in a list or sub-list (i.e. either the current string is empty, or it contains the single character "{", or the last two characters of the current string are " {"). **Tcl_DStringAppendElement** returns a pointer to the characters of the new string.

Tcl_DStringStartSublist and **Tcl_DStringEndSublist** can be used to create nested lists. To append a list element that is itself a sublist, first call **Tcl_DStringStartSublist**, then call **Tcl_DStringAppendElement** for each of the elements in the sublist, then call **Tcl_DStringEndSublist** to end the sublist. **Tcl_DStringStartSublist** appends a space character if needed, followed by an open brace; **Tcl_DStringEndSublist** appends a close brace. Lists can be nested to any depth.

Tcl_DStringLength is a macro that returns the current length of a dynamic string (not including the terminating null character). **Tcl_DStringValue** is a macro that returns a pointer to the current contents of a dynamic string.

Tcl_DStringSetLength changes the length of a dynamic string. If *newLength* is less than the string's current length, then the string is truncated. If *newLength* is greater than the string's current length, then the string will become longer and new space will be allocated for the string if needed. However, **Tcl_DStringSetLength** will not initialize the new space except to provide a terminating null character; it is up to the caller to fill in the new space. **Tcl_DStringSetLength** does not free up the string's storage space even if the string is truncated to zero length, so **Tcl_DStringFree** will still need to be called.

Tcl_DStringFree should be called when you're finished using the string. It frees up any memory that was allocated for the string and reinitializes the string's value to an empty string.

Tcl_DStringResult sets the result of *interp* to the value of the dynamic string given by *dsPtr*. It does this by moving a pointer from *dsPtr* to *interp->result*. This saves the cost of allocating new memory and copying the string. **Tcl_DStringResult** also reinitializes the dynamic string to an empty string.

Tcl_DStringGetResult does the opposite of **Tcl_DStringResult**. It sets the value of *dsPtr* to the result of *interp* and it clears *interp*'s result. If possible it does this by moving a pointer rather than by copying the string.

KEYWORDS

append, dynamic string, free, result

NAME

Tcl_DetachPids, Tcl_ReapDetachedProcs − manage child processes in background

SYNOPSIS

#include <tcl.h>

Tcl_DetachPids(*numPids, pidPtr*)

Tcl_ReapDetachedProcs()

ARGUMENTS

int	*numPids*	(in)	Number of process ids contained in the array pointed to by *pidPtr*.
int	∗*pidPtr*	(in)	Address of array containing *numPids* process ids.

DESCRIPTION

Tcl_DetachPids and **Tcl_ReapDetachedProcs** provide a mechanism for managing subprocesses that are running in background. These procedures are needed because the parent of a process must eventually invoke the **waitpid** kernel call (or one of a few other similar kernel calls) to wait for the child to exit. Until the parent waits for the child, the child's state cannot be completely reclaimed by the system. If a parent continually creates children and doesn't wait on them, the system's process table will eventually overflow, even if all the children have exited.

Tcl_DetachPids may be called to ask Tcl to take responsibility for one or more processes whose process ids are contained in the *pidPtr* array passed as argument. The caller presumably has started these processes running in background and doesn't want to have to deal with them again.

Tcl_ReapDetachedProcs invokes the **waitpid** kernel call on each of the background processes so that its state can be cleaned up if it has exited. If the process hasn't exited yet, **Tcl_ReapDetachedProcs** doesn't wait for it to exit; it will check again the next time it is invoked. Tcl automatically calls **Tcl_ReapDetachedProcs** each time the **exec** command is executed, so in most cases it isn't necessary for any code outside of Tcl to invoke **Tcl_ReapDetachedProcs**. However, if you call **Tcl_DetachPids** in situations where the **exec** command may never get executed, you may wish to call **Tcl_ReapDetachedProcs** from time to time so that background processes can be cleaned up.

KEYWORDS

background, child, detach, process, wait

NAME

Tcl_DoOneEvent − wait for events and invoke event handlers

SYNOPSIS

#include <tcl.h>

int
Tcl_DoOneEvent(*flags*)

ARGUMENTS

int	*flags*	(in)	This parameter is normally zero. It may be an OR-ed combination of any of the following flag bits: TCL_WINDOW_EVENTS, TCL_FILE_EVENTS, TCL_TIMER_EVENTS, TCL_IDLE_EVENTS, TCL_ALL_EVENTS, or TCL_DONT_WAIT.

DESCRIPTION

This procedure is the entry point to Tcl's event loop; it is responsible for waiting for events and dispatching event handlers created with procedures such as **Tk_CreateEventHandler**, **Tcl_CreateFileHandler**, **Tcl_CreateTimerHandler**, and **Tcl_DoWhenIdle**. **Tcl_DoOneEvent** checks to see if events are already present on the Tcl event queue; if so, it calls the handler(s) for the first (oldest) event, removes it from the queue, and returns. If there are no events ready to be handled, then **Tcl_DoOneEvent** checks for new events from all possible sources. If any are found, it puts all of them on Tcl's event queue, calls handlers for the first event on the queue, and returns. If no events are found, **Tcl_DoOneEvent** checks for **Tcl_DoWhenIdle** callbacks; if any are found, it invokes all of them and returns. Finally, if no events or idle callbacks have been found, then **Tcl_DoOneEvent** sleeps until an event occurs; then it adds any new events to the Tcl event queue, calls handlers for the first event, and returns. The normal return value is 1 to signify that some event was processed (see below for other alternatives).

If the *flags* argument to **Tcl_DoOneEvent** is non-zero, it restricts the kinds of events that will be processed by **Tcl_DoOneEvent**. *Flags* may be an OR-ed combination of any of the following bits:

TCL_WINDOW_EVENTS −	Process window system events.
TCL_FILE_EVENTS −	Process file events.
TCL_TIMER_EVENTS −	Process timer events.
TCL_IDLE_EVENTS −	Process idle callbacks.
TCL_ALL_EVENTS −	Process all kinds of events: equivalent to OR-ing together all of the above flags or specifying none of them.
TCL_DONT_WAIT −	Don't sleep: process only events that are ready at the time of the call.

If any of the flags **TCL_WINDOW_EVENTS**, **TCL_FILE_EVENTS**, **TCL_TIMER_EVENTS**, or **TCL_IDLE_EVENTS** is set, then the only events that will be considered are those for which flags are set. Setting none of these flags is equivalent to the value **TCL_ALL_EVENTS**, which causes all event types to be processed. If an application has defined additional event sources with **Tcl_CreateEventSource**, then additional *flag* values may also be valid, depending on those event sources.

The **TCL_DONT_WAIT** flag causes **Tcl_DoOneEvent** not to put the process to sleep: it will check for events but if none are found then it returns immediately with a return value of 0 to indicate that no work was done. **Tcl_DoOneEvent** will also return 0 without doing anything if the only alternative is to block forever (this can happen, for example, if *flags* is **TCL_IDLE_EVENTS** and there are no **Tcl_DoWhenIdle** callbacks pending, or if no event handlers or timer handlers exist).

Tcl_DoOneEvent may be invoked recursively. For example, it is possible to invoke **Tcl_DoOneEvent** recursively from a handler called by **Tcl_DoOneEvent**. This sort of operation is useful in some modal situations, such as when a notification dialog has been popped up and an application wishes to wait for the user to click a button in the dialog before doing anything else.

KEYWORDS
callback, event, handler, idle, timer

NAME

Tcl_DoWhenIdle, Tcl_CancelIdleCall − invoke a procedure when there are no pending events

SYNOPSIS

#include <tcl.h>

Tcl_DoWhenIdle(*proc, clientData*)

Tcl_CancelIdleCall(*proc, clientData*)

ARGUMENTS

Tcl_IdleProc	∗*proc*	(in)	Procedure to invoke.
ClientData	*clientData*	(in)	Arbitrary one-word value to pass to *proc*.

DESCRIPTION

Tcl_DoWhenIdle arranges for *proc* to be invoked when the application becomes idle. The application is considered to be idle when **Tcl_DoOneEvent** has been called, couldn't find any events to handle, and is about to go to sleep waiting for an event to occur. At this point all pending **Tcl_DoWhenIdle** handlers are invoked. For each call to **Tcl_DoWhenIdle** there will be a single call to *proc*; after *proc* is invoked the handler is automatically removed. **Tcl_DoWhenIdle** is only usable in programs that use **Tcl_DoOneEvent** to dispatch events.

Proc should have arguments and result that match the type **Tcl_IdleProc**:

```
typedef void Tcl_IdleProc(ClientData clientData);
```

The *clientData* parameter to *proc* is a copy of the *clientData* argument given to **Tcl_DoWhenIdle**. Typically, *clientData* points to a data structure containing application-specific information about what *proc* should do.

Tcl_CancelIdleCall may be used to cancel one or more previous calls to **Tcl_DoWhenIdle**: if there is a **Tcl_DoWhenIdle** handler registered for *proc* and *clientData*, then it is removed without invoking it. If there is more than one handler on the idle list that refers to *proc* and *clientData*, all of the handlers are removed. If no existing handlers match *proc* and *clientData* then nothing happens.

Tcl_DoWhenIdle is most useful in situations where (a) a piece of work will have to be done but (b) it's possible that something will happen in the near future that will change what has to be done or require something different to be done. **Tcl_DoWhenIdle** allows the actual work to be deferred until all pending events have been processed. At this point the exact work to be done will presumably be known and it can be done exactly once.

For example, **Tcl_DoWhenIdle** might be used by an editor to defer display updates until all pending commands have been processed. Without this feature, redundant redisplays might occur in some situations, such as the processing of a command file.

BUGS

At present it is not safe for an idle callback to reschedule itself continuously. This will interact badly with certain features of Tk that attempt to wait for all idle callbacks to complete. If you would like for an idle callback to reschedule itself continuously, it is better to use a timer handler with a zero timeout period.

KEYWORDS

callback, defer, idle callback

NAME

Tcl_NewDoubleObj, Tcl_SetDoubleObj, Tcl_GetDoubleFromObj − manipulate Tcl objects as floating-point values

SYNOPSIS

#include <tcl.h>

Tcl_Obj *
Tcl_NewDoubleObj(*doubleValue*)

Tcl_SetDoubleObj(*objPtr, doubleValue*)

int
Tcl_GetDoubleFromObj(*interp, objPtr, doublePtr*)

ARGUMENTS

double	*doubleValue*	(in)	A double-precision floating point value used to initialize or set a double object.
Tcl_Obj	**objPtr*	(in/out)	For **Tcl_SetDoubleObj**, this points to the object to be converted to double type. For **Tcl_GetDoubleFromObj**, this refers to the object from which to get a double value; if *objPtr* does not already point to a double object, an attempt will be made to convert it to one.
Tcl_Interp	**interp*	(in/out)	If an error occurs during conversion, an error message is left in the interpreter's result object unless *interp* is NULL.
double	**doublePtr*	(out)	Points to place to store the double value obtained from *objPtr*.

DESCRIPTION

These procedures are used to create, modify, and read double Tcl objects from C code. **Tcl_NewDoubleObj** and **Tcl_SetDoubleObj** will create a new object of double type or modify an existing object to have double type. Both of these procedures set the object to have the double-precision floating point value given by *doubleValue*; **Tcl_NewDoubleObj** returns a pointer to a newly created object with reference count zero. Both procedures set the object's type to be double and assign the double value to the object's internal representation *doubleValue* member. **Tcl_SetDoubleObj** invalidates any old string representation and, if the object is not already a double object, frees any old internal representation.

Tcl_GetDoubleFromObj attempts to return a double value from the Tcl object *objPtr*. If the object is not already a double object, it will attempt to convert it to one. If an error occurs during conversion, it returns **TCL_ERROR** and leaves an error message in the interpreter's result object unless *interp* is NULL. Otherwise, it returns **TCL_OK** and stores the double value in the address given by *doublePtr*. If the object is not already a double object, the conversion will free any old internal representation.

SEE ALSO

Tcl_NewObj, Tcl_DecrRefCount, Tcl_IncrRefCount, Tcl_GetObjResult

KEYWORDS

double, double object, double type, internal representation, object, object type, string representation

NAME

Tcl_Eval, Tcl_VarEval, Tcl_EvalFile, Tcl_GlobalEval − execute Tcl commands

SYNOPSIS

#include <tcl.h>

int
Tcl_Eval(*interp, cmd*)

int
Tcl_VarEval(*interp, string, string, ...* **(char** ∗**) NULL**)

int
Tcl_EvalFile(*interp, fileName*)

int
Tcl_GlobalEval(*interp, cmd*)

ARGUMENTS

Tcl_Interp	∗*interp*	(in)	Interpreter in which to execute the command. A string result will be stored in *interp->result*.
char	∗*cmd*	(in)	Command (or sequence of commands) to execute. Must be in writable memory (**Tcl_Eval** makes temporary modifications to the command).
char	∗*string*	(in)	String forming part of Tcl command.
char	∗*fileName*	(in)	Name of file containing Tcl command string.

DESCRIPTION

All four of these procedures execute Tcl commands. **Tcl_Eval** is the core procedure and is used by all the others. It executes the commands in the script held by *cmd* until either an error occurs or it reaches the end of the script.

Note that **Tcl_Eval** and **Tcl_GlobalEval** have been largely replaced by the object-based procedures **Tcl_EvalObj** and **Tcl_GlobalEvalObj**. Those object-based procedures evaluate a script held in a Tcl object instead of a string. The object argument can retain the bytecode instructions for the script and so avoid reparsing the script each time it is executed. **Tcl_Eval** is implemented using **Tcl_EvalObj** but is slower because it must reparse the script each time since there is no object to retain the bytecode instructions.

The return value from **Tcl_Eval** is one of the Tcl return codes **TCL_OK**, **TCL_ERROR**, **TCL_RETURN**, **TCL_BREAK**, or **TCL_CONTINUE**, and *interp->result* will point to a string with additional information (a result value or error message). If an error occurs during compilation, this return information describes the error. Otherwise, this return information corresponds to the last command executed from *cmd*.

Tcl_VarEval takes any number of string arguments of any length, concatenates them into a single string, then calls **Tcl_Eval** to execute that string as a Tcl command. It returns the result of the command and also modifies *interp->result* in the usual fashion for Tcl commands. The last argument to **Tcl_VarEval** must be NULL to indicate the end of arguments.

Tcl_EvalFile reads the file given by *fileName* and evaluates its contents as a Tcl command by calling **Tcl_Eval**. It returns a standard Tcl result that reflects the result of evaluating the file. If the file couldn't be read then a Tcl error is returned to describe why the file couldn't be read.

During the processing of a Tcl command it is legal to make nested calls to evaluate other commands (this is how procedures and some control structures are implemented). If a code other than **TCL_OK** is returned from a nested **Tcl_Eval** invocation, then the caller should normally return immediately, passing that same return code back to its caller, and so on until the top-level application is reached. A few commands, like **for**, will check for certain return codes, like **TCL_BREAK** and **TCL_CONTINUE**, and process them specially without returning.

Tcl_Eval keeps track of how many nested **Tcl_Eval** invocations are in progress for *interp*. If a code of **TCL_RETURN**, **TCL_BREAK**, or **TCL_CONTINUE** is about to be returned from the topmost **Tcl_Eval** invocation for *interp*, it converts the return code to **TCL_ERROR** and sets *interp->result* to point to an error message indicating that the **return**, **break**, or **continue** command was invoked in an inappropriate place. This means that top-level applications should never see a return code from **Tcl_Eval** other then **TCL_OK** or **TCL_ERROR**.

SEE ALSO

Tcl_EvalObj, Tcl_GlobalEvalObj

KEYWORDS

command, execute, file, global, object, object result, variable

NAME

Tcl_EvalObj, Tcl_GlobalEvalObj − execute Tcl commands

SYNOPSIS

#include <tcl.h>

int
Tcl_EvalObj(*interp, objPtr*)

int
Tcl_GlobalEvalObj(*interp, objPtr*)

ARGUMENTS

Tcl_Interp	∗*interp*	(in)	Interpreter in which to execute the command. The command's result will be stored in the interpreter's result object and can be retrieved using **Tcl_GetObjResult**.
Tcl_Obj	∗*objPtr*	(in)	A Tcl object containing a command string (or sequence of commands in a string) to execute.

DESCRIPTION

These two procedures execute Tcl commands. **Tcl_EvalObj** is the core procedure and is used by **Tcl_GlobalEvalObj**. It executes the commands in the script held by *objPtr* until either an error occurs or it reaches the end of the script. If this is the first time *objPtr* has been executed, its commands are compiled into bytecode instructions that are then executed if there are no compilation errors.

The return value from **Tcl_EvalObj** is one of the Tcl return codes **TCL_OK**, **TCL_ERROR**, **TCL_RETURN**, **TCL_BREAK**, or **TCL_CONTINUE**, and a result object containing additional information (a result value or error message) that can be retrieved using **Tcl_GetObjResult**. If an error occurs during compilation, this return information describes the error. Otherwise, this return information corresponds to the last command executed from *objPtr*.

Tcl_GlobalEvalObj is similar to **Tcl_EvalObj** except that it processes the command at global level. This means that the variable context for the command consists of global variables only (it ignores any Tcl procedure that is active). This produces an effect similar to the Tcl command ''**uplevel 0**''.

During the processing of a Tcl command it is legal to make nested calls to evaluate other commands (this is how procedures and some control structures are implemented). If a code other than **TCL_OK** is returned from a nested **Tcl_EvalObj** invocation, then the caller should normally return immediately, passing that same return code back to its caller, and so on until the top-level application is reached. A few commands, like **for**, will check for certain return codes, like **TCL_BREAK** and **TCL_CONTINUE**, and process them specially without returning.

Tcl_EvalObj keeps track of how many nested **Tcl_EvalObj** invocations are in progress for *interp*. If a code of **TCL_RETURN**, **TCL_BREAK**, or **TCL_CONTINUE** is about to be returned from the topmost **Tcl_EvalObj** invocation for *interp*, it converts the return code to **TCL_ERROR** and sets the interpreter's result object to point to an error message indicating that the **return**, **break**, or **continue** command was invoked in an inappropriate place. This means that top-level applications should never see a return code from **Tcl_EvalObj** other then **TCL_OK** or **TCL_ERROR**.

SEE ALSO

Tcl_GetObjResult, Tcl_SetObjResult

KEYWORDS

command, execute, file, global, object, object result, variable

NAME

Tcl_Exit, Tcl_Finalize, Tcl_CreateExitHandler, Tcl_DeleteExitHandler − end the application (and invoke exit handlers)

SYNOPSIS

#include <tcl.h>

Tcl_Exit(*status*)

Tcl_Finalize()

Tcl_CreateExitHandler(*proc, clientData*)

Tcl_DeleteExitHandler(*proc, clientData*)

ARGUMENTS

int	*status*	(in)	Provides information about why application exited. Exact meaning may be platform-specific. 0 usually means a normal exit, any nonzero value usually means that an error occurred.
Tcl_ExitProc	∗*proc*	(in)	Procedure to invoke before exiting application.
ClientData	*clientData*	(in)	Arbitrary one-word value to pass to *proc*.

DESCRIPTION

The procedures described here provide a graceful mechanism to end the execution of a **Tcl** application. Exit handlers are invoked to cleanup the application's state before ending the execution of **Tcl** code.

Invoke **Tcl_Exit** to end a **Tcl** application and to exit from this process. This procedure is invoked by the **exit** command, and can be invoked anyplace else to terminate the application. No-one should ever invoke the **exit** system procedure directly; always invoke **Tcl_Exit** instead, so that it can invoke exit handlers. Note that if other code invokes **exit** system procedure directly, or otherwise causes the application to terminate without calling **Tcl_Exit**, the exit handlers will not be run. **Tcl_Exit** internally invokes the **exit** system call, thus it never returns control to its caller.

Tcl_Finalize is similar to **Tcl_Exit** except that it does not exit from the current process. It is useful for cleaning up when a process is finished using **Tcl** but wishes to continue executing, and when **Tcl** is used in a dynamically loaded extension that is about to be unloaded. On some systems **Tcl** is automatically notified when it is being unloaded, and it calls **Tcl_Finalize** internally; on these systems it not necessary for the caller to explicitly call **Tcl_Finalize**. However, to ensure portability, your code should always invoke **Tcl_Finalize** when **Tcl** is being unloaded, to ensure that the code will work on all platforms. **Tcl_Finalize** can be safely called more than once.

Tcl_CreateExitHandler arranges for *proc* to be invoked by **Tcl_Finalize** and **Tcl_Exit**. This provides a hook for cleanup operations such as flushing buffers and freeing global memory. *Proc* should match the type **Tcl_ExitProc**:

typedef void Tcl_ExitProc(ClientData *clientData*);

The *clientData* parameter to *proc* is a copy of the *clientData* argument given to **Tcl_CreateExitHandler** when the callback was created. Typically, *clientData* points to a data structure containing application-specific information about what to do in *proc*.

Tcl_DeleteExitHandler may be called to delete a previously-created exit handler. It removes the handler indicated by *proc* and *clientData* so that no call to *proc* will be made. If no such handler exists then **Tcl_DeleteExitHandler** does nothing.

Tcl_Finalize and **Tcl_Exit** execute all registered exit handlers, in reverse order from the order in which they were registered. This matches the natural order in which extensions are loaded and unloaded; if extension **A** loads extension **B**, it usually unloads **B** before it itself is unloaded. If extension **A** registers its exit handlers before loading extension **B**, this ensures that any exit handlers for **B** will be executed before the exit handlers for **A**.

KEYWORDS

callback, cleanup, dynamic loading, end application, exit, unloading

NAME

Tcl_ExprLong, Tcl_ExprDouble, Tcl_ExprBoolean, Tcl_ExprString − evaluate an expression

SYNOPSIS

#include <tcl.h>

int
Tcl_ExprLong(*interp, string, longPtr*)

int
Tcl_ExprDouble(*interp, string, doublePtr*)

int
Tcl_ExprBoolean(*interp, string, booleanPtr*)

int
Tcl_ExprString(*interp, string*)

ARGUMENTS

Tcl_Interp	∗*interp*	(in)	Interpreter in whose context to evaluate *string* or *objPtr*.
char	∗*string*	(in)	Expression to be evaluated. Must be in writable memory (the expression parser makes temporary modifications to the string during parsing, which it undoes before returning).
long	∗*longPtr*	(out)	Pointer to location in which to store the integer value of the expression.
int	∗*doublePtr*	(out)	Pointer to location in which to store the floating-point value of the expression.
int	∗*booleanPtr*	(out)	Pointer to location in which to store the 0/1 boolean value of the expression.

DESCRIPTION

These four procedures all evaluate the expression given by the *string* argument and return the result in one of four different forms. The expression can have any of the forms accepted by the **expr** command. Note that these procedures have been largely replaced by the object-based procedures **Tcl_ExprLongObj**, **Tcl_ExprDoubleObj**, **Tcl_ExprBooleanObj**, and **Tcl_ExprStringObj**. Those object-based procedures evaluate an expression held in a Tcl object instead of a string. The object argument can retain an internal representation that is more efficient to execute.

The *interp* argument refers to an interpreter used to evaluate the expression (e.g. for variables and nested Tcl commands) and to return error information. *interp->result* is assumed to be initialized in the standard fashion when they are invoked.

For all of these procedures the return value is a standard Tcl result: **TCL_OK** means the expression was successfully evaluated, and **TCL_ERROR** means that an error occurred while evaluating the expression. If **TCL_ERROR** is returned then *interp->result* will hold a message describing the error. If an error occurs while executing a Tcl command embedded in the expression then that error will be returned.

If the expression is successfully evaluated, then its value is returned in one of four forms, depending on which procedure is invoked. **Tcl_ExprLong** stores an integer value at ∗*longPtr*. If the expression's actual value is a floating-point number, then it is truncated to an integer. If the expression's actual value is a non-numeric string then an error is returned.

Tcl_ExprDouble stores a floating-point value at **doublePtr*. If the expression's actual value is an integer, it is converted to floating-point. If the expression's actual value is a non-numeric string then an error is returned.

Tcl_ExprBoolean stores a 0/1 integer value at **booleanPtr*. If the expression's actual value is an integer or floating-point number, then they store 0 at **booleanPtr* if the value was zero and 1 otherwise. If the expression's actual value is a non-numeric string then it must be one of the values accepted by **Tcl_GetBoolean** such as ''yes'' or ''no'', or else an error occurs.

Tcl_ExprString returns the value of the expression as a string stored in *interp->result*. If the expression's actual value is an integer then **Tcl_ExprString** converts it to a string using **sprintf** with a ''%d'' converter. If the expression's actual value is a floating-point number, then **Tcl_ExprString** calls **Tcl_PrintDouble** to convert it to a string.

SEE ALSO

Tcl_ExprLongObj, Tcl_ExprDoubleObj, Tcl_ExprBooleanObj, Tcl_ExprObj

KEYWORDS

boolean, double, evaluate, expression, integer, object, string

NAME

Tcl_ExprLongObj, Tcl_ExprDoubleObj, Tcl_ExprBooleanObj, Tcl_ExprObj − evaluate an expression

SYNOPSIS

#include <tcl.h>

int
Tcl_ExprLongObj(*interp, objPtr, longPtr*)

int
Tcl_ExprDoubleObj(*interp, objPtr, doublePtr*)

int
Tcl_ExprBooleanObj(*interp, objPtr, booleanPtr*)

int
Tcl_ExprObj(*interp, objPtr, resultPtrPtr*)

ARGUMENTS

Tcl_Interp	∗*interp*	(in)	Interpreter in whose context to evaluate *string* or *objPtr*.
Tcl_Obj	∗*objPtr*	(in)	Pointer to an object containing the expression to evaluate.
long	∗*longPtr*	(out)	Pointer to location in which to store the integer value of the expression.
int	∗*doublePtr*	(out)	Pointer to location in which to store the floating-point value of the expression.
int	∗*booleanPtr*	(out)	Pointer to location in which to store the 0/1 boolean value of the expression.
Tcl_Obj	∗*resultPtrPtr*	(out)	Pointer to location in which to store a pointer to the object that is the result of the expression.

DESCRIPTION

These four procedures all evaluate an expression, returning the result in one of four different forms. The expression is given by the *objPtr* argument, and it can have any of the forms accepted by the **expr** command.

The *interp* argument refers to an interpreter used to evaluate the expression (e.g. for variables and nested Tcl commands) and to return error information.

For all of these procedures the return value is a standard Tcl result: **TCL_OK** means the expression was successfully evaluated, and **TCL_ERROR** means that an error occurred while evaluating the expression. If **TCL_ERROR** is returned, then a message describing the error can be retrieved using **Tcl_GetObjResult**. If an error occurs while executing a Tcl command embedded in the expression then that error will be returned.

If the expression is successfully evaluated, then its value is returned in one of four forms, depending on which procedure is invoked. **Tcl_ExprLongObj** stores an integer value at ∗*longPtr*. If the expression's actual value is a floating-point number, then it is truncated to an integer. If the expression's actual value is a non-numeric string then an error is returned.

Tcl_ExprDoubleObj stores a floating-point value at ∗*doublePtr*. If the expression's actual value is an integer, it is converted to floating-point. If the expression's actual value is a non-numeric string then an error is returned.

Tcl_ExprBooleanObj stores a 0/1 integer value at ∗*booleanPtr*. If the expression's actual value is an integer or floating-point number, then they store 0 at ∗*booleanPtr* if the value was zero and 1 otherwise. If the expression's actual value is a non-numeric string then it must be one of the values accepted by **Tcl_GetBoolean** such as ''yes'' or ''no'', or else an error occurs.

If **Tcl_ExprObj** successfully evaluates the expression, it stores a pointer to the Tcl object containing the expression's value at ∗*resultPtrPtr*. In this case, the caller is responsible for calling **Tcl_DecrRefCount** to decrement the object's reference count when it is finished with the object.

SEE ALSO

Tcl_ExprLong, Tcl_ExprDouble, Tcl_ExprBoolean, Tcl_ExprString, Tcl_GetObjResult

KEYWORDS

boolean, double, evaluate, expression, integer, object, string

NAME

Tcl_FindExecutable, Tcl_GetNameOfExecutable − identify or return the name of the binary file containing the application

SYNOPSIS

#include <tcl.h>

char ∗
Tcl_FindExecutable(*argv0*)

CONST char ∗
Tcl_GetNameOfExecutable()

ARGUMENTS

char	∗*argv0*	(in)	The first command-line argument to the program, which gives the application's name.

DESCRIPTION

The **Tcl_FindExecutable** procedure computes the full path name of the executable file from which the application was invoked and saves it for Tcl's internal use. The executable's path name is needed for several purposes in Tcl. For example, it is needed on some platforms in the implementation of the **load** command. It is also returned by the **info nameofexecutable** command.

On UNIX platforms this procedure is typically invoked as the very first thing in the application's main program; it must be passed *argv[0]* as its argument. **Tcl_FindExecutable** uses *argv0* along with the **PATH** environment variable to find the application's executable, if possible. If it fails to find the binary, then future calls to **info nameofexecutable** will return an empty string.

Tcl_GetNameOfExecutable simply returns a pointer to the internal full path name of the executable file as computed by **Tcl_FindExecutable**. This procedure call is the C API equivalent to the **info nameofexecutable** command. NULL is returned if the internal full path name has not been computed or unknown.

KEYWORDS

binary, executable file

NAME

Tcl_GetIndexFromObj − lookup string in table of keywords

SYNOPSIS

#include <tcl.h>

int
Tcl_GetIndexFromObj(*interp, objPtr, tablePtr, msg, flags, indexPtr*)

ARGUMENTS

Tcl_Interp	∗*interp*	(in)	Interpreter to use for error reporting; if NULL, then no message is provided on errors.
Tcl_Obj	∗*objPtr*	(in/out)	The string value of this object is used to search through *tablePtr*. The internal representation is modified to hold the index of the matching table entry.
char	∗∗*tablePtr*	(in)	An array of null-terminated strings. The end of the array is marked by a NULL string pointer.
char	∗*msg*	(in)	Null-terminated string describing what is being looked up, such as **option**. This string is included in error messages.
int	*flags*	(in)	OR-ed combination of bits providing additional information for operation. The only bit that is currently defined is **TCL_EXACT**.
int	∗*indexPtr*	(out)	The index of the string in *tablePtr* that matches the value of *objPtr* is returned here.

DESCRIPTION

This procedure provides an efficient way for looking up keywords, switch names, option names, and similar things where the value of an object must be one of a predefined set of values. *ObjPtr* is compared against each of the strings in *tablePtr* to find a match. A match occurs if *objPtr*'s string value is identical to one of the strings in *tablePtr*, or if it is a unique abbreviation for exactly one of the strings in *tablePtr* and the **TCL_EXACT** flag was not specified; in either case the index of the matching entry is stored at ∗*indexPtr* and TCL_OK is returned.

If there is no matching entry, TCL_ERROR is returned and an error message is left in *interp*'s result if *interp* isn't NULL. *Msg* is included in the error message to indicate what was being looked up. For example, if *msg* is **option** the error message will have a form like **bad option "firt": must be first, second, or third**.

If **Tcl_GetIndexFromObj** completes successfully it modifies the internal representation of *objPtr* to hold the address of the table and the index of the matching entry. If **Tcl_GetIndexFromObj** is invoked again with the same *objPtr* and *tablePtr* arguments (e.g. during a reinvocation of a Tcl command), it returns the matching index immediately without having to redo the lookup operation. Note: **Tcl_GetIndexFromObj** assumes that the entries in *tablePtr* are static: they must not change between invocations.

SEE ALSO

Tcl_WrongNumArgs

KEYWORDS

index, object, table lookup

NAME

Tcl_GetInt, Tcl_GetDouble, Tcl_GetBoolean − convert from string to integer, double, or boolean

SYNOPSIS

#include <tcl.h>

int
Tcl_GetInt(*interp, string, intPtr*)

int
Tcl_GetDouble(*interp, string, doublePtr*)

int
Tcl_GetBoolean(*interp, string, boolPtr*)

ARGUMENTS

Tcl_Interp	∗*interp*	(in)	Interpreter to use for error reporting.
char	∗*string*	(in)	Textual value to be converted.
int	∗*intPtr*	(out)	Points to place to store integer value converted from *string*.
double	∗*doublePtr*	(out)	Points to place to store double-precision floating-point value converted from *string*.
int	∗*boolPtr*	(out)	Points to place to store boolean value (0 or 1) converted from *string*.

DESCRIPTION

These procedures convert from strings to integers or double-precision floating-point values or booleans (represented as 0- or 1-valued integers). Each of the procedures takes a *string* argument, converts it to an internal form of a particular type, and stores the converted value at the location indicated by the procedure's third argument. If all goes well, each of the procedures returns TCL_OK. If *string* doesn't have the proper syntax for the desired type then TCL_ERROR is returned, an error message is left in *interp->result*, and nothing is stored at ∗*intPtr* or ∗*doublePtr* or ∗*boolPtr*.

Tcl_GetInt expects *string* to consist of a collection of integer digits, optionally signed and optionally preceded by white space. If the first two characters of *string* are ''0x'' then *string* is expected to be in hexadecimal form; otherwise, if the first character of *string* is ''0'' then *string* is expected to be in octal form; otherwise, *string* is expected to be in decimal form.

Tcl_GetDouble expects *string* to consist of a floating-point number, which is: white space; a sign; a sequence of digits; a decimal point; a sequence of digits; the letter ''e''; and a signed decimal exponent. Any of the fields may be omitted, except that the digits either before or after the decimal point must be present and if the ''e'' is present then it must be followed by the exponent number.

Tcl_GetBoolean expects *string* to specify a boolean value. If *string* is any of **0**, **false**, **no**, or **off**, then **Tcl_GetBoolean** stores a zero value at ∗*boolPtr*. If *string* is any of **1**, **true**, **yes**, or **on**, then 1 is stored at ∗*boolPtr*. Any of these values may be abbreviated, and upper-case spellings are also acceptable.

KEYWORDS

boolean, conversion, double, floating-point, integer

NAME

Tcl_GetOpenFile − Get a standard IO File ∗ handle from a channel. (Unix only)

SYNOPSIS

#include <tcl.h>

int
Tcl_GetOpenFile(*interp, string, write, checkUsage, filePtr*)

ARGUMENTS

Tcl_Interp	∗*interp*	(in)	Tcl interpreter from which file handle is to be obtained.
char	∗*string*	(in)	String identifying channel, such as **stdin** or **file4**.
int	*write*	(in)	Non-zero means the file will be used for writing, zero means it will be used for reading.
int	*checkUsage*	(in)	If non-zero, then an error will be generated if the file wasn't opened for the access indicated by *write*.
ClientData	∗*filePtr*	(out)	Points to word in which to store pointer to FILE structure for the file given by *string*.

DESCRIPTION

Tcl_GetOpenFile takes as argument a file identifier of the form returned by the **open** command and returns at ∗*filePtr* a pointer to the FILE structure for the file. The *write* argument indicates whether the FILE pointer will be used for reading or writing. In some cases, such as a channel that connects to a pipeline of subprocesses, different FILE pointers will be returned for reading and writing. **Tcl_GetOpenFile** normally returns TCL_OK. If an error occurs in **Tcl_GetOpenFile** (e.g. *string* didn't make any sense or *checkUsage* was set and the file wasn't opened for the access specified by *write*) then TCL_ERROR is returned and *interp->result* will contain an error message. In the current implementation *checkUsage* is ignored and consistency checks are always performed.

Note that this interface is only supported on the Unix platform.

KEYWORDS

channel, file handle, permissions, pipeline, read, write

NAME

Tcl_GetStdChannel, Tcl_SetStdChannel − procedures for retrieving and replacing the standard channels

SYNOPSIS

#include <tcl.h>

Tcl_Channel
Tcl_GetStdChannel(*type*)

Tcl_SetStdChannel(*channel*, *type*)

ARGUMENTS

int	*type*	(in)	The identifier for the standard channel to retrieve or modify. Must be one of **TCL_STDIN**, **TCL_STDOUT**, or **TCL_STDERR**.
Tcl_Channel	*channel*	(in)	The channel to use as the new value for the specified standard channel.

DESCRIPTION

Tcl defines three special channels that are used by various I/O related commands if no other channels are specified. The standard input channel has a channel name of **stdin** and is used by **read** and **gets**. The standard output channel is named **stdout** and is used by **puts**. The standard error channel is named **stderr** and is used for reporting errors. In addition, the standard channels are inherited by any child processes created using **exec** or **open** in the absence of any other redirections.

The standard channels are actually aliases for other normal channels. The current channel associated with a standard channel can be retrieved by calling **Tcl_GetStdChannel** with one of **TCL_STDIN**, **TCL_STDOUT**, or **TCL_STDERR** as the *type*. The return value will be a valid channel, or NULL.

A new channel can be set for the standard channel specified by *type* by calling **Tcl_SetStdChannel** with a new channel or NULL in the *channel* argument. If the specified channel is closed by a later call to **Tcl_Close**, then the corresponding standard channel will automatically be set to NULL.

If **Tcl_GetStdChannel** is called before **Tcl_SetStdChannel**, Tcl will construct a new channel to wrap the appropriate platform-specific standard file handle. If **Tcl_SetStdChannel** is called before **Tcl_GetStdChannel**, then the default channel will not be created.

If one of the standard channels is set to NULL, either by calling **Tcl_SetStdChannel** with a null *channel* argument, or by calling **Tcl_Close** on the channel, then the next call to **Tcl_CreateChannel** will automatically set the standard channel with the newly created channel. If more than one standard channel is NULL, then the standard channels will be assigned starting with standard input, followed by standard output, with standard error being last.

SEE ALSO

Tcl_Close(3), Tcl_CreateChannel(3)

KEYWORDS

standard channel, standard input, standard output, standard error

NAME

Tcl_InitHashTable, Tcl_DeleteHashTable, Tcl_CreateHashEntry, Tcl_DeleteHashEntry, Tcl_FindHashEntry, Tcl_GetHashValue, Tcl_SetHashValue, Tcl_GetHashKey, Tcl_FirstHashEntry, Tcl_NextHashEntry, Tcl_HashStats − procedures to manage hash tables

SYNOPSIS

#include <tcl.h>

Tcl_InitHashTable(*tablePtr, keyType*)

Tcl_DeleteHashTable(*tablePtr*)

Tcl_HashEntry ∗
Tcl_CreateHashEntry(*tablePtr, key, newPtr*)

Tcl_DeleteHashEntry(*entryPtr*)

Tcl_HashEntry ∗
Tcl_FindHashEntry(*tablePtr, key*)

ClientData
Tcl_GetHashValue(*entryPtr*)

Tcl_SetHashValue(*entryPtr, value*)

char ∗
Tcl_GetHashKey(*tablePtr, entryPtr*)

Tcl_HashEntry ∗
Tcl_FirstHashEntry(*tablePtr, searchPtr*)

Tcl_HashEntry ∗
Tcl_NextHashEntry(*searchPtr*)

char ∗
Tcl_HashStats(*tablePtr*)

ARGUMENTS

Tcl_HashTable	∗*tablePtr*	(in)	Address of hash table structure (for all procedures but **Tcl_InitHashTable**, this must have been initialized by previous call to **Tcl_InitHashTable**).
int	*keyType*	(in)	Kind of keys to use for new hash table. Must be either TCL_STRING_KEYS, TCL_ONE_WORD_KEYS, or an integer value greater than 1.
char	∗*key*	(in)	Key to use for probe into table. Exact form depends on *keyType* used to create table.
int	∗*newPtr*	(out)	The word at ∗*newPtr* is set to 1 if a new entry was created and 0 if there was already an entry for *key*.
Tcl_HashEntry	∗*entryPtr*	(in)	Pointer to hash table entry.
ClientData	*value*	(in)	New value to assign to hash table entry. Need not have type ClientData, but must fit in same space as ClientData.

Tcl_HashSearch	∗*searchPtr*	(in)	Pointer to record to use to keep track of progress in enumerating all the entries in a hash table.

DESCRIPTION

A hash table consists of zero or more entries, each consisting of a key and a value. Given the key for an entry, the hashing routines can very quickly locate the entry, and hence its value. There may be at most one entry in a hash table with a particular key, but many entries may have the same value. Keys can take one of three forms: strings, one-word values, or integer arrays. All of the keys in a given table have the same form, which is specified when the table is initialized.

The value of a hash table entry can be anything that fits in the same space as a ''char ∗'' pointer. Values for hash table entries are managed entirely by clients, not by the hash module itself. Typically each entry's value is a pointer to a data structure managed by client code.

Hash tables grow gracefully as the number of entries increases, so that there are always less than three entries per hash bucket, on average. This allows for fast lookups regardless of the number of entries in a table.

Tcl_InitHashTable initializes a structure that describes a new hash table. The space for the structure is provided by the caller, not by the hash module. The value of *keyType* indicates what kinds of keys will be used for all entries in the table. *KeyType* must have one of the following values:

TCL_STRING_KEYS Keys are null-terminated ASCII strings. They are passed to hashing routines using the address of the first character of the string.

TCL_ONE_WORD_KEYS Keys are single-word values; they are passed to hashing routines and stored in hash table entries as ''char ∗'' values. The pointer value is the key; it need not (and usually doesn't) actually point to a string.

other If *keyType* is not TCL_STRING_KEYS or TCL_ONE_WORD_KEYS, then it must be an integer value greater than 1. In this case the keys will be arrays of ''int'' values, where *keyType* gives the number of ints in each key. This allows structures to be used as keys. All keys must have the same size. Array keys are passed into hashing functions using the address of the first int in the array.

Tcl_DeleteHashTable deletes all of the entries in a hash table and frees up the memory associated with the table's bucket array and entries. It does not free the actual table structure (pointed to by *tablePtr*), since that memory is assumed to be managed by the client. **Tcl_DeleteHashTable** also does not free or otherwise manipulate the values of the hash table entries. If the entry values point to dynamically-allocated memory, then it is the client's responsibility to free these structures before deleting the table.

Tcl_CreateHashEntry locates the entry corresponding to a particular key, creating a new entry in the table if there wasn't already one with the given key. If an entry already existed with the given key then ∗*newPtr* is set to zero. If a new entry was created, then ∗*newPtr* is set to a non-zero value and the value of the new entry will be set to zero. The return value from **Tcl_CreateHashEntry** is a pointer to the entry, which may be used to retrieve and modify the entry's value or to delete the entry from the table.

Tcl_DeleteHashEntry will remove an existing entry from a table. The memory associated with the entry itself will be freed, but the client is responsible for any cleanup associated with the entry's value, such as freeing a structure that it points to.

Tcl_FindHashEntry is similar to **Tcl_CreateHashEntry** except that it doesn't create a new entry if the key doesn't exist; instead, it returns NULL as result.

Tcl_GetHashValue and **Tcl_SetHashValue** are used to read and write an entry's value, respectively. Values are stored and retrieved as type ''ClientData'', which is large enough to hold a pointer value. On almost all machines this is large enough to hold an integer value too.

Tcl_GetHashKey returns the key for a given hash table entry, either as a pointer to a string, a one-word (''char *'') key, or as a pointer to the first word of an array of integers, depending on the *keyType* used to create a hash table. In all cases **Tcl_GetHashKey** returns a result with type ''char *''. When the key is a string or array, the result of **Tcl_GetHashKey** points to information in the table entry; this information will remain valid until the entry is deleted or its table is deleted.

Tcl_FirstHashEntry and **Tcl_NextHashEntry** may be used to scan all of the entries in a hash table. A structure of type ''Tcl_HashSearch'', provided by the client, is used to keep track of progress through the table. **Tcl_FirstHashEntry** initializes the search record and returns the first entry in the table (or NULL if the table is empty). Each subsequent call to **Tcl_NextHashEntry** returns the next entry in the table or NULL if the end of the table has been reached. A call to **Tcl_FirstHashEntry** followed by calls to **Tcl_NextHashEntry** will return each of the entries in the table exactly once, in an arbitrary order. It is unadvisable to modify the structure of the table, e.g. by creating or deleting entries, while the search is in progress.

Tcl_HashStats returns a dynamically-allocated string with overall information about a hash table, such as the number of entries it contains, the number of buckets in its hash array, and the utilization of the buckets. It is the caller's responsibility to free the result string by passing it to **free**.

The header file **tcl.h** defines the actual data structures used to implement hash tables. This is necessary so that clients can allocate Tcl_HashTable structures and so that macros can be used to read and write the values of entries. However, users of the hashing routines should never refer directly to any of the fields of any of the hash-related data structures; use the procedures and macros defined here.

KEYWORDS

hash table, key, lookup, search, value

NAME

Tcl_NewIntObj, Tcl_NewLongObj, Tcl_SetIntObj, Tcl_SetLongObj, Tcl_GetIntFromObj, Tcl_GetLongFromObj − manipulate Tcl objects as integers

SYNOPSIS

#include <tcl.h>

Tcl_Obj *
Tcl_NewIntObj(*intValue*)

Tcl_Obj *
Tcl_NewLongObj(*longValue*)

Tcl_SetIntObj(*objPtr, intValue*)

Tcl_SetLongObj(*objPtr, longValue*)

int
Tcl_GetIntFromObj(*interp, objPtr, intPtr*)

int
Tcl_GetLongFromObj(*interp, objPtr, longPtr*)

ARGUMENTS

int	*intValue*	(in)	Integer value used to initialize or set an integer object.
long	*longValue*	(in)	Long integer value used to initialize or set an integer object.
Tcl_Obj	**objPtr*	(in/out)	For **Tcl_SetIntObj** and **Tcl_SetLongObj**, this points to the object to be converted to integer type. For **Tcl_GetIntFromObj** and **Tcl_GetLongFromObj**, this refers to the object from which to get an integer or long integer value; if *objPtr* does not already point to an integer object, an attempt will be made to convert it to one.
Tcl_Interp	**interp*	(in/out)	If an error occurs during conversion, an error message is left in the interpreter's result object unless *interp* is NULL.
int	**intPtr*	(out)	Points to place to store the integer value obtained by **Tcl_GetIntFromObj** from *objPtr*.
long	**longPtr*	(out)	Points to place to store the long integer value obtained by **Tcl_GetLongFromObj** from *objPtr*.

DESCRIPTION

These procedures are used to create, modify, and read integer Tcl objects from C code. **Tcl_NewIntObj**, **Tcl_NewLongObj**, **Tcl_SetIntObj**, and **Tcl_SetLongObj** create a new object of integer type or modify an existing object to have integer type. **Tcl_NewIntObj** and **Tcl_SetIntObj** set the object to have the integer value given by *intValue*, while **Tcl_NewLongObj** and **Tcl_SetLongObj** set the object to have the long integer value given by *longValue*. **Tcl_NewIntObj** and **Tcl_NewLongObj** return a pointer to a newly created object with reference count zero. These procedures set the object's type to be integer and assign the integer value to the object's internal representation *longValue* member. **Tcl_SetIntObj** and **Tcl_SetLongObj** invalidate any old string representation and, if the object is not already an integer object, free any old internal representation.

Tcl_GetIntFromObj and **Tcl_GetLongFromObj** attempt to return an integer value from the Tcl object *objPtr*. If the object is not already an integer object, they will attempt to convert it to one. If an error occurs during conversion, they return **TCL_ERROR** and leave an error message in the interpreter's result object unless *interp* is NULL. Also, if the long integer held in the object's internal representation *longValue* member can not be represented in a (non-long) integer, **Tcl_GetIntFromObj** returns **TCL_ERROR** and leaves an error message in the interpreter's result object unless *interp* is NULL. Otherwise, both procedures return **TCL_OK** and store the integer or the long integer value in the address given by *intPtr* and *longPtr* respectively. If the object is not already an integer object, the conversion will free any old internal representation.

SEE ALSO

Tcl_NewObj, Tcl_DecrRefCount, Tcl_IncrRefCount, Tcl_GetObjResult

KEYWORDS

integer, integer object, integer type, internal representation, object, object type, string representation

NAME

Tcl_Interp − client-visible fields of interpreter structures

SYNOPSIS

```
#include <tcl.h>

typedef struct {
        char *result;
        Tcl_FreeProc *freeProc;
        int errorLine;
} Tcl_Interp;

typedef void Tcl_FreeProc(char *blockPtr);
```

DESCRIPTION

The **Tcl_CreateInterp** procedure returns a pointer to a Tcl_Interp structure. This pointer is then passed into other Tcl procedures to process commands in the interpreter and perform other operations on the interpreter. Interpreter structures contain many many fields that are used by Tcl, but only three that may be accessed by clients: *result*, *freeProc*, and *errorLine*.

The *result* and *freeProc* fields are used to return results or error messages from commands. This information is returned by command procedures back to **Tcl_Eval**, and by **Tcl_Eval** back to its callers. The *result* field points to the string that represents the result or error message, and the *freeProc* field tells how to dispose of the storage for the string when it isn't needed anymore. The easiest way for command procedures to manipulate these fields is to call procedures like **Tcl_SetResult** or **Tcl_AppendResult**; they will hide all the details of managing the fields. The description below is for those procedures that manipulate the fields directly.

Whenever a command procedure returns, it must ensure that the *result* field of its interpreter points to the string being returned by the command. The *result* field must always point to a valid string. If a command wishes to return no result then *interp->result* should point to an empty string. Normally, results are assumed to be statically allocated, which means that the contents will not change before the next time **Tcl_Eval** is called or some other command procedure is invoked. In this case, the *freeProc* field must be zero. Alternatively, a command procedure may dynamically allocate its return value (e.g. using **Tcl_Alloc**) and store a pointer to it in *interp->result*. In this case, the command procedure must also set *interp->freeProc* to the address of a procedure that can free the value, or **TCL_DYNAMIC** if the storage was allocated directly by Tcl or by a call to **Tcl_Alloc**. If *interp->freeProc* is non-zero, then Tcl will call *freeProc* to free the space pointed to by *interp->result* before it invokes the next command. If a client procedure overwrites *interp->result* when *interp->freeProc* is non-zero, then it is responsible for calling *freeProc* to free the old *interp->result* (the **Tcl_FreeResult** macro should be used for this purpose).

FreeProc should have arguments and result that match the **Tcl_FreeProc** declaration above: it receives a single argument which is a pointer to the result value to free. In most applications **TCL_DYNAMIC** is the only non-zero value ever used for *freeProc*. However, an application may store a different procedure address in *freeProc* in order to use an alternate memory allocator or in order to do other cleanup when the result memory is freed.

As part of processing each command, **Tcl_Eval** initializes *interp->result* and *interp->freeProc* just before calling the command procedure for the command. The *freeProc* field will be initialized to zero, and *interp->result* will point to an empty string. Commands that do not return any value can simply leave the fields alone. Furthermore, the empty string pointed to by *result* is actually part of an array of **TCL_RESULT_SIZE** characters (approximately 200). If a command wishes to return a short string, it can simply copy it to the area pointed to by *interp->result*. Or, it can use the sprintf procedure to generate a

short result string at the location pointed to by *interp->result*.

It is a general convention in Tcl-based applications that the result of an interpreter is normally in the initialized state described in the previous paragraph. Procedures that manipulate an interpreter's result (e.g. by returning an error) will generally assume that the result has been initialized when the procedure is called. If such a procedure is to be called after the result has been changed, then **Tcl_ResetResult** should be called first to reset the result to its initialized state.

The *errorLine* field is valid only after **Tcl_Eval** returns a **TCL_ERROR** return code. In this situation the *errorLine* field identifies the line number of the command being executed when the error occurred. The line numbers are relative to the command being executed: 1 means the first line of the command passed to **Tcl_Eval**, 2 means the second line, and so on. The *errorLine* field is typically used in conjunction with **Tcl_AddErrorInfo** to report information about where an error occurred. *ErrorLine* should not normally be modified except by **Tcl_Eval**.

KEYWORDS

free, initialized, interpreter, malloc, result

NAME

Tcl_LinkVar, Tcl_UnlinkVar, Tcl_UpdateLinkedVar − link Tcl variable to C variable

SYNOPSIS

#include <tcl.h>

int
Tcl_LinkVar(*interp, varName, addr, type*)

Tcl_UnlinkVar(*interp, varName*)

Tcl_UpdateLinkedVar(*interp, varName*)

ARGUMENTS

Tcl_Interp	∗*interp*	(in)	Interpreter that contains *varName*. Also used by **Tcl_LinkVar** to return error messages.
char	∗*varName*	(in)	Name of global variable. Must be in writable memory: Tcl may make temporary modifications to it while parsing the variable name.
char	∗*addr*	(in)	Address of C variable that is to be linked to *varName*.
int	*type*	(in)	Type of C variable. Must be one of TCL_LINK_INT, TCL_LINK_DOUBLE, TCL_LINK_BOOLEAN, or TCL_LINK_STRING, optionally OR'ed with TCL_LINK_READ_ONLY to make Tcl variable read-only.

DESCRIPTION

Tcl_LinkVar uses variable traces to keep the Tcl variable named by *varName* in sync with the C variable at the address given by *addr*. Whenever the Tcl variable is read the value of the C variable will be returned, and whenever the Tcl variable is written the C variable will be updated to have the same value. **Tcl_LinkVar** normally returns TCL_OK; if an error occurs while setting up the link (e.g. because *varName* is the name of array) then TCL_ERROR is returned and *interp->result* contains an error message.

The *type* argument specifies the type of the C variable, and must have one of the following values, optionally OR'ed with TCL_LINK_READ_ONLY:

TCL_LINK_INT

The C variable is of type **int**. Any value written into the Tcl variable must have a proper integer form acceptable to **Tcl_GetInt**; attempts to write non-integer values into *varName* will be rejected with Tcl errors.

TCL_LINK_DOUBLE

The C variable is of type **double**. Any value written into the Tcl variable must have a proper real form acceptable to **Tcl_GetDouble**; attempts to write non-real values into *varName* will be rejected with Tcl errors.

TCL_LINK_BOOLEAN

The C variable is of type **int**. If its value is zero then it will read from Tcl as ''0''; otherwise it will read from Tcl as ''1''. Whenever *varName* is modified, the C variable will be set to a 0 or 1 value. Any value written into the Tcl variable must have a proper boolean form acceptable to **Tcl_GetBoolean**; attempts to write non-boolean values into *varName* will be rejected with Tcl errors.

TCL_LINK_STRING

The C variable is of type **char** ∗. If its value is not null then it must be a pointer to a string allocated with **Tcl_Alloc**. Whenever the Tcl variable is modified the current C string will be freed and

new memory will be allocated to hold a copy of the variable's new value. If the C variable contains a null pointer then the Tcl variable will read as ''NULL''.

If the TCL_LINK_READ_ONLY flag is present in *type* then the variable will be read-only from Tcl, so that its value can only be changed by modifying the C variable. Attempts to write the variable from Tcl will be rejected with errors.

Tcl_UnlinkVar removes the link previously set up for the variable given by *varName*. If there does not exist a link for *varName* then the procedure has no effect.

Tcl_UpdateLinkedVar may be invoked after the C variable has changed to force the Tcl variable to be updated immediately. In many cases this procedure is not needed, since any attempt to read the Tcl variable will return the latest value of the C variable. However, if a trace has been set on the Tcl variable (such as a Tk widget that wishes to display the value of the variable), the trace will not trigger when the C variable has changed. **Tcl_UpdateLinkedVar** ensures that any traces on the Tcl variable are invoked.

KEYWORDS

boolean, integer, link, read-only, real, string, traces, variable

NAME

Tcl_ListObjAppendList, Tcl_ListObjAppendElement, Tcl_NewListObj, Tcl_SetListObj, Tcl_ListObjGetElements, Tcl_ListObjLength, Tcl_ListObjIndex, Tcl_ListObjReplace − manipulate Tcl objects as lists

SYNOPSIS

#include <tcl.h>

int
Tcl_ListObjAppendList(*interp, listPtr, elemListPtr*)

int
Tcl_ListObjAppendElement(*interp, listPtr, objPtr*)

Tcl_Obj *
Tcl_NewListObj(*objc, objv*)

Tcl_SetListObj(*objPtr, objc, objv*)

int
Tcl_ListObjGetElements(*interp, listPtr, objcPtr, objvPtr*)

int
Tcl_ListObjLength(*interp, listPtr, intPtr*)

int
Tcl_ListObjIndex(*interp, listPtr, index, objPtrPtr*)

int
Tcl_ListObjReplace(*interp, listPtr, first, count, objc, objv*)

ARGUMENTS

Tcl_Interp	**interp*	(in)	If an error occurs while converting an object to be a list object, an error message is left in the interpreter's result object unless *interp* is NULL.
Tcl_Obj	**listPtr*	(in/out)	Points to the list object to be manipulated. If *listPtr* does not already point to a list object, an attempt will be made to convert it to one.
Tcl_Obj	**elemListPtr*	(in/out)	For **Tcl_ListObjAppendList**, this points to a list object containing elements to be appended onto *listPtr*. Each element of **elemListPtr* will become a new element of *listPtr*. If **elemListPtr* is not NULL and does not already point to a list object, an attempt will be made to convert it to one.
Tcl_Obj	**objPtr*	(in)	For **Tcl_ListObjAppendElement**, points to the Tcl object that will be appended to *listPtr*. For **Tcl_SetListObj**, this points to the Tcl object that will be converted to a list object containing the *objc* elements of the array referenced by *objv*.
int	**objcPtr*	(in)	Points to location where **Tcl_ListObjGetElements** stores the number of element objects in *listPtr*.
Tcl_Obj	****objvPtr*	(out)	A location where **Tcl_ListObjGetElements** stores a pointer to an array of pointers to the element objects of *listPtr*.

int	*objc*	(in)	The number of Tcl objects that **Tcl_NewListObj** will insert into a new list object, and **Tcl_ListObjReplace** will insert into *listPtr*. For **Tcl_SetListObj**, the number of Tcl objects to insert into *objPtr*.
Tcl_Obj	*CONST *objv*[]	(in)	An array of pointers to objects. **Tcl_NewListObj** will insert these objects into a new list object and **Tcl_ListObjReplace** will insert them into an existing *listPtr*. Each object will become a separate list element.
int	**intPtr*	(out)	Points to location where **Tcl_ListObjLength** stores the length of the list.
int	*index*	(in)	Index of the list element that **Tcl_ListObjIndex** is to return. The first element has index 0.
Tcl_Obj	***objPtrPtr*	(out)	Points to place where **Tcl_ListObjIndex** is to store a pointer to the resulting list element object.
int	*first*	(in)	Index of the starting list element that **Tcl_ListObjReplace** is to replace. The list's first element has index 0.
int	*count*	(in)	The number of elements that **Tcl_ListObjReplace** is to replace.

DESCRIPTION

Tcl list objects have an internal representation that supports the efficient indexing and appending. The procedures described in this man page are used to create, modify, index, and append to Tcl list objects from C code.

Tcl_ListObjAppendList and **Tcl_ListObjAppendElement** both add one or more objects to the end of the list object referenced by *listPtr*. **Tcl_ListObjAppendList** appends each element of the list object referenced by *elemListPtr* while **Tcl_ListObjAppendElement** appends the single object referenced by *objPtr*. Both procedures will convert the object referenced by *listPtr* to a list object if necessary. If an error occurs during conversion, both procedures return **TCL_ERROR** and leave an error message in the interpreter's result object if *interp* is not NULL. Similarly, if *elemListPtr* does not already refer to a list object, **Tcl_ListObjAppendList** will attempt to convert it to one and if an error occurs during conversion, will return **TCL_ERROR** and leave an error message in the interpreter's result object if interp is not NULL. Both procedures invalidate any old string representation of *listPtr* and, if it was converted to a list object, free any old internal representation. Similarly, **Tcl_ListObjAppendList** frees any old internal representation of *elemListPtr* if it converts it to a list object. After appending each element in *elemListPtr*, **Tcl_ListObjAppendList** increments the element's reference count since *listPtr* now also refers to it. For the same reason, **Tcl_ListObjAppendElement** increments *objPtr*'s reference count. If no error occurs, the two procedures return **TCL_OK** after appending the objects.

Tcl_NewListObj and **Tcl_SetListObj** create a new object or modify an existing object to hold the *objc* elements of the array referenced by *objv* where each element is a pointer to a Tcl object. If *objc* is less than or equal to zero, they return an empty object. The new object's string representation is left invalid. The two procedures increment the reference counts of the elements in *objc* since the list object now refers to them. The new list object returned by **Tcl_NewListObj** has reference count zero.

Tcl_ListObjGetElements returns a count and a pointer to an array of the elements in a list object. It returns the count by storing it in the address *objcPtr*. Similarly, it returns the array pointer by storing it in the address *objvPtr*. If *listPtr* is not already a list object, **Tcl_ListObjGetElements** will attempt to convert it to one; if the conversion fails, it returns **TCL_ERROR** and leaves an error message in the interpreter's result object if *interp* is not NULL. Otherwise it returns **TCL_OK** after storing the count and array pointer.

Tcl_ListObjLength returns the number of elements in the list object referenced by *listPtr*. It returns this count by storing an integer in the address *intPtr*. If the object is not already a list object, **Tcl_ListObjLength** will attempt to convert it to one; if the conversion fails, it returns **TCL_ERROR** and leaves an error message in the interpreter's result object if *interp* is not NULL. Otherwise it returns **TCL_OK** after storing the list's length.

The procedure **Tcl_ListObjIndex** returns a pointer to the object at element *index* in the list referenced by *listPtr*. It returns this object by storing a pointer to it in the address *objPtrPtr*. If *listPtr* does not already refer to a list object, **Tcl_ListObjIndex** will attempt to convert it to one; if the conversion fails, it returns **TCL_ERROR** and leaves an error message in the interpreter's result object if *interp* is not NULL. If the index is out of range, that is, *index* is negative or greater than or equal to the number of elements in the list, **Tcl_ListObjIndex** stores a NULL in *objPtrPtr* and returns **TCL_OK**. Otherwise it returns **TCL_OK** after storing the element's object pointer. The reference count for the list element is not incremented; the caller must do that if it needs to retain a pointer to the element.

Tcl_ListObjReplace replaces zero or more elements of the list referenced by *listPtr* with the *objc* objects in the array referenced by *objv*. If *listPtr* does not point to a list object, **Tcl_ListObjReplace** will attempt to convert it to one; if the conversion fails, it returns **TCL_ERROR** and leaves an error message in the interpreter's result object if *interp* is not NULL. Otherwise, it returns **TCL_OK** after replacing the objects. If *objv* is NULL, no new elements are added. If the argument *first* is zero or negative, it refers to the first element. If *first* is greater than or equal to the number of elements in the list, then no elements are deleted; the new elements are appended to the list. *count* gives the number of elements to replace. If *count* is zero or negative then no elements are deleted; the new elements are simply inserted before the one designated by *first*. **Tcl_ListObjReplace** invalidates *listPtr*'s old string representation. The reference counts of any elements inserted from *objv* are incremented since the resulting list now refers to them. Similarly, the reference counts for any replaced objects are decremented.

Because **Tcl_ListObjReplace** combines both element insertion and deletion, it can be used to implement a number of list operations. For example, the following code inserts the *objc* objects referenced by the array of object pointers *objv* just before the element *index* of the list referenced by *listPtr*:

```
result = Tcl_ListObjReplace(interp, listPtr, index, 0, objc, objv);
```

Similarly, the following code appends the *objc* objects referenced by the array *objv* to the end of the list *listPtr*:

```
result = Tcl_ListObjLength(interp, listPtr, &length);
if (result == TCL_OK) {
    result = Tcl_ListObjReplace(interp, listPtr, length, 0, objc, objv);
}
```

The *count* list elements starting at *first* can be deleted by simply calling **Tcl_ListObjReplace** with a NULL *objvPtr*:

```
result = Tcl_ListObjReplace(interp, listPtr, first, count, 0, NULL);
```

SEE ALSO

Tcl_NewObj, Tcl_DecrRefCount, Tcl_IncrRefCount, Tcl_GetObjResult

KEYWORDS

append, index, insert, internal representation, length, list, list object, list type, object, object type, replace, string representation

NAME

Tcl_CreateEventSource, Tcl_DeleteEventSource, Tcl_SetMaxBlockTime, Tcl_QueueEvent, Tcl_DeleteEvents, Tcl_WaitForEvent, Tcl_SetTimer, Tcl_ServiceAll, Tcl_ServiceEvent, Tcl_GetServiceMode, Tcl_SetServiceMode − the event queue and notifier interfaces

SYNOPSIS

#include <tcl.h>

Tcl_CreateEventSource(*setupProc, checkProc, clientData***)**

Tcl_DeleteEventSource(*setupProc, checkProc, clientData***)**

Tcl_SetMaxBlockTime(*timePtr***)**

Tcl_QueueEvent(*evPtr, position*)

Tcl_DeleteEvents(*deleteProc, clientData*)

int
Tcl_WaitForEvent(*timePtr*)

Tcl_SetTimer(*timePtr*)

int
Tcl_ServiceAll()

int
Tcl_ServiceEvent(*flags*)

int
Tcl_GetServiceMode()

int
Tcl_SetServiceMode(*mode*)

ARGUMENTS

Tcl_EventSetupProc	∗*setupProc*	(in)	Procedure to invoke to prepare for event wait in **Tcl_DoOneEvent**.
Tcl_EventCheckProc	∗*checkProc*	(in)	Procedure for **Tcl_DoOneEvent** to invoke after waiting for events. Checks to see if any events have occurred and, if so, queues them.
ClientData	*clientData*	(in)	Arbitrary one-word value to pass to *setupProc*, *checkProc*, or *deleteProc*.
Tcl_Time	∗*timePtr*	(in)	Indicates the maximum amount of time to wait for an event. This is specified as an interval (how long to wait), not an absolute time (when to wakeup). If the pointer passed to **Tcl_WaitForEvent** is NULL, it means there is no maximum wait time: wait forever if necessary.
Tcl_Event	∗*evPtr*	(in)	An event to add to the event queue. The storage for the

			event must have been allocated by the caller using **Tcl_Alloc** or **ckalloc**.
Tcl_QueuePosition	*position*	(in)	Where to add the new event in the queue: **TCL_QUEUE_TAIL**, **TCL_QUEUE_HEAD**, or **TCL_QUEUE_MARK**.
int	*flags*	(in)	What types of events to service. These flags are the same as those passed to **Tcl_DoOneEvent**.
Tcl_EventDeleteProc	∗*deleteProc*	(in)	Procedure to invoke for each queued event in **Tcl_DeleteEvents**.
int	*mode*	(in)	Inidicates whether events should be serviced by **Tcl_ServiceAll**. Must be one of **TCL_SERVICE_NONE** or **TCL_SERVICE_ALL**.

INTRODUCTION

The interfaces described here are used to customize the Tcl event loop. The two most common customizations are to add new sources of events and to merge Tcl's event loop with some other event loop, such as one provided by an application in which Tcl is embedded. Each of these tasks is described in a separate section below.

The procedures in this manual entry are the building blocks out of which the Tcl event notifier is constructed. The event notifier is the lowest layer in the Tcl event mechanism. It consists of three things:

[1] Event sources: these represent the ways in which events can be generated. For example, there is a timer event source that implements the **Tcl_CreateTimerHandler** procedure and the **after** command, and there is a file event source that implements the **Tcl_CreateFileHandler** procedure on Unix systems. An event source must work with the notifier to detect events at the right times, record them on the event queue, and eventually notify higher-level software that they have occurred. The procedures **Tcl_CreateEventSource**, **Tcl_DeleteEventSource**, and **Tcl_SetMaxBlockTime**, **Tcl_QueueEvent**, and **Tcl_DeleteEvents** are used primarily by event sources.

[2] The event queue: there is a single queue for the whole application, containing events that have been detected but not yet serviced. Event sources place events onto the queue so that they may be processed in order at appropriate times during the event loop. The event queue guarantees a fair discipline of event handling, so that no event source can starve the others. It also allows events to be saved for servicing at a future time. **Tcl_QueueEvent** is used (primarily by event sources) to add events to the event queue and **Tcl_DeleteEvents** is used to remove events from the queue without processing them.

[3] The event loop: in order to detect and process events, the application enters a loop that waits for events to occur, places them on the event queue, and then processes them. Most applications will do this by calling the procedure **Tcl_DoOneEvent**, which is described in a separate manual entry.

Most Tcl applications need not worry about any of the internals of the Tcl notifier. However, the notifier now has enough flexibility to be retargeted either for a new platform or to use an external event loop (such as the Motif event loop, when Tcl is embedded in a Motif application). The procedures **Tcl_WaitForEvent** and **Tcl_SetTimer** are normally implemented by Tcl, but may be replaced with new versions to retarget the notifier (the **Tcl_Sleep**, **Tcl_CreateFileHandler**, and **Tcl_DeleteFileHandler** must also be replaced; see CREATING A NEW NOTIFIER below for details). The procedures **Tcl_ServiceAll**, **Tcl_ServiceEvent**, **Tcl_GetServiceMode**, and **Tcl_SetServiceMode** are provided to help connect Tcl's event loop to an external event loop such as Motif's.

NOTIFIER BASICS

The easiest way to understand how the notifier works is to consider what happens when **Tcl_DoOneEvent** is called. **Tcl_DoOneEvent** is passed a *flags* argument that indicates what sort of events it is OK to process and also whether or not to block if no events are ready. **Tcl_DoOneEvent** does the following things:

[1] Check the event queue to see if it contains any events that can be serviced. If so, service the first possible event, remove it from the queue, and return. It does this by calling **Tcl_ServiceEvent** and passing in the *flags* argument.

[2] Prepare to block for an event. To do this, **Tcl_DoOneEvent** invokes a *setup procedure* in each event source. The event source will perform event-source specific initialization and possibly call **Tcl_SetMaxBlockTime** to limit how long **Tcl_WaitForEvent** will block if no new events occur.

[3] Call **Tcl_WaitForEvent**. This procedure is implemented differently on different platforms; it waits for an event to occur, based on the information provided by the event sources. It may cause the application to block if *timePtr* specifies an interval other than 0. **Tcl_WaitForEvent** returns when something has happened, such as a file becoming readable or the interval given by *timePtr* expiring. If there are no events for **Tcl_WaitForEvent** to wait for, so that it would block forever, then it returns immediately and **Tcl_DoOneEvent** returns 0.

[4] Call a *check procedure* in each event source. The check procedure determines whether any events of interest to this source occurred. If so, the events are added to the event queue.

[5] Check the event queue to see if it contains any events that can be serviced. If so, service the first possible event, remove it from the queue, and return.

[6] See if there are idle callbacks pending. If so, invoke all of them and return.

[7] Either return 0 to indicate that no events were ready, or go back to step [2] if blocking was requested by the caller.

CREATING A NEW EVENT SOURCE

An event source consists of three procedures invoked by the notifier, plus additional C procedures that are invoked by higher-level code to arrange for event-driven callbacks. The three procedures called by the notifier consist of the setup and check procedures described above, plus an additional procedure that is invoked when an event is removed from the event queue for servicing.

The procedure **Tcl_CreateEventSource** creates a new event source. Its arguments specify the setup procedure and check procedure for the event source. *SetupProc* should match the following prototype:

```
typedef void Tcl_EventSetupProc(
    ClientData clientData,
    int flags);
```

The *clientData* argument will be the same as the *clientData* argument to **Tcl_CreateEventSource**; it is typically used to point to private information managed by the event source. The *flags* argument will be the same as the *flags* argument passed to **Tcl_DoOneEvent** except that it will never be 0 (**Tcl_DoOneEvent** replaces 0 with **TCL_ALL_EVENTS**). *Flags* indicates what kinds of events should be considered; if the bit corresponding to this event source isn't set, the event source should return immediately without doing anything. For example, the file event source checks for the **TCL_FILE_EVENTS** bit.

SetupProc's job is to make sure that the application wakes up when events of the desired type occur. This is typically done in a platform-dependent fashion. For example, under Unix an event source might call **Tcl_CreateFileHandler**; under Windows it might request notification with a Windows event. For timer-driven event sources such as timer events or any polled event, the event source can call **Tcl_SetMaxBlockTime** to force the application to wake up after a specified time even if no events have occurred. If no event source calls **Tcl_SetMaxBlockTime** then **Tcl_WaitForEvent** will wait as long as necessary for an event to occur; otherwise, it will only wait as long as the shortest interval passed to **Tcl_SetMaxBlockTime** by one of the event sources. If an event source knows that it already has events ready to report, it can request a

zero maximum block time. For example, the setup procedure for the X event source looks to see if there are events already queued. If there are, it calls **Tcl_SetMaxBlockTime** with a 0 block time so that **Tcl_WaitForEvent** does not block if there is no new data on the X connection. The *timePtr* argument to **Tcl_WaitForEvent** points to a structure that describes a time interval in seconds and microseconds:

```
typedef struct Tcl_Time {
    long sec;
    long usec;
} Tcl_Time;
```

The *usec* field should be less than 1000000.

Information provided to **Tcl_SetMaxBlockTime** is only used for the next call to **Tcl_WaitForEvent**; it is discarded after **Tcl_WaitForEvent** returns. The next time an event wait is done each of the event sources' setup procedures will be called again, and they can specify new information for that event wait.

If the application uses an external event loop rather than **Tcl_DoOneEvent**, the event sources may need to call **Tcl_SetMaxBlockTime** at other times. For example, if a new event handler is registered that needs to poll for events, the event source may call **Tcl_SetMaxBlockTime** to set the block time to zero to force the external event loop to call Tcl. In this case, **Tcl_SetMaxBlockTime** invokes **Tcl_SetTimer** with the shortest interval seen since the last call to **Tcl_DoOneEvent** or **Tcl_ServiceAll**.

In addition to the generic procedure **Tcl_SetMaxBlockTime**, other platform-specific procedures may also be available for *setupProc*, if there is additional information needed by **Tcl_WaitForEvent** on that platform. For example, on Unix systems the **Tcl_CreateFileHandler** interface can be used to wait for file events.

The second procedure provided by each event source is its check procedure, indicated by the *checkProc* argument to **Tcl_CreateEventSource**. *CheckProc* must match the following prototype:

```
typedef void Tcl_EventCheckProc(
    ClientData clientData,
    int flags);
```

The arguments to this procedure are the same as those for *setupProc*. **CheckProc** is invoked by **Tcl_DoOneEvent** after it has waited for events. Presumably at least one event source is now prepared to queue an event. **Tcl_DoOneEvent** calls each of the event sources in turn, so they all have a chance to queue any events that are ready. The check procedure does two things. First, it must see if any events have triggered. Different event sources do this in different ways.

If an event source's check procedure detects an interesting event, it must add the event to Tcl's event queue. To do this, the event source calls **Tcl_QueueEvent**. The *evPtr* argument is a pointer to a dynamically allocated structure containing the event (see below for more information on memory management issues). Each event source can define its own event structure with whatever information is relevant to that event source. However, the first element of the structure must be a structure of type **Tcl_Event**, and the address of this structure is used when communicating between the event source and the rest of the notifier. A **Tcl_Event** has the following definition:

```
typedef struct Tcl_Event {
   Tcl_EventProc *proc;
   struct Tcl_Event *nextPtr;
};
```

The event source must fill in the *proc* field of the event before calling **Tcl_QueueEvent**. The *nextPtr* is used to link together the events in the queue and should not be modified by the event source.

An event may be added to the queue at any of three positions, depending on the *position* argument to **Tcl_QueueEvent**:

TCL_QUEUE_TAIL Add the event at the back of the queue, so that all other pending events will be serviced first. This is almost always the right place for new events.

TCL_QUEUE_HEAD Add the event at the front of the queue, so that it will be serviced before all other queued events.

TCL_QUEUE_MARK Add the event at the front of the queue, unless there are other events at the front whose position is **TCL_QUEUE_MARK**; if so, add the new event just after all other **TCL_QUEUE_MARK** events. This value of *position* is used to insert an ordered sequence of events at the front of the queue, such as a series of Enter and Leave events synthesized during a grab or ungrab operation in Tk.

When it is time to handle an event from the queue (steps 1 and 4 above) **Tcl_ServiceEvent** will invoke the *proc* specified in the first queued **Tcl_Event** structure. *Proc* must match the following prototype:

```
typedef int Tcl_EventProc(
    Tcl_Event *evPtr,
    int flags);
```

The first argument to *proc* is a pointer to the event, which will be the same as the first argument to the **Tcl_QueueEvent** call that added the event to the queue. The second argument to *proc* is the *flags* argument for the current call to **Tcl_ServiceEvent**; this is used by the event source to return immediately if its events are not relevant.

It is up to *proc* to handle the event, typically by invoking one or more Tcl commands or C-level callbacks. Once the event source has finished handling the event it returns 1 to indicate that the event can be removed from the queue. If for some reason the event source decides that the event cannot be handled at this time, it may return 0 to indicate that the event should be deferred for processing later; in this case **Tcl_ServiceEvent** will go on to the next event in the queue and attempt to service it. There are several reasons why an event source might defer an event. One possibility is that events of this type are excluded by the *flags* argument. For example, the file event source will always return 0 if the **TCL_FILE_EVENTS** bit isn't set in *flags*. Another example of deferring events happens in Tk if **Tk_RestrictEvents** has been invoked to defer certain kinds of window events.

When *proc* returns 1, **Tcl_ServiceEvent** will remove the event from the event queue and free its storage. Note that the storage for an event must be allocated by the event source (using **Tcl_Alloc** or the Tcl macro **ckalloc**) before calling **Tcl_QueueEvent**, but it will be freed by **Tcl_ServiceEvent**, not by the event source.

Tcl_DeleteEvents can be used to explicitly remove one or more events from the event queue. **Tcl_DeleteEvents** calls *proc* for each event in the queue, deleting those for with the procedure returns 1. Events for which the procedure returns 0 are left in the queue. *Proc* should match the following prototype:

```
typedef int Tcl_EventDeleteProc(
    Tcl_Event *evPtr,
    ClientData clientData);
```

The *clientData* argument will be the same as the *clientData* argument to **Tcl_DeleteEvents**; it is typically used to point to private information managed by the event source. The *evPtr* will point to the next event in the queue.

CREATING A NEW NOTIFIER

The notifier consists of all the procedures described in this manual entry, plus **Tcl_DoOneEvent** and **Tcl_Sleep**, which are available on all platforms, and **Tcl_CreateFileHandler** and **Tcl_DeleteFileHandler**, which are Unix-specific. Most of these procedures are generic, in that they are the same for all notifiers. However, five of the procedures are notifier-dependent: **Tcl_SetTimer**, **Tcl_Sleep**, **Tcl_WaitForEvent**, **Tcl_CreateFileHandler** and **Tcl_DeleteFileHandler**. To support a new platform or to integrate Tcl with an application-specific event loop, you must write new versions of these procedures.

Tcl_WaitForEvent is the lowest-level procedure in the notifier; it is responsible for waiting for an ''interesting'' event to occur or for a given time to elapse. Before **Tcl_WaitForEvent** is invoked, each of the event sources' setup procedure will have been invoked. The *timePtr* argument to **Tcl_WaitForEvent** gives the maximum time to block for an event, based on calls to **Tcl_SetMaxBlockTime** made by setup procedures and on other information (such as the **TCL_DONT_WAIT** bit in *flags*).

Ideally, **Tcl_WaitForEvent** should only wait for an event to occur; it should not actually process the event in any way. Later on, the event sources will process the raw events and create Tcl_Events on the event queue in their *checkProc* procedures. However, on some platforms (such as Windows) this isn't possible; events may be processed in **Tcl_WaitForEvent**, including queuing Tcl_Events and more (for example, callbacks for native widgets may be invoked). The return value from **Tcl_WaitForEvent** must be either 0, 1, or −1. On platforms such as Windows where events get processed in **Tcl_WaitForEvent**, a return value of 1 means that there may be more events still pending that haven't been processed. This is a sign to the caller that it must call **Tcl_WaitForEvent** again if it wants all pending events to be processed. A 0 return value means that calling **Tcl_WaitForEvent** again will not have any effect: either this is a platform where **Tcl_WaitForEvent** only waits without doing any event processing, or **Tcl_WaitForEvent** knows for sure that there are no additional events to process (e.g. it returned because the time elapsed). Finally, a return value of −1 means that the event loop is no longer operational and the application should probably unwind and terminate. Under Windows this happens when a WM_QUIT message is received; under Unix it happens when **Tcl_WaitForEvent** would have waited forever because there were no active event sources and the timeout was infinite.

If the notifier will be used with an external event loop, then it must also support the **Tcl_SetTimer** interface. **Tcl_SetTimer** is invoked by **Tcl_SetMaxBlockTime** whenever the maximum blocking time has been reduced. **Tcl_SetTimer** should arrange for the external event loop to invoke **Tcl_ServiceAll** after the specified interval even if no events have occurred. This interface is needed because **Tcl_WaitForEvent** isn't invoked when there is an external event loop. If the notifier will only be used from **Tcl_DoOneEvent**, then **Tcl_SetTimer** need not do anything.

On Unix systems, the file event source also needs support from the notifier. The file event source consists of the **Tcl_CreateFileHandler** and **Tcl_DeleteFileHandler** procedures, which are described elsewhere.

The **Tcl_Sleep** and **Tcl_DoOneEvent** interfaces are described elsewhere.

The easiest way to create a new notifier is to look at the code for an existing notifier, such as the files **unix/tclUnixNotfy.c** or **win/tclWinNotify.c** in the Tcl source distribution.

EXTERNAL EVENT LOOPS

The notifier interfaces are designed so that Tcl can be embedded into applications that have their own private event loops. In this case, the application does not call **Tcl_DoOneEvent** except in the case of recursive event loops such as calls to the Tcl commands **update** or **vwait**. Most of the time is spent in the external event loop of the application. In this case the notifier must arrange for the external event loop to call back into Tcl when something happens on the various Tcl event sources. These callbacks should arrange for appropriate Tcl events to be placed on the Tcl event queue.

Because the external event loop is not calling **Tcl_DoOneEvent** on a regular basis, it is up to the notifier to arrange for **Tcl_ServiceEvent** to be called whenever events are pending on the Tcl event queue. The easiest way to do this is to invoke **Tcl_ServiceAll** at the end of each callback from the external event loop. This will ensure that all of the event sources are polled, any queued events are serviced, and any pending idle handlers are processed before returning control to the application. In addition, event sources that need to poll for events can call **Tcl_SetMaxBlockTime** to force the external event loop to call Tcl even if no events are available on the system event queue.

As a side effect of processing events detected in the main external event loop, Tcl may invoke **Tcl_DoOneEvent** to start a recursive event loop in commands like **vwait**. **Tcl_DoOneEvent** will invoke

the external event loop, which will result in callbacks as described in the preceding paragraph, which will result in calls to **Tcl_ServiceAll**. However, in these cases it is undesirable to service events in **Tcl_ServiceAll**. Servicing events there is unnecessary because control will immediately return to the external event loop and hence to **Tcl_DoOneEvent**, which can service the events itself. Furthermore, **Tcl_DoOneEvent** is supposed to service only a single event, whereas **Tcl_ServiceAll** normally services all pending events. To handle this situation, **Tcl_DoOneEvent** sets a flag for **Tcl_ServiceAll** that causes it to return without servicing any events. This flag is called the *service mode*; **Tcl_DoOneEvent** restores it to its previous value before it returns.

In some cases, however, it may be necessary for **Tcl_ServiceAll** to service events even when it has been invoked from **Tcl_DoOneEvent**. This happens when there is yet another recursive event loop invoked via an event handler called by **Tcl_DoOneEvent** (such as one that is part of a native widget). In this case, **Tcl_DoOneEvent** may not have a chance to service events so **Tcl_ServiceAll** must service them all. Any recursive event loop that calls an external event loop rather than **Tcl_DoOneEvent** must reset the service mode so that all events get processed in **Tcl_ServiceAll**. This is done by invoking the **Tcl_SetServiceMode** procedure. If **Tcl_SetServiceMode** is passed **TCL_SERVICE_NONE**, then calls to **Tcl_ServiceAll** will return immediately without processing any events. If **Tcl_SetServiceMode** is passed **TCL_SERVICE_ALL**, then calls to **Tcl_ServiceAll** will behave normally. **Tcl_SetServiceMode** returns the previous value of the service mode, which should be restored when the recursive loop exits. **Tcl_GetServiceMode** returns the current value of the service mode.

KEYWORDS

event, notifier, event queue, event sources, file events, timer, idle, service mode

NAME

Tcl_ObjSetVar2, Tcl_ObjGetVar2 − manipulate Tcl variables

SYNOPSIS

#include <tcl.h>

Tcl_Obj *
Tcl_ObjSetVar2(*interp, part1Ptr, part2Ptr, newValuePtr, flags*)

Tcl_Obj *
Tcl_ObjGetVar2(*interp, part1Ptr, part2Ptr, flags*)

ARGUMENTS

Tcl_Interp	*interp	(in)	Interpreter containing variable.
Tcl_Obj	*part1Ptr	(in)	Points to a Tcl object containing the variable's name. The name may include a series of **::** namespace qualifiers to specify a variable in a particular namespace. May refer to a scalar variable or an element of an array variable.
Tcl_Obj	*part2Ptr	(in)	If non-NULL, points to an object containing the name of an element within an array and *part1Ptr* must refer to an array variable.
Tcl_Obj	*newValuePtr	(in)	Points to a Tcl object containing the new value for the variable.
int	*flags*	(in)	OR-ed combination of bits providing additional information for operation. See below for valid values.

DESCRIPTION

These two procedures may be used to read and modify Tcl variables from C code. **Tcl_ObjSetVar2** will create a new variable or modify an existing one. It sets the specified variable to the object referenced by *newValuePtr* and returns a pointer to the object which is the variable's new value. The returned object may not be the same one referenced by *newValuePtr*; this might happen because variable traces may modify the variable's value. The reference count for the variable's old value is decremented and the reference count for its new value is incremented. If the new value for the variable is not the same one referenced by *newValuePtr* (perhaps as a result of a variable trace), then *newValuePtr*'s reference count is left unchanged. The reference count for the returned object is not incremented to reflect the returned reference. If the caller needs to keep a reference to the object, say in a data structure, it must increment its reference count using **Tcl_IncrRefCount**. If an error occurs in setting the variable (e.g. an array variable is referenced without giving an index into the array), then NULL is returned.

The variable name specified to **Tcl_ObjSetVar2** consists of two parts. *part1Ptr* contains the name of a scalar or array variable. If *part2Ptr* is NULL, the variable must be a scalar. If *part2Ptr* is not NULL, it contains the name of an element in the array named by *part2Ptr*. As a special case, if the flag TCL_PARSE_PART1 is specified, *part1Ptr* may contain both an array and an element name: if the name contains an open parenthesis and ends with a close parenthesis, then the value between the parentheses is treated as an element name (which can have any string value) and the characters before the first open parenthesis are treated as the name of an array variable. If the flag TCL_PARSE_PART1 is given, *part2Ptr* should be NULL since the array and element names are taken from *part2Ptr*.

The *flags* argument may be used to specify any of several options to the procedures. It consists of an OR-ed combination of any of the following bits:

TCL_GLOBAL_ONLY

Under normal circumstances the procedures look up variables as follows: If a procedure call is active in *interp*, a variable is looked up at the current level of procedure call. Otherwise, a variable

is looked up first in the current namespace, then in the global namespace. However, if this bit is set in *flags* then the variable is looked up only in the global namespace even if there is a procedure call active. If both **TCL_GLOBAL_ONLY** and **TCL_NAMESPACE_ONLY** are given, **TCL_GLOBAL_ONLY** is ignored.

TCL_NAMESPACE_ONLY
Under normal circumstances the procedures look up variables as follows: If a procedure call is active in *interp*, a variable is looked up at the current level of procedure call. Otherwise, a variable is looked up first in the current namespace, then in the global namespace. However, if this bit is set in *flags* then the variable is looked up only in the current namespace even if there is a procedure call active.

TCL_LEAVE_ERR_MSG
If an error is returned and this bit is set in *flags*, then an error message will be left in the interpreter's result, where it can be retrieved with **Tcl_GetObjResult** or **Tcl_GetStringResult**. If this flag bit isn't set then no error message is left and the interpreter's result will not be modified.

TCL_APPEND_VALUE
If this bit is set then *newValuePtr* is appended to the current value, instead of replacing it. If the variable is currently undefined, then this bit is ignored.

TCL_LIST_ELEMENT
If this bit is set, then *newValuePtr* is converted to a valid Tcl list element before setting (or appending to) the variable. A separator space is appended before the new list element unless the list element is going to be the first element in a list or sublist (i.e. the variable's current value is empty, or contains the single character "{", or ends in " }").

TCL_PARSE_PART1
If this bit is set, then **Tcl_ObjGetVar2** and **Tcl_ObjSetVar2** will parse *part1Ptr* to obtain both an array name and an element name. If the name in *part1Ptr* contains an open parenthesis and ends with a close parenthesis, the name is treated as the name of an element of an array; otherwise, the name in *part1Ptr* is interpreted as the name of a scalar variable. When this bit is set, *part2Ptr* is ignored.

Tcl_ObjGetVar2 returns the value of the specified variable. Its arguments are treated the same way as those for **Tcl_ObjSetVar2**. It returns a pointer to the object which is the variable's value. The reference count for the returned object is not incremented. If the caller needs to keep a reference to the object, say in a data structure, it must increment the reference count using **Tcl_IncrRefCount**. If an error occurs in setting the variable (e.g. an array variable is referenced without giving an index into the array), then NULL is returned.

SEE ALSO

Tcl_GetObjResult, Tcl_GetStringResult, Tcl_GetVar, Tcl_GetVar2, Tcl_SetVar, Tcl_SetVar2, Tcl_TraceVar, Tcl_UnsetVar, Tcl_UnsetVar2

KEYWORDS

array, interpreter, object, scalar, set, unset, variable

NAME

Tcl_NewObj, Tcl_DuplicateObj, Tcl_IncrRefCount, Tcl_DecrRefCount, Tcl_IsShared − manipulate Tcl objects

SYNOPSIS

#include <tcl.h>

Tcl_Obj *
Tcl_NewObj()

Tcl_Obj *
Tcl_DuplicateObj(*objPtr*)

Tcl_IncrRefCount(*objPtr*)

Tcl_DecrRefCount(*objPtr*)

int
Tcl_IsShared(*objPtr*)

Tcl_InvalidateStringRep(*objPtr*)

ARGUMENTS

Tcl_Obj	*objPtr	(in)	Points to an object; must have been the result of a previous call to **Tcl_NewObj**.

INTRODUCTION

This man page presents an overview of Tcl objects and how they are used. It also describes generic procedures for managing Tcl objects. These procedures are used to create and copy objects, and increment and decrement the count of references (pointers) to objects. The procedures are used in conjunction with ones that operate on specific types of objects such as **Tcl_GetIntFromObj** and **Tcl_ListObjAppendElement**. The individual procedures are described along with the data structures they manipulate.

Tcl's *dual-ported* objects provide a general-purpose mechanism for storing and exchanging Tcl values. They largely replace the use of strings in Tcl. For example, they are used to store variable values, command arguments, command results, and scripts. Tcl objects behave like strings but also hold an internal representation that can be manipulated more efficiently. For example, a Tcl list is now represented as an object that holds the list's string representation as well as an array of pointers to the objects for each list element. Dual-ported objects avoid most runtime type conversions. They also improve the speed of many operations since an appropriate representation is immediately available. The compiler itself uses Tcl objects to cache the instruction bytecodes resulting from compiling scripts.

The two representations are a cache of each other and are computed lazily. That is, each representation is only computed when necessary, it is computed from the other representation, and, once computed, it is saved. In addition, a change in one representation invalidates the other one. As an example, a Tcl program doing integer calculations can operate directly on a variable's internal machine integer representation without having to constantly convert between integers and strings. Only when it needs a string representing the variable's value, say to print it, will the program regenerate the string representation from the integer. Although objects contain an internal representation, their semantics are defined in terms of strings: an up-to-date string can always be obtained, and any change to the object will be reflected in that string when the object's string representation is fetched. Because of this representation invalidation and regeneration, it is dangerous for extension writers to access **Tcl_Obj** fields directly. It is better to access Tcl_Obj information using procedures like **Tcl_GetStringFromObj**.

Objects are allocated on the heap and are referenced using a pointer to their **Tcl_Obj** structure. Objects are shared as much as possible. This significantly reduces storage requirements because some objects such as long lists are very large. Also, most Tcl values are only read and never modified. This is especially true for procedure arguments, which can be shared between the caller and the called procedure. Assignment and argument binding is done by simply assigning a pointer to the value. Reference counting is used to determine when it is safe to reclaim an object's storage.

Tcl objects are typed. An object's internal representation is controlled by its type. Seven types are predefined in the Tcl core including integer, double, list, and bytecode. Extension writers can extend the set of types by using the procedure **Tcl_RegisterObjType** .

THE TCL_OBJ STRUCTURE

Each Tcl object is represented by a **Tcl_Obj** structure which is defined as follows.

```
typedef struct Tcl_Obj {
    int refCount;
    char *bytes;
    int length;
    Tcl_ObjType *typePtr;
    union {
        long longValue;
        double doubleValue;
        VOID *otherValuePtr;
        struct {
            VOID *ptr1;
            VOID *ptr2;
        } twoPtrValue;
    } internalRep;
} Tcl_Obj;
```

The *bytes* and the *length* members together hold an object's string representation, which is a *counted* or *binary string* that may contain binary data with embedded null bytes. *bytes* points to the first byte of the string representation. The *length* member gives the number of bytes. The byte array must always have a null after the last byte, at offset *length*; this allows string representations that do not contain nulls to be treated as conventional null-terminated C strings. C programs use **Tcl_GetStringFromObj** to get an object's string representation. If *bytes* is NULL, the string representation is invalid.

An object's type manages its internal representation. The member *typePtr* points to the Tcl_ObjType structure that describes the type. If *typePtr* is NULL, the internal representation is invalid.

The *internalRep* union member holds an object's internal representation. This is either a (long) integer, a double-precision floating point number, a pointer to a value containing additional information needed by the object's type to represent the object, or two arbitrary pointers.

The *refCount* member is used to tell when it is safe to free an object's storage. It holds the count of active references to the object. Maintaining the correct reference count is a key responsibility of extension writers. Reference counting is discussed below in the section **STORAGE MANAGEMENT OF OBJECTS**.

Although extension writers can directly access the members of a Tcl_Obj structure, it is much better to use the appropriate procedures and macros. For example, extension writers should never read or update *refCount* directly; they should use macros such as **Tcl_IncrRefCount** and **Tcl_IsShared** instead.

A key property of Tcl objects is that they hold two representations. An object typically starts out containing only a string representation: it is untyped and has a NULL *typePtr*. An object containing an empty string or a copy of a specified string is created using **Tcl_NewObj** or **Tcl_NewStringObj** respectively. An object's string value is gotten with **Tcl_GetStringFromObj** and changed with **Tcl_SetStringObj**. If the object is later passed to a procedure like **Tcl_GetIntFromObj** that requires a specific internal

representation, the procedure will create one and set the object's *typePtr*. The internal representation is computed from the string representation. An object's two representations are duals of each other: changes made to one are reflected in the other. For example, **Tcl_ListObjReplace** will modify an object's internal representation and the next call to **Tcl_GetStringFromObj** will reflect that change.

Representations are recomputed lazily for efficiency. A change to one representation made by a procedure such as **Tcl_ListObjReplace** is not reflected immediately in the other representation. Instead, the other representation is marked invalid so that it is only regenerated if it is needed later. Most C programmers never have to be concerned with how this is done and simply use procedures such as **Tcl_GetBooleanFromObj** or **Tcl_ListObjIndex**. Programmers that implement their own object types must check for invalid representations and mark representations invalid when necessary. The procedure **Tcl_InvalidateStringRep** is used to mark an object's string representation invalid and to free any storage associated with the old string representation.

Objects usually remain one type over their life, but occasionally an object must be converted from one type to another. For example, a C program might build up a string in an object with repeated calls to **Tcl_StringObjAppend**, and then call **Tcl_ListObjIndex** to extract a list element from the object. The same object holding the same string value can have several different internal representations at different times. Extension writers can also force an object to be converted from one type to another using the **Tcl_ConvertToType** procedure. Only programmers that create new object types need to be concerned about how this is done. A procedure defined as part of the object type's implementation creates a new internal representation for an object and changes its *typePtr*. See the man page for **Tcl_RegisterObjType** to see how to create a new object type.

EXAMPLE OF THE LIFETIME OF AN OBJECT

As an example of the lifetime of an object, consider the following sequence of commands:

```
set x 123
```

This assigns to *x* an untyped object whose *bytes* member points to **123** and *length* member contains 3. The object's *typePtr* member is NULL.

```
puts "x is $x"
```

x's string representation is valid (since *bytes* is non-NULL) and is fetched for the command.

```
incr x
```

The **incr** command first gets an integer from *x*'s object by calling **Tcl_GetIntFromObj**. This procedure checks whether the object is already an integer object. Since it is not, it converts the object by setting the object's *internalRep.longValue* member to the integer **123** and setting the object's *typePtr* to point to the integer Tcl_ObjType structure. Both representations are now valid. **incr** increments the object's integer internal representation then invalidates its string representation (by calling **Tcl_InvalidateStringRep**) since the string representation no longer corresponds to the internal representation.

```
puts "x is now $x"
```

The string representation of *x*'s object is needed and is recomputed. The string representation is now **124**. and both representations are again valid.

STORAGE MANAGEMENT OF OBJECTS

Tcl objects are allocated on the heap and are shared as much as possible to reduce storage requirements. Reference counting is used to determine when an object is no longer needed and can safely be freed. An object just created by **Tcl_NewObj** or **Tcl_NewStringObj** has *refCount* 0. The macro **Tcl_IncrRefCount** increments the reference count when a new reference to the object is created. The macro **Tcl_DecrRefCount** decrements the count when a reference is no longer needed and, if the object's reference count drops to zero, frees its storage. An object shared by different code or data structures has *refCount* greater than 1. Incrementing an object's reference count ensures that it won't be freed too early or have its value change accidently.

As an example, the bytecode interpreter shares argument objects between calling and called Tcl procedures to avoid having to copy objects. It assigns the call's argument objects to the procedure's formal parameter variables. In doing so, it calls **Tcl_IncrRefCount** to increment the reference count of each argument since there is now a new reference to it from the formal parameter. When the called procedure returns, the interpreter calls **Tcl_DecrRefCount** to decrement each argument's reference count. When an object's reference count drops to zero, **Tcl_DecrRefCount** reclaims its storage. Most command procedures do not have to be concerned about reference counting since they use an object's value immediately and don't retain a pointer to the object after they return. However, if they do retain a pointer to an object in a data structure, they must be careful to increment its reference count since the retained pointer is a new reference.

Command procedures that directly modify objects such as those for **lappend** and **linsert** must be careful to copy a shared object before changing it. They must first check whether the object is shared by calling **Tcl_IsShared**. If the object is shared they must copy the object by using **Tcl_DuplicateObj**; this returns a new duplicate of the original object that has *refCount* 0. If the object is not shared, the command procedure "owns" the object and can safely modify it directly. For example, the following code appears in the command procedure that implements **linsert**. This procedure modifies the list object passed to it in *objv[1]* by inserting *objc-3* new elements before *index*.

```
listPtr = objv[1];
if (Tcl_IsShared(listPtr)) {
    listPtr = Tcl_DuplicateObj(listPtr);
}
result = Tcl_ListObjReplace(interp, listPtr, index, 0, (objc-3), &(objv[3]));
```

As another example, **incr**'s command procedure must check whether the variable's object is shared before incrementing the integer in its internal representation. If it is shared, it needs to duplicate the object in order to avoid accidently changing values in other data structures.

SEE ALSO

Tcl_ConvertToType, Tcl_GetIntFromObj, Tcl_ListObjAppendElement, Tcl_ListObjIndex, Tcl_ListObjReplace, Tcl_RegisterObjType

KEYWORDS

internal representation, object, object creation, object type, reference counting, string representation, type conversion

NAME

Tcl_RegisterObjType, Tcl_GetObjType, Tcl_AppendAllObjTypes, Tcl_ConvertToType − manipulate Tcl object types

SYNOPSIS

#include <tcl.h>

Tcl_RegisterObjType(*typePtr*)

Tcl_ObjType *
Tcl_GetObjType(*typeName*)

int
Tcl_AppendAllObjTypes(*interp, objPtr*)

int
Tcl_ConvertToType(*interp, objPtr, typePtr*)

ARGUMENTS

Tcl_ObjType	**typePtr*	(in)	Points to the structure containing information about the Tcl object type. This storage must must live forever, typically by being statically allocated.
char	**typeName*	(in)	The name of a Tcl object type that **Tcl_GetObjType** should look up.
Tcl_Interp	**interp*	(in)	Interpreter to use for error reporting.
Tcl_Obj	**objPtr*	(in)	For **Tcl_AppendAllObjTypes**, this points to the object onto which it appends the name of each object type as a list element. For **Tcl_ConvertToType**, this points to an object that must have been the result of a previous call to **Tcl_NewObj**.

DESCRIPTION

The procedures in this man page manage Tcl object types. The are used to register new object types, look up types, and force conversions from one type to another.

Tcl_RegisterObjType registers a new Tcl object type in the table of all object types supported by Tcl. The argument *typePtr* points to a Tcl_ObjType structure that describes the new type by giving its name and by supplying pointers to four procedures that implement the type. If the type table already containes a type with the same name as in *typePtr*, it is replaced with the new type. The Tcl_ObjType structure is described in the section **THE TCL_OBJTYPE STRUCTURE** below.

Tcl_GetObjType returns a pointer to the Tcl_ObjType with name *typeName*. It returns NULL if no type with that name is registered.

Tcl_AppendAllObjTypes appends the name of each object type as a list element onto the Tcl object referenced by *objPtr*. The return value is **TCL_OK** unless there was an error converting *objPtr* to a list object; in that case **TCL_ERROR** is returned.

Tcl_ConvertToType converts an object from one type to another if possible. It creates a new internal representation for *objPtr* appropriate for the target type *typePtr* and sets its *typePtr* member to that type. Any internal representation for *objPtr*'s old type is freed. If an error occurs during conversion, it returns **TCL_ERROR** and leaves an error message in the result object for *interp* unless *interp* is NULL. Otherwise, it returns **TCL_OK**. Passing a NULL *interp* allows this procedure to be used as a test whether the conversion can be done (and in fact was done).

THE TCL_OBJTYPE STRUCTURE

Extension writers can define new object types by defining four procedures, initializing a Tcl_ObjType structure to describe the type, and calling **Tcl_RegisterObjType**. The **Tcl_ObjType** structure is defined as follows:

```
typedef struct Tcl_ObjType {
    char *name;
    Tcl_FreeInternalRepProc *freeIntRepProc;
    Tcl_DupInternalRepProc *dupIntRepProc;
    Tcl_UpdateStringProc *updateStringProc;
    Tcl_SetFromAnyProc *setFromAnyProc;
} Tcl_ObjType;
```

The *name* member describes the name of the type, e.g. **int**. Extension writers can look up an object type using its name with the **Tcl_GetObjType** procedure. The remaining four members are pointers to procedures called by the generic Tcl object code:

The *setFromAnyProc* member contains the address of a function called to create a valid internal representation from an object's string representation.

```
typedef int (Tcl_SetFromAnyProc) (Tcl_Interp *interp, Tcl_Obj *objPtr);
```

If an internal representation can't be created from the string, it returns **TCL_ERROR** and puts a message describing the error in the result object for *interp* unless *interp* is NULL. If *setFromAnyProc* is successful, it stores the new internal representation, sets *objPtr*'s *typePtr* member to point to *setFromAnyProc*'s **Tcl_ObjType**, and returns **TCL_OK**. Before setting the new internal representation, the *setFromAnyProc* must free any internal representation of *objPtr*'s old type; it does this by calling the old type's *freeIntRepProc* if it is not NULL. As an example, the *setFromAnyProc* for the builtin Tcl integer type gets an up-to-date string representation for *objPtr* by calling **Tcl_GetStringFromObj**. It parses the string to obtain an integer and, if this succeeds, stores the integer in *objPtr*'s internal representation and sets *objPtr*'s *typePtr* member to point to the integer type's Tcl_ObjType structure.

The *updateStringProc* member contains the address of a function called to create a valid string representation from an object's internal representation.

```
typedef void (Tcl_UpdateStringProc) (Tcl_Obj *objPtr);
```

objPtr's *bytes* member is always NULL when it is called. It must always set *bytes* non-NULL before returning. We require the string representation's byte array to have a null after the last byte, at offset *length*; this allows string representations that do not contain null bytes to be treated as conventional null character-terminated C strings. Storage for the byte array must be allocated in the heap by **Tcl_Alloc**. Note that *updateStringProc*s must allocate enough storage for the string's bytes and the terminating null byte. The *updateStringProc* for Tcl's builtin list type, for example, builds an array of strings for each element object and then calls **Tcl_Merge** to construct a string with proper Tcl list structure. It stores this string as the list object's string representation.

The *dupIntRepProc* member contains the address of a function called to copy an internal representation from one object to another.

```
typedef void (Tcl_DupInternalRepProc) (Tcl_Obj *srcPtr, Tcl_Obj *dupPtr);
```

dupPtr's internal representation is made a copy of *srcPtr*'s internal representation. Before the call, *srcPtr*'s internal representation is valid and *dupPtr*'s is not. *srcPtr*'s object type determines what copying its internal representation means. For example, the *dupIntRepProc* for the Tcl integer type simply copies an integer. The builtin list type's *dupIntRepProc* allocates a new array that points at the original element objects; the elements are shared between the two lists (and their reference counts are incremented to reflect the new references).

The *freeIntRepProc* member contains the address of a function that is called when an object is freed.

```
typedef void (Tcl_FreeInternalRepProc) (Tcl_Obj *objPtr);
```

The *freeIntRepProc* function can deallocate the storage for the object's internal representation and do other type-specific processing necessary when an object is freed. For example, Tcl list objects have an

internalRep.otherValuePtr that points to an array of pointers to each element in the list. The list type's *freeIntRepProc* decrements the reference count for each element object (since the list will no longer refer to those objects), then deallocates the storage for the array of pointers. The *freeIntRepProc* member can be set to NULL to indicate that the internal representation does not require freeing.

SEE ALSO

Tcl_NewObj, Tcl_DecrRefCount, Tcl_IncrRefCount

KEYWORDS

internal representation, object, object type, string representation, type conversion

NAME

Tcl_OpenFileChannel, Tcl_OpenCommandChannel, Tcl_MakeFileChannel, Tcl_GetChannel, Tcl_RegisterChannel, Tcl_UnregisterChannel, Tcl_Close, Tcl_Read, Tcl_Gets, Tcl_Write, Tcl_Flush, Tcl_Seek, Tcl_Tell, Tcl_Eof, Tcl_InputBlocked, Tcl_InputBuffered, Tcl_GetChannelOption, Tcl_SetChannelOption − buffered I/O facilities using channels

SYNOPSIS

#include <tcl.h>

typedef ... Tcl_Channel;

Tcl_Channel
Tcl_OpenFileChannel(*interp, fileName, mode, permissions*)

Tcl_Channel
Tcl_OpenCommandChannel(*interp, argc, argv, flags*)

Tcl_Channel
Tcl_MakeFileChannel(*handle, readOrWrite*)

Tcl_Channel
Tcl_GetChannel(*interp, channelName, modePtr*)

void
Tcl_RegisterChannel(*interp, channel*)

int
Tcl_UnregisterChannel(*interp, channel*)

int
Tcl_Close(*interp, channel*)

int
Tcl_Read(*channel, buf, toRead*)

int
Tcl_Gets(*channel, lineRead*)

int
Tcl_GetsObj(*channel, lineObjPtr*)

int
Tcl_Write(*channel, buf, toWrite*)

int
Tcl_Flush(*channel*)

int
Tcl_Seek(*channel, offset, seekMode*)

int
Tcl_Tell(*channel*)

int
Tcl_GetChannelOption(*interp, channel, optionName, optionValue*)

int
Tcl_SetChannelOption(*interp, channel, optionName, newValue*)

int
Tcl_Eof(*channel*)

int
Tcl_InputBlocked(*channel*)

int
Tcl_InputBuffered(*channel*)

ARGUMENTS

Tcl_Interp	∗*interp*	(in)	Used for error reporting and to look up a channel registered in it.
char	∗*fileName*	(in)	The name of a local or network file.
char	∗*mode*	(in)	Specifies how the file is to be accessed. May have any of the values allowed for the *mode* argument to the Tcl **open** command. For **Tcl_OpenCommandChannel**, may be NULL.
int	*permissions*	(in)	POSIX-style permission flags such as 0644. If a new file is created, these permissions will be set on the created file.
int	*argc*	(in)	The number of elements in *argv*.
char	∗∗*argv*	(in)	Arguments for constructing a command pipeline. These values have the same meaning as the non-switch arguments to the Tcl **exec** command.
int	*flags*	(in)	Specifies the disposition of the stdio handles in pipeline: OR-ed combination of **TCL_STDIN**, **TCL_STDOUT**, **TCL_STDERR**, and **TCL_ENFORCE_MODE**. If **TCL_STDIN** is set, stdin for the first child in the pipe is the pipe channel, otherwise it is the same as the standard input of the invoking process; likewise for **TCL_STDOUT** and **TCL_STDERR**. If **TCL_ENFORCE_MODE** is not set, then the pipe can redirect stdio handles to override the stdio handles for which **TCL_STDIN**, **TCL_STDOUT** and **TCL_STDERR** have been set. If it is set, then such redirections cause an error.
ClientData	*handle*	(in)	Operating system specific handle for I/O to a file. For Unix this is a file descriptor, for Windows it is a HANDLE.
int	*readOrWrite*	(in)	OR-ed combination of **TCL_READABLE** and **TCL_WRITABLE** to indicate what operations are valid on *handle*.

int	*modePtr	(out)	Points at an integer variable that will receive an OR-ed combination of **TCL_READABLE** and **TCL_WRITABLE** denoting whether the channel is open for reading and writing.
Tcl_Channel	*channel*	(in)	A Tcl channel for input or output. Must have been the return value from a procedure such as **Tcl_OpenFileChannel**.
char	*buf	(in)	An array of bytes in which to store channel input, or from which to read channel output.
int	*len*	(in)	The length of the input or output.
int	*atEnd*	(in)	If nonzero, store the input at the end of the input queue, otherwise store it at the head of the input queue.
int	*toRead*	(in)	The number of bytes to read from the channel.
Tcl_DString	*lineRead	(in)	A pointer to a Tcl dynamic string in which to store the line read from the channel. Must have been initialized by the caller. The line read will be appended to any data already in the dynamic string.
Tcl_Obj	*linePtrObj	(in)	A pointer to a Tcl object in which to store the line read from the channel. The line read will be appended to the current value of the object.
int	*toWrite*	(in)	The number of bytes to read from *buf* and output to the channel.
int	*offset*	(in)	How far to move the access point in the channel at which the next input or output operation will be applied, measured in bytes from the position given by *seekMode*. May be either positive or negative.
int	*seekMode*	(in)	Relative to which point to seek; used with *offset* to calculate the new access point for the channel. Legal values are **SEEK_SET**, **SEEK_CUR**, and **SEEK_END**.
char	*optionName	(in)	The name of an option applicable to this channel, such as **−blocking**. May have any of the values accepted by the **fconfigure** command.
Tcl_DString	*optionValue	(in)	Where to store the value of an option or a list of all options and their values. Must have been initialized by the caller.
char	*newValue	(in)	New value for the option given by *optionName*.

DESCRIPTION

The Tcl channel mechanism provides a device-independent and platform-independent mechanism for performing buffered input and output operations on a variety of file, socket, and device types. The channel mechanism is extensible to new channel types, by providing a low level channel driver for the new type; the channel driver interface is described in the manual entry for **Tcl_CreateChannel**. The channel mechanism provides a buffering scheme modelled after Unix's standard I/O, and it also allows for nonblocking I/O on channels.

The procedures described in this manual entry comprise the C APIs of the generic layer of the channel architecture. For a description of the channel driver architecture and how to implement channel drivers for new types of channels, see the manual entry for **Tcl_CreateChannel**.

TCL_OPENFILECHANNEL

Tcl_OpenFileChannel opens a file specified by *fileName* and returns a channel handle that can be used to perform input and output on the file. This API is modelled after the **fopen** procedure of the Unix standard I/O library. The syntax and meaning of all arguments is similar to those given in the Tcl **open** command when opening a file. If an error occurs while opening the channel, **Tcl_OpenFileChannel** returns NULL and records a POSIX error code that can be retrieved with **Tcl_GetErrno**. In addition, if *interp* is non-NULL, **Tcl_OpenFileChannel** leaves an error message in *interp->result* after any error.

The newly created channel is not registered in the supplied interpreter; to register it, use **Tcl_RegisterChannel**, described below. If one of the standard channels, **stdin, stdout** or **stderr** was previously closed, the act of creating the new channel also assigns it as a replacement for the standard channel.

TCL_OPENCOMMANDCHANNEL

Tcl_OpenCommandChannel provides a C-level interface to the functions of the **exec** and **open** commands. It creates a sequence of subprocesses specified by the *argv* and *argc* arguments and returns a channel that can be used to communicate with these subprocesses. The *flags* argument indicates what sort of communication will exist with the command pipeline.

If the **TCL_STDIN** flag is set then the standard input for the first subprocess will be tied to the channel: writing to the channel will provide input to the subprocess. If **TCL_STDIN** is not set, then standard input for the first subprocess will be the same as this application's standard input. If **TCL_STDOUT** is set then standard output from the last subprocess can be read from the channel; otherwise it goes to this application's standard output. If **TCL_STDERR** is set, standard error output for all subprocesses is returned to the channel and results in an error when the channel is closed; otherwise it goes to this application's standard error. If **TCL_ENFORCE_MODE** is not set, then *argc* and *argv* can redirect the stdio handles to override **TCL_STDIN**, **TCL_STDOUT**, and **TCL_STDERR**; if it is set, then it is an error for argc and argv to override stdio channels for which **TCL_STDIN**, **TCL_STDOUT**, and **TCL_STDERR** have been set.

If an error occurs while opening the channel, **Tcl_OpenCommandChannel** returns NULL and records a POSIX error code that can be retrieved with **Tcl_GetErrno**. In addition, **Tcl_OpenCommandChannel** leaves an error message in *interp->result* if *interp* is not NULL.

The newly created channel is not registered in the supplied interpreter; to register it, use **Tcl_RegisterChannel**, described below. If one of the standard channels, **stdin, stdout** or **stderr** was previously closed, the act of creating the new channel also assigns it as a replacement for the standard channel.

TCL_MAKEFILECHANNEL

Tcl_MakeFileChannel makes a **Tcl_Channel** from an existing, platform-specific, file handle. The newly created channel is not registered in the supplied interpreter; to register it, use **Tcl_RegisterChannel**, described below. If one of the standard channels, **stdin, stdout** or **stderr** was previously closed, the act of creating the new channel also assigns it as a replacement for the standard channel.

TCL_GETCHANNEL

Tcl_GetChannel returns a channel given the *channelName* used to create it with **Tcl_CreateChannel** and a pointer to a Tcl interpreter in *interp*. If a channel by that name is not registered in that interpreter, the procedure returns NULL. If the *mode* argument is not NULL, it points at an integer variable that will receive an OR-ed combination of **TCL_READABLE** and **TCL_WRITABLE** describing whether the channel is

open for reading and writing.

TCL_REGISTERCHANNEL

Tcl_RegisterChannel adds a channel to the set of channels accessible in *interp*. After this call, Tcl programs executing in that interpreter can refer to the channel in input or output operations using the name given in the call to **Tcl_CreateChannel**. After this call, the channel becomes the property of the interpreter, and the caller should not call **Tcl_Close** for the channel; the channel will be closed automatically when it is unregistered from the interpreter.

Code executing outside of any Tcl interpreter can call **Tcl_RegisterChannel** with *interp* as NULL, to indicate that it wishes to hold a reference to this channel. Subsequently, the channel can be registered in a Tcl interpreter and it will only be closed when the matching number of calls to **Tcl_UnregisterChannel** have been made. This allows code executing outside of any interpreter to safely hold a reference to a channel that is also registered in a Tcl interpreter.

TCL_UNREGISTERCHANNEL

Tcl_UnregisterChannel removes a channel from the set of channels accessible in *interp*. After this call, Tcl programs will no longer be able to use the channel's name to refer to the channel in that interpreter. If this operation removed the last registration of the channel in any interpreter, the channel is also closed and destroyed.

Code not associated with a Tcl interpreter can call **Tcl_UnregisterChannel** with *interp* as NULL, to indicate to Tcl that it no longer holds a reference to that channel. If this is the last reference to the channel, it will now be closed.

TCL_CLOSE

Tcl_Close destroys the channel *channel*, which must denote a currently open channel. The channel should not be registered in any interpreter when **Tcl_Close** is called. Buffered output is flushed to the channel's output device prior to destroying the channel, and any buffered input is discarded. If this is a blocking channel, the call does not return until all buffered data is successfully sent to the channel's output device. If this is a nonblocking channel and there is buffered output that cannot be written without blocking, the call returns immediately; output is flushed in the background and the channel will be closed once all of the buffered data has been output. In this case errors during flushing are not reported.

If the channel was closed successfully, **Tcl_Close** returns **TCL_OK**. If an error occurs, **Tcl_Close** returns **TCL_ERROR** and records a POSIX error code that can be retrieved with **Tcl_GetErrno**. If the channel is being closed synchronously and an error occurs during closing of the channel and *interp* is not NULL, an error message is left in *interp->result*.

Note: it is not safe to call **Tcl_Close** on a channel that has been registered using **Tcl_RegisterChannel**; see the documentation for **Tcl_RegisterChannel**, above, for details. If the channel has ever been given as the **chan** argument in a call to **Tcl_RegisterChannel**, you should instead use **Tcl_UnregisterChannel**, which will internally call **Tcl_Close** when all calls to **Tcl_RegisterChannel** have been matched by corresponding calls to **Tcl_UnregisterChannel**.

TCL_READ

Tcl_Read consumes up to *toRead* bytes of data from *channel* and stores it at *buf*. The return value of **Tcl_Read** is the number of characters written at *buf*. The buffer produced by **Tcl_Read** is not NULL terminated. Its contents are valid from the zeroth position up to and excluding the position indicated by the return value. If an error occurs, the return value is -1 and **Tcl_Read** records a POSIX error code that can be retrieved with **Tcl_GetErrno**.

The return value may be smaller than the value of *toRead*, indicating that less data than requested was available, also called a *short read*. In blocking mode, this can only happen on an end-of-file. In nonblocking mode, a short read can also occur if there is not enough input currently available: **Tcl_Read** returns a short count rather than waiting for more data.

If the channel is in blocking mode, a return value of zero indicates an end of file condition. If the channel is in nonblocking mode, a return value of zero indicates either that no input is currently available or an end of file condition. Use **Tcl_Eof** and **Tcl_InputBlocked** to tell which of these conditions actually occurred.

Tcl_Read translates platform-specific end-of-line representations into the canonical **\n** internal representation according to the current end-of-line recognition mode. End-of-line recognition and the various platform-specific modes are described in the manual entry for the Tcl **fconfigure** command.

TCL_GETS AND TCL_GETSOBJ

Tcl_Gets reads a line of input from a channel and appends all of the characters of the line except for the terminating end-of-line character(s) to the dynamic string given by *dsPtr*. The end-of-line character(s) are read and discarded.

If a line was successfully read, the return value is greater than or equal to zero, and it indicates the number of characters stored in the dynamic string. If an error occurs, **Tcl_Gets** returns -1 and records a POSIX error code that can be retrieved with **Tcl_GetErrno**. **Tcl_Gets** also returns -1 if the end of the file is reached; the **Tcl_Eof** procedure can be used to distinguish an error from an end-of-file condition.

If the channel is in nonblocking mode, the return value can also be -1 if no data was available or the data that was available did not contain an end-of-line character. When -1 is returned, the **Tcl_InputBlocked** procedure may be invoked to determine if the channel is blocked because of input unavailability.

Tcl_GetsObj is the same as **Tcl_Gets** except the resulting characters are appended to a Tcl object **lineObjPtr** rather than a dynamic string.

TCL_WRITE

Tcl_Write accepts *toWrite* bytes of data at *buf* for output on *channel*. This data may not appear on the output device immediately. If the data should appear immediately, call **Tcl_Flush** after the call to **Tcl_Write**, or set the **-buffering** option on the channel to **none**. If you wish the data to appear as soon as an end of line is accepted for output, set the **−buffering** option on the channel to **line** mode.

The *toWrite* argument specifies how many bytes of data are provided in the *buf* argument. If it is negative, **Tcl_Write** expects the data to be NULL terminated and it outputs everything up to the NULL.

The return value of **Tcl_Write** is a count of how many characters were accepted for output to the channel. This is either equal to *toWrite* or -1 to indicate that an error occurred. If an error occurs, **Tcl_Write** also records a POSIX error code that may be retrieved with **Tcl_GetErrno**.

Newline characters in the output data are translated to platform-specific end-of-line sequences according to the **−translation** option for the channel.

TCL_FLUSH

Tcl_Flush causes all of the buffered output data for *channel* to be written to its underlying file or device as soon as possible. If the channel is in blocking mode, the call does not return until all the buffered data has been sent to the channel or some error occurred. The call returns immediately if the channel is nonblocking; it starts a background flush that will write the buffered data to the channel eventually, as fast as the channel is able to absorb it.

The return value is normally **TCL_OK**. If an error occurs, **Tcl_Flush** returns **TCL_ERROR** and records a POSIX error code that can be retrieved with **Tcl_GetErrno**.

TCL_SEEK

Tcl_Seek moves the access point in *channel* where subsequent data will be read or written. Buffered output is flushed to the channel and buffered input is discarded, prior to the seek operation.

Tcl_Seek normally returns the new access point. If an error occurs, **Tcl_Seek** returns -1 and records a POSIX error code that can be retrieved with **Tcl_GetErrno**. After an error, the access point may or may not have been moved.

TCL_TELL

Tcl_Tell returns the current access point for a channel. The returned value is -1 if the channel does not support seeking.

TCL_GETCHANNELOPTION

Tcl_GetChannelOption retrieves, in *dsPtr*, the value of one of the options currently in effect for a channel, or a list of all options and their values. The *channel* argument identifies the channel for which to query an option or retrieve all options and their values. If *optionName* is not NULL, it is the name of the option to query; the option's value is copied to the Tcl dynamic string denoted by *optionValue*. If *optionName* is NULL, the function stores an alternating list of option names and their values in *optionValue*, using a series of calls to **Tcl_DStringAppendElement**. The various preexisting options and their possible values are described in the manual entry for the Tcl **fconfigure** command. Other options can be added by each channel type. These channel type specific options are described in the manual entry for the Tcl command that creates a channel of that type; for example, the additional options for TCP based channels are described in the manual entry for the Tcl **socket** command. The procedure normally returns **TCL_OK**. If an error occurs, it returns **TCL_ERROR** and calls **Tcl_SetErrno** to store an appropriate POSIX error code.

TCL_SETCHANNELOPTION

Tcl_SetChannelOption sets a new value for an option on *channel*. *OptionName* is the option to set and *newValue* is the value to set. The procedure normally returns **TCL_OK**. If an error occurs, it returns **TCL_ERROR**; in addition, if *interp* is non-NULL, **Tcl_SetChannelOption** leaves an error message in *interp->result*.

TCL_EOF

Tcl_Eof returns a nonzero value if *channel* encountered an end of file during the last input operation.

TCL_INPUTBLOCKED

Tcl_InputBlocked returns a nonzero value if *channel* is in nonblocking mode and the last input operation returned less data than requested because there was insufficient data available. The call always returns zero if the channel is in blocking mode.

TCL_INPUTBUFFERED

Tcl_InputBuffered returns the number of bytes of input currently buffered in the internal buffers for a channel. If the channel is not open for reading, this function always returns zero.

PLATFORM ISSUES

The handles returned from **Tcl_GetChannelHandle** depend on the platform and the channel type. On Unix platforms, the handle is always a Unix file descriptor as returned from the **open** system call. On Windows platforms, the handle is a file **HANDLE** when the channel was created with **Tcl_OpenFileChannel**, **Tcl_OpenCommandChannel**, or **Tcl_MakeFileChannel**. Other channel types may return a different type of handle on Windows platforms. On the Macintosh platform, the handle is a file reference number as

returned from **HOpenDF**.

SEE ALSO

DString(3), fconfigure(n), filename(n), fopen(2), Tcl_CreateChannel(3)

KEYWORDS

access point, blocking, buffered I/O, channel, channel driver, end of file, flush, input, nonblocking, output, read, seek, write

NAME

Tcl_OpenTcpClient, Tcl_MakeTcpClientChannel, Tcl_OpenTcpServer − procedures to open channels using TCP sockets

SYNOPSIS

#include <tcl.h>

Tcl_Channel
Tcl_OpenTcpClient(*interp, port, host, myaddr, myport, async*)

Tcl_Channel
Tcl_MakeTcpClientChannel(*sock*)

Tcl_Channel
Tcl_OpenTcpServer(*interp, port, myaddr, proc, clientData*)

ARGUMENTS

Tcl_Interp	∗*interp*	(in)	Tcl interpreter to use for error reporting. If non-NULL and an error occurs, an error message is left in *interp->result*.
int	*port*	(in)	A port number to connect to as a client or to listen on as a server.
char	∗*host*	(in)	A string specifying a host name or address for the remote end of the connection.
int	*myport*	(in)	A port number for the client's end of the socket. If 0, a port number is allocated at random.
char	∗*myaddr*	(in)	A string specifying the host name or address for network interface to use for the local end of the connection. If NULL, a default interface is chosen.
int	*async*	(in)	If nonzero, the client socket is connected asynchronously to the server.
ClientData	*sock*	(in)	Platform-specific handle for client TCP socket.
Tcl_TcpAcceptProc	∗*proc*	(in)	Pointer to a procedure to invoke each time a new connection is accepted via the socket.
ClientData	*clientData*	(in)	Arbitrary one-word value to pass to *proc*.

DESCRIPTION

These functions are convenience procedures for creating channels that communicate over TCP sockets. The operations on a channel are described in the manual entry for **Tcl_OpenFileChannel**.

TCL_OPENTCPCLIENT

Tcl_OpenTcpClient opens a client TCP socket connected to a *port* on a specific *host*, and returns a channel that can be used to communicate with the server. The host to connect to can be specified either as a domain name style name (e.g. **www.sunlabs.com**), or as a string containing the alphanumeric representation of its four-byte address (e.g. **127.0.0.1**). Use the string **localhost** to connect to a TCP socket on the host on which the function is invoked.

The *myaddr* and *myport* arguments allow a client to specify an address for the local end of the connection. If *myaddr* is NULL, then an interface is chosen automatically by the operating system. If *myport* is 0, then a port number is chosen at random by the operating system.

If *async* is zero, the call to **Tcl_OpenTcpClient** returns only after the client socket has either successfully connected to the server, or the attempted connection has failed. If *async* is nonzero the socket is connected asynchronously and the returned channel may not yet be connected to the server when the call to **Tcl_OpenTcpClient** returns. If the channel is in blocking mode and an input or output operation is done on the channel before the connection is completed or fails, that operation will wait until the connection either completes successfully or fails. If the channel is in nonblocking mode, the input or output operation will return immediately and a subsequent call to **Tcl_InputBlocked** on the channel will return nonzero.

The returned channel is opened for reading and writing. If an error occurs in opening the socket, **Tcl_OpenTcpClient** returns NULL and records a POSIX error code that can be retrieved with **Tcl_GetErrno**. In addition, if *interp* is non-NULL, an error message is left in *interp->result*.

The newly created channel is not registered in the supplied interpreter; to register it, use **Tcl_RegisterChannel**. If one of the standard channels, **stdin, stdout** or **stderr** was previously closed, the act of creating the new channel also assigns it as a replacement for the standard channel.

TCL_MAKETCPCLIENTCHANNEL

Tcl_MakeTcpClientChannel creates a **Tcl_Channel** around an existing, platform specific, handle for a client TCP socket.

The newly created channel is not registered in the supplied interpreter; to register it, use **Tcl_RegisterChannel**. If one of the standard channels, **stdin, stdout** or **stderr** was previously closed, the act of creating the new channel also assigns it as a replacement for the standard channel.

TCL_OPENTCPSERVER

Tcl_OpenTcpServer opens a TCP socket on the local host on a specified *port* and uses the Tcl event mechanism to accept requests from clients to connect to it. The *myaddr* argument specifies the network interface. If *myaddr* is NULL the special address INADDR_ANY should be used to allow connections from any network interface. Each time a client connects to this socket, Tcl creates a channel for the new connection and invokes *proc* with information about the channel. *Proc* must match the following prototype:

```
typedef void Tcl_TcpAcceptProc(
    ClientData clientData,
    Tcl_Channel channel,
    char *hostName,
    int port);
```

The *clientData* argument will be the same as the *clientData* argument to **Tcl_OpenTcpServer**, *channel* will be the handle for the new channel, *hostName* points to a string containing the name of the client host making the connection, and *port* will contain the client's port number. The new channel is opened for both input and output. If *proc* raises an error, the connection is closed automatically. *Proc* has no return value, but if it wishes to reject the connection it can close *channel*.

Tcl_OpenTcpServer normally returns a pointer to a channel representing the server socket. If an error occurs, **Tcl_OpenTcpServer** returns NULL and records a POSIX error code that can be retrieved with **Tcl_GetErrno**. In addition, if *interp->result* is non-NULL, an error message is left in *interp->result*.

The channel returned by **Tcl_OpenTcpServer** cannot be used for either input or output. It is simply a handle for the socket used to accept connections. The caller can close the channel to shut down the server and disallow further connections from new clients.

TCP server channels operate correctly only in applications that dispatch events through **Tcl_DoOneEvent** or through Tcl commands such as **vwait**; otherwise Tcl will never notice that a connection request from a remote client is pending.

The newly created channel is not registered in the supplied interpreter; to register it, use **Tcl_Register-Channel**. If one of the standard channels, **stdin, stdout** or **stderr** was previously closed, the act of creating the new channel also assigns it as a replacement for the standard channel.

PLATFORM ISSUES

On Unix platforms, the socket handle is a Unix file descriptor as returned by the **socket** system call. On the Windows platform, the socket handle is a **SOCKET** as defined in the WinSock API. On the Macintosh platform, the socket handle is a **StreamPtr**.

SEE ALSO

Tcl_OpenFileChannel(3), Tcl_RegisterChannel(3), vwait(n)

KEYWORDS

client, server, TCP

NAME

Tcl_PkgRequire, Tcl_PkgProvide − package version control

SYNOPSIS

#include <tcl.h>

char *
Tcl_PkgRequire(*interp, name, version, exact*)

int
Tcl_PkgProvide(*interp, name, version*)

ARGUMENTS

Tcl_Interp	*interp	(in)	Interpreter where package is needed or available.
char	*name	(in)	Name of package.
char	*version	(in)	A version string consisting of one or more decimal numbers separated by dots.
int	*exact*	(in)	Non-zero means that only the particular version specified by *version* is acceptable. Zero means that newer versions than *version* are also acceptable as long as they have the same major version number as *version*.

DESCRIPTION

These procedures provide C-level interfaces to Tcl's package and version management facilities. **Tcl_PkgRequire** is equivalent to the **package require** command, and **Tcl_PkgProvide** is equivalent to the **package provide** command. See the documentation for the Tcl commands for details on what these procedures do. If **Tcl_PkgRequire** completes successfully it returns a pointer to the version string for the version of the package that is provided in the interpreter (which may be different than *version*); if an error occurs it returns NULL and leaves an error message in *interp->result*. **Tcl_PkgProvide** returns TCL_OK if it completes successfully; if an error occurs it returns TCL_ERROR and leaves an error message in *interp->result*.

KEYWORDS

package, provide, require, version

NAME

Tcl_Preserve, Tcl_Release, Tcl_EventuallyFree − avoid freeing storage while it's being used

SYNOPSIS

#include <tcl.h>

Tcl_Preserve(*clientData*)

Tcl_Release(*clientData*)

Tcl_EventuallyFree(*clientData, freeProc*)

ARGUMENTS

ClientData	*clientData*	(in)	Token describing structure to be freed or reallocated. Usually a pointer to memory for structure.
Tcl_FreeProc	∗*freeProc*	(in)	Procedure to invoke to free *clientData*.

DESCRIPTION

These three procedures help implement a simple reference count mechanism for managing storage. They are designed to solve a problem having to do with widget deletion, but are also useful in many other situations. When a widget is deleted, its widget record (the structure holding information specific to the widget) must be returned to the storage allocator. However, it's possible that the widget record is in active use by one of the procedures on the stack at the time of the deletion. This can happen, for example, if the command associated with a button widget causes the button to be destroyed: an X event causes an event-handling C procedure in the button to be invoked, which in turn causes the button's associated Tcl command to be executed, which in turn causes the button to be deleted, which in turn causes the button's widget record to be de-allocated. Unfortunately, when the Tcl command returns, the button's event-handling procedure will need to reference the button's widget record. Because of this, the widget record must not be freed as part of the deletion, but must be retained until the event-handling procedure has finished with it. In other situations where the widget is deleted, it may be possible to free the widget record immediately.

Tcl_Preserve and **Tcl_Release** implement short-term reference counts for their *clientData* argument. The *clientData* argument identifies an object and usually consists of the address of a structure. The reference counts guarantee that an object will not be freed until each call to **Tcl_Preserve** for the object has been matched by calls to **Tcl_Release**. There may be any number of unmatched **Tcl_Preserve** calls in effect at once.

Tcl_EventuallyFree is invoked to free up its *clientData* argument. It checks to see if there are unmatched **Tcl_Preserve** calls for the object. If not, then **Tcl_EventuallyFree** calls *freeProc* immediately. Otherwise **Tcl_EventuallyFree** records the fact that *clientData* needs eventually to be freed. When all calls to **Tcl_Preserve** have been matched with calls to **Tcl_Release** then *freeProc* will be called by **Tcl_Release** to do the cleanup.

All the work of freeing the object is carried out by *freeProc*. *FreeProc* must have arguments and result that match the type **Tcl_FreeProc**:

typedef void Tcl_FreeProc(char ∗*blockPtr*);

The *blockPtr* argument to *freeProc* will be the same as the *clientData* argument to **Tcl_EventuallyFree**. The type of *blockPtr* (**char** ∗) is different than the type of the *clientData* argument to **Tcl_EventuallyFree** for historical reasons, but the value is the same.

This mechanism can be used to solve the problem described above by placing **Tcl_Preserve** and **Tcl_Release** calls around actions that may cause undesired storage re-allocation. The mechanism is intended only for short-term use (i.e. while procedures are pending on the stack); it will not work

efficiently as a mechanism for long-term reference counts. The implementation does not depend in any way on the internal structure of the objects being freed; it keeps the reference counts in a separate structure.

KEYWORDS
free, reference count, storage

NAME

Tcl_PrintDouble − Convert floating value to string

SYNOPSIS

#include <tcl.h>

Tcl_PrintDouble(*interp, value, dst*)

ARGUMENTS

Tcl_Interp	∗*interp*	(in)	Before Tcl 8.0, the **tcl_precision** variable in this interpreter controlled the conversion. As of Tcl 8.0, this argument is ignored and the conversion is controlled by the **tcl_precision** variable that is now shared by all interpreters.
double	*value*	(in)	Floating-point value to be converted.
char	∗*dst*	(out)	Where to store string representing *value*. Must have at least TCL_DOUBLE_SPACE characters of storage.

DESCRIPTION

Tcl_PrintDouble generates a string that represents the value of *value* and stores it in memory at the location given by *dst*. It uses **%g** format to generate the string, with one special twist: the string is guaranteed to contain either a ‘‘.’’ or an ‘‘e’’ so that it doesn’t look like an integer. Where **%g** would generate an integer with no decimal point, **Tcl_PrintDouble** adds ‘‘.0’’.

KEYWORDS

conversion, double-precision, floating-point, string

NAME

Tcl_RecordAndEvalObj − save command on history list before evaluating

SYNOPSIS

#include <tcl.h>

int
Tcl_RecordAndEvalObj(*interp, cmdPtr, flags*)

ARGUMENTS

Tcl_Interp	∗*interp*	(in)	Tcl interpreter in which to evaluate command.
Tcl_Obj	∗*cmdPtr*	(in)	Points to a Tcl object containing a command (or sequence of commands) to execute.
int	*flags*	(in)	An OR'ed combination of flag bits. TCL_NO_EVAL means record the command but don't evaluate it. TCL_EVAL_GLOBAL means evaluate the command at global level instead of the current stack level.

DESCRIPTION

Tcl_RecordAndEvalObj is invoked to record a command as an event on the history list and then execute it using **Tcl_EvalObj** (or **Tcl_GlobalEvalObj** if the TCL_EVAL_GLOBAL bit is set in *flags*). It returns a completion code such as TCL_OK just like **Tcl_EvalObj**, as well as a result object containing additional information (a result value or error message) that can be retrieved using **Tcl_GetObjResult**. If you don't want the command recorded on the history list then you should invoke **Tcl_EvalObj** instead of **Tcl_RecordAndEvalObj**. Normally **Tcl_RecordAndEvalObj** is only called with top-level commands typed by the user, since the purpose of history is to allow the user to re-issue recently-invoked commands. If the *flags* argument contains the TCL_NO_EVAL bit then the command is recorded without being evaluated.

SEE ALSO

Tcl_EvalObj, Tcl_GetObjResult

KEYWORDS

command, event, execute, history, interpreter, object, record

NAME

Tcl_RecordAndEval − save command on history list before evaluating

SYNOPSIS

#include <tcl.h>

int
Tcl_RecordAndEval(*interp, cmd, flags*)

ARGUMENTS

Tcl_Interp	∗*interp*	(in)	Tcl interpreter in which to evaluate command.
char	∗*cmd*	(in)	Command (or sequence of commands) to execute.
int	*flags*	(in)	An OR'ed combination of flag bits. TCL_NO_EVAL means record the command but don't evaluate it. TCL_EVAL_GLOBAL means evaluate the command at global level instead of the current stack level.

DESCRIPTION

Tcl_RecordAndEval is invoked to record a command as an event on the history list and then execute it using **Tcl_Eval** (or **Tcl_GlobalEval** if the TCL_EVAL_GLOBAL bit is set in *flags*). It returns a completion code such as TCL_OK just like **Tcl_Eval** and it leaves information in *interp->result*. If you don't want the command recorded on the history list then you should invoke **Tcl_Eval** instead of **Tcl_RecordAndEval**. Normally **Tcl_RecordAndEval** is only called with top-level commands typed by the user, since the purpose of history is to allow the user to re-issue recently-invoked commands. If the *flags* argument contains the TCL_NO_EVAL bit then the command is recorded without being evaluated.

Note that **Tcl_RecordAndEval** has been largely replaced by the object-based procedure **Tcl_RecordAndEvalObj**. That object-based procedure records and optionally executes a command held in a Tcl object instead of a string.

SEE ALSO

Tcl_RecordAndEvalObj

KEYWORDS

command, event, execute, history, interpreter, record

NAME

Tcl_RegExpMatch, Tcl_RegExpCompile, Tcl_RegExpExec, Tcl_RegExpRange − Pattern matching with regular expressions

SYNOPSIS

#include <tcl.h>

int
Tcl_RegExpMatch(*interp*, *string*, *pattern*)

Tcl_RegExp
Tcl_RegExpCompile(*interp*, *pattern*)

int
Tcl_RegExpExec(*interp*, *regexp*, *string*, *start*)

Tcl_RegExpRange(*regexp*, *index*, *startPtr*, *endPtr*)

ARGUMENTS

Tcl_Interp	∗*interp*	(in)	Tcl interpreter to use for error reporting.
char	∗*string*	(in)	String to check for a match with a regular expression.
char	∗*pattern*	(in)	String in the form of a regular expression pattern.
Tcl_RegExp	*regexp*	(in)	Compiled regular expression. Must have been returned previously by **Tcl_RegExpCompile**.
char	∗*start*	(in)	If *string* is just a portion of some other string, this argument identifies the beginning of the larger string. If it isn't the same as *string*, then no ^ matches will be allowed.
int	*index*	(in)	Specifies which range is desired: 0 means the range of the entire match, 1 or greater means the range that matched a parenthesized sub-expression.
char	∗∗*startPtr*	(out)	The address of the first character in the range is stored here, or NULL if there is no such range.
char	∗∗*endPtr*	(out)	The address of the character just after the last one in the range is stored here, or NULL if there is no such range.

DESCRIPTION

Tcl_RegExpMatch determines whether its *pattern* argument matches *regexp*, where *regexp* is interpreted as a regular expression using the same rules as for the **regexp** Tcl command. If there is a match then **Tcl_RegExpMatch** returns 1. If there is no match then **Tcl_RegExpMatch** returns 0. If an error occurs in the matching process (e.g. *pattern* is not a valid regular expression) then **Tcl_RegExpMatch** returns −1 and leaves an error message in *interp->result*.

Tcl_RegExpCompile, **Tcl_RegExpExec**, and **Tcl_RegExpRange** provide lower-level access to the regular expression pattern matcher. **Tcl_RegExpCompile** compiles a regular expression string into the internal form used for efficient pattern matching. The return value is a token for this compiled form, which can be used in subsequent calls to **Tcl_RegExpExec** or **Tcl_RegExpRange**. If an error occurs while compiling the regular expression then **Tcl_RegExpCompile** returns NULL and leaves an error message in *interp->result*. Note: the return value from **Tcl_RegExpCompile** is only valid up to the next call to **Tcl_RegExpCompile**; it is not safe to retain these values for long periods of time.

Tcl_RegExpExec executes the regular expression pattern matcher. It returns 1 if *string* contains a range of characters that match *regexp*, 0 if no match is found, and −1 if an error occurs. In the case of an error, **Tcl_RegExpExec** leaves an error message in *interp->result*. When searching a string for multiple matches of a pattern, it is important to distinguish between the start of the original string and the start of the current search. For example, when searching for the second occurrence of a match, the *string* argument might point to the character just after the first match; however, it is important for the pattern matcher to know that this is not the start of the entire string, so that it doesn't allow ^ atoms in the pattern to match. The *start* argument provides this information by pointing to the start of the overall string containing *string*. *Start* will be less than or equal to *string*; if it is less than *string* then no ^ matches will be allowed.

Tcl_RegExpRange may be invoked after **Tcl_RegExpExec** returns; it provides detailed information about what ranges of the string matched what parts of the pattern. **Tcl_RegExpRange** returns a pair of pointers in ∗*startPtr* and ∗*endPtr* that identify a range of characters in the source string for the most recent call to **Tcl_RegExpExec**. *Index* indicates which of several ranges is desired: if *index* is 0, information is returned about the overall range of characters that matched the entire pattern; otherwise, information is returned about the range of characters that matched the *index*'th parenthesized subexpression within the pattern. If there is no range corresponding to *index* then NULL is stored in ∗*firstPtr* and ∗*lastPtr*.

KEYWORDS

match, pattern, regular expression, string, subexpression

NAME

Tcl_SetErrno, Tcl_GetErrno − manipulate errno to store and retrieve error codes

SYNOPSIS

#include <tcl.h>

void
Tcl_SetErrno(*errorCode*)

int
Tcl_GetErrno()

ARGUMENTS

int	*errorCode*	(in)	A POSIX error code such as **ENOENT**.

DESCRIPTION

Tcl_SetErrno and **Tcl_GetErrno** provide portable access to the **errno** variable, which is used to record a POSIX error code after system calls and other operations such as **Tcl_Gets**. These procedures are necessary because global variable accesses cannot be made across module boundaries on some platforms.

Tcl_SetErrno sets the **errno** variable to the value of the *errorCode* argument C procedures that wish to return error information to their callers via **errno** should call **Tcl_SetErrno** rather than setting **errno** directly.

Tcl_GetErrno returns the current value of **errno**. Procedures wishing to access **errno** should call this procedure instead of accessing **errno** directly.

KEYWORDS

errno, error code, global variables

NAME

Tcl_SetRecursionLimit − set maximum allowable nesting depth in interpreter

SYNOPSIS

#include <tcl.h>

int
Tcl_SetRecursionLimit(*interp, depth*)

ARGUMENTS

Tcl_Interp	∗*interp*	(in)	Interpreter whose recursion limit is to be set. Must be greater than zero.
int	*depth*	(in)	New limit for nested calls to **Tcl_Eval** for *interp*.

DESCRIPTION

At any given time Tcl enforces a limit on the number of recursive calls that may be active for **Tcl_Eval** and related procedures such as **Tcl_GlobalEval**. Any call to **Tcl_Eval** that exceeds this depth is aborted with an error. By default the recursion limit is 1000.

Tcl_SetRecursionLimit may be used to change the maximum allowable nesting depth for an interpreter. The *depth* argument specifies a new limit for *interp*, and **Tcl_SetRecursionLimit** returns the old limit. To read out the old limit without modifying it, invoke **Tcl_SetRecursionDepth** with *depth* equal to 0.

The **Tcl_SetRecursionLimit** only sets the size of the Tcl call stack: it cannot by itself prevent stack overflows on the C stack being used by the application. If your machine has a limit on the size of the C stack, you may get stack overflows before reaching the limit set by **Tcl_SetRecursionLimit**. If this happens, see if there is a mechanism in your system for increasing the maximum size of the C stack.

KEYWORDS

nesting depth, recursion

NAME

Tcl_SetObjResult, Tcl_GetObjResult, Tcl_SetResult, Tcl_GetStringResult, Tcl_AppendResult, Tcl_AppendElement, Tcl_ResetResult − manipulate Tcl result

SYNOPSIS

#include <tcl.h>

Tcl_SetObjResult(*interp, objPtr*)

Tcl_Obj ∗
Tcl_GetObjResult(*interp*)

Tcl_SetResult(*interp, string, freeProc*)

char ∗
Tcl_GetStringResult(*interp*)

Tcl_AppendResult(*interp, string, string, ... ,* **(char** ∗**) NULL**)

Tcl_AppendElement(*interp, string*)

Tcl_ResetResult(*interp*)

Tcl_FreeResult(*interp*)

ARGUMENTS

Tcl_Interp	∗*interp*	(out)	Interpreter whose result is to be modified or read.
Tcl_Obj	∗*objPtr*	(in)	Object value to become result for *interp*.
char	∗*string*	(in)	String value to become result for *interp* or to be appended to the existing result.
Tcl_FreeProc	∗*freeProc*	(in)	Address of procedure to call to release storage at *string*, or **TCL_STATIC**, **TCL_DYNAMIC**, or **TCL_VOLATILE**.

DESCRIPTION

The procedures described here are utilities for manipulating the result value in a Tcl interpreter. The interpreter result may be either a Tcl object or a string. For example, **Tcl_SetObjResult** and **Tcl_SetResult** set the interpreter result to, respectively, an object and a string. Similarly, **Tcl_GetObjResult** and **Tcl_GetStringResult** return the interpreter result as an object and as a string. The procedures always keep the string and object forms of the interpreter result consistent. For example, if **Tcl_SetObjResult** is called to set the result to an object, then **Tcl_GetStringResult** is called, it will return the object's string value.

Tcl_SetObjResult arranges for *objPtr* to be the result for *interp*, replacing any existing result. The result is left pointing to the object referenced by *objPtr*. *objPtr*'s reference count is incremented since there is now a new reference to it from *interp*. The reference count for any old result object is decremented and the old result object is freed if no references to it remain.

Tcl_GetObjResult returns the result for *interp* as an object. The object's reference count is not incremented; if the caller needs to retain a long-term pointer to the object they should use **Tcl_IncrRefCount** to increment its reference count in order to keep it from being freed too early or accidently changed.

Tcl_SetResult arranges for *string* to be the result for the current Tcl command in *interp*, replacing any existing result. The *freeProc* argument specifies how to manage the storage for the *string* argument; it is discussed in the section **THE TCL_FREEPROC ARGUMENT TO TCL_SETRESULT** below. If *string* is **NULL**, then *freeProc* is ignored and **Tcl_SetResult** re-initializes *interp*'s result to point to an empty string.

Tcl_GetStringResult returns the result for *interp* as an string. If the result was set to an object by a **Tcl_SetObjResult** call, the object form will be converted to a string and returned. If the object's string representation contains null bytes, this conversion will lose information. For this reason, programmers are encouraged to write their code to use the new object API procedures and to call **Tcl_GetObjResult** instead.

Tcl_ResetResult clears the result for *interp* and leaves the result in its normal empty initialized state. If the result is an object, its reference count is decremented and the result is left pointing to an unshared object representing an empty string. If the result is a dynamically allocated string, its memory is free∗d and the result is left as a empty string. **Tcl_ResetResult** also clears the error state managed by **Tcl_AddErrorInfo**, **Tcl_AddObjErrorInfo**, and **Tcl_SetErrorCode**.

OLD STRING PROCEDURES

Use of the following procedures is deprecated since they manipulate the Tcl result as a string. Procedures such as **Tcl_SetObjResult** that manipulate the result as an object can be significantly more efficient.

Tcl_AppendResult makes it easy to build up Tcl results in pieces. It takes each of its *string* arguments and appends them in order to the current result associated with *interp*. If the result is in its initialized empty state (e.g. a command procedure was just invoked or **Tcl_ResetResult** was just called), then **Tcl_AppendResult** sets the result to the concatenation of its *string* arguments. **Tcl_AppendResult** may be called repeatedly as additional pieces of the result are produced. **Tcl_AppendResult** takes care of all the storage management issues associated with managing *interp*'s result, such as allocating a larger result area if necessary. It also converts the current interpreter result from an object to a string, if necessary, before appending the argument strings. Any number of *string* arguments may be passed in a single call; the last argument in the list must be a NULL pointer.

Tcl_AppendElement is similar to **Tcl_AppendResult** in that it allows results to be built up in pieces. However, **Tcl_AppendElement** takes only a single *string* argument and it appends that argument to the current result as a proper Tcl list element. **Tcl_AppendElement** adds backslashes or braces if necessary to ensure that *interp*'s result can be parsed as a list and that *string* will be extracted as a single element. Under normal conditions, **Tcl_AppendElement** will add a space character to *interp*'s result just before adding the new list element, so that the list elements in the result are properly separated. However if the new list element is the first in a list or sub-list (i.e. *interp*'s current result is empty, or consists of the single character ''{'', or ends in the characters '' {'') then no space is added.

Tcl_FreeResult performs part of the work of **Tcl_ResetResult**. It frees up the memory associated with *interp*'s result. It also sets *interp->freeProc* to zero, but doesn't change *interp->result* or clear error state. **Tcl_FreeResult** is most commonly used when a procedure is about to replace one result value with another.

DIRECT ACCESS TO INTERP->RESULT IS DEPRECATED

It used to be legal for programs to directly read and write *interp->result* to manipulate the interpreter result. Direct access to *interp->result* is now strongly deprecated because it can make the result's string and object forms inconsistent. Programs should always read the result using the procedures **Tcl_GetObjResult** or **Tcl_GetStringResult**, and write the result using **Tcl_SetObjResult** or **Tcl_SetResult**.

THE TCL_FREEPROC ARGUMENT TO TCL_SETRESULT

Tcl_SetResult's *freeProc* argument specifies how the Tcl system is to manage the storage for the *string* argument. If **Tcl_SetResult** or **Tcl_SetObjResult** are called at a time when *interp* holds a string result,

they do whatever is necessary to dispose of the old string result (see the **Tcl_Interp** manual entry for details on this).

If *freeProc* is **TCL_STATIC** it means that *string* refers to an area of static storage that is guaranteed not to be modified until at least the next call to **Tcl_Eval**. If *freeProc* is **TCL_DYNAMIC** it means that *string* was allocated with a call to **Tcl_Alloc** and is now the property of the Tcl system. **Tcl_SetResult** will arrange for the string's storage to be released by calling **Tcl_Free** when it is no longer needed. If *freeProc* is **TCL_VOLATILE** it means that *string* points to an area of memory that is likely to be overwritten when **Tcl_SetResult** returns (e.g. it points to something in a stack frame). In this case **Tcl_SetResult** will make a copy of the string in dynamically allocated storage and arrange for the copy to be the result for the current Tcl command.

If *freeProc* isn't one of the values **TCL_STATIC**, **TCL_DYNAMIC**, and **TCL_VOLATILE**, then it is the address of a procedure that Tcl should call to free the string. This allows applications to use non-standard storage allocators. When Tcl no longer needs the storage for the string, it will call *freeProc*. *FreeProc* should have arguments and result that match the type **Tcl_FreeProc**:

```
typedef void Tcl_FreeProc(char *blockPtr);
```

When *freeProc* is called, its *blockPtr* will be set to the value of *string* passed to **Tcl_SetResult**.

SEE ALSO

Tcl_AddErrorInfo, Tcl_CreateObjCommand, Tcl_SetErrorCode, Tcl_Interp

KEYWORDS

append, command, element, list, object, result, return value, interpreter

NAME

Tcl_SetVar, Tcl_SetVar2, Tcl_GetVar, Tcl_GetVar2, Tcl_UnsetVar, Tcl_UnsetVar2 − manipulate Tcl variables

SYNOPSIS

#include <tcl.h>

char *
Tcl_SetVar(*interp, varName, newValue, flags*)

char *
Tcl_SetVar2(*interp, name1, name2, newValue, flags*)

char *
Tcl_GetVar(*interp, varName, flags*)

char *
Tcl_GetVar2(*interp, name1, name2, flags*)

int
Tcl_UnsetVar(*interp, varName, flags*)

int
Tcl_UnsetVar2(*interp, name1, name2, flags*)

ARGUMENTS

Tcl_Interp	*interp	(in)	Interpreter containing variable.
char	*varName	(in)	Name of variable. May include a series of **::** namespace qualifiers to specify a variable in a particular namespace. May refer to a scalar variable or an element of an array variable. If the name references an element of an array, then it must be in writable memory: Tcl will make temporary modifications to it while looking up the name.
char	*newValue	(in)	New value for variable.
int	*flags*	(in)	OR-ed combination of bits providing additional information for operation. See below for valid values.
char	*name1	(in)	Name of scalar variable, or name of array variable if *name2* is non-NULL. May include a series of **::** namespace qualifiers to specify a variable in a particular namespace.
char	*name2	(in)	If non-NULL, gives name of element within array and *name1* must refer to an array variable.

DESCRIPTION

These procedures may be used to create, modify, read, and delete Tcl variables from C code.

Note that **Tcl_GetVar** and **Tcl_SetVar** have been largely replaced by the object-based procedures **Tcl_ObjGetVar2** and **Tcl_ObjSetVar2**. Those object-based procedures read, modify, and create a variable whose name is held in a Tcl object instead of a string. They also return a pointer to the object which is the variable's value instead of returning a string. Operations on objects can be faster since objects hold an internal representation that can be manipulated more efficiently.

Tcl_SetVar and **Tcl_SetVar2** will create a new variable or modify an existing one. Both of these procedures set the given variable to the value given by *newValue*, and they return a pointer to a copy of the variable's new value, which is stored in Tcl's variable structure. Tcl keeps a private copy of the variable's value, so the caller may change *newValue* after these procedures return without affecting the value of the variable. If an error occurs in setting the variable (e.g. an array variable is referenced without giving an index into the array), they return NULL.

The name of the variable may be specified to **Tcl_SetVar** and **Tcl_SetVar2** in either of two ways. If **Tcl_SetVar** is called, the variable name is given as a single string, *varName*. If *varName* contains an open parenthesis and ends with a close parenthesis, then the value between the parentheses is treated as an index (which can have any string value) and the characters before the first open parenthesis are treated as the name of an array variable. If *varName* doesn't have parentheses as described above, then the entire string is treated as the name of a scalar variable. If **Tcl_SetVar2** is called, then the array name and index have been separated by the caller into two separate strings, *name1* and *name2* respectively; if *name2* is zero it means that a scalar variable is being referenced.

The *flags* argument may be used to specify any of several options to the procedures. It consists of an OR-ed combination of the following bits. Note that the flag bit TCL_PARSE_PART1 is only meaningful for the procedures Tcl_SetVar2 and Tcl_GetVar2.

TCL_GLOBAL_ONLY
: Under normal circumstances the procedures look up variables as follows: If a procedure call is active in *interp*, a variable is looked up at the current level of procedure call. Otherwise, a variable is looked up first in the current namespace, then in the global namespace. However, if this bit is set in *flags* then the variable is looked up only in the global namespace even if there is a procedure call active. If both **TCL_GLOBAL_ONLY** and **TCL_NAMESPACE_ONLY** are given, **TCL_GLOBAL_ONLY** is ignored.

TCL_NAMESPACE_ONLY
: Under normal circumstances the procedures look up variables as follows: If a procedure call is active in *interp*, a variable is looked up at the current level of procedure call. Otherwise, a variable is looked up first in the current namespace, then in the global namespace. However, if this bit is set in *flags* then the variable is looked up only in the current namespace even if there is a procedure call active.

TCL_LEAVE_ERR_MSG
: If an error is returned and this bit is set in *flags*, then an error message will be left in the interpreter's result, where it can be retrieved with **Tcl_GetObjResult** or **Tcl_GetStringResult**. If this flag bit isn't set then no error message is left and the interpreter's result will not be modified.

TCL_APPEND_VALUE
: If this bit is set then *newValue* is appended to the current value, instead of replacing it. If the variable is currently undefined, then this bit is ignored.

TCL_LIST_ELEMENT
: If this bit is set, then *newValue* is converted to a valid Tcl list element before setting (or appending to) the variable. A separator space is appended before the new list element unless the list element is going to be the first element in a list or sublist (i.e. the variable's current value is empty, or contains the single character "{", or ends in " }").

TCL_PARSE_PART1
: If this bit is set when calling *Tcl_SetVar2* and *Tcl_GetVar2*, *name1* may contain both an array and an element name: if the name contains an open parenthesis and ends with a close parenthesis, then the value between the parentheses is treated as an element name (which can have any string value) and the characters before the first open parenthesis are treated as the name of an array variable. If the flag TCL_PARSE_PART1 is given, *name2* should be NULL since the array and element names

are taken from *name1*.

Tcl_GetVar and **Tcl_GetVar2** return the current value of a variable. The arguments to these procedures are treated in the same way as the arguments to **Tcl_SetVar** and **Tcl_SetVar2**. Under normal circumstances, the return value is a pointer to the variable's value (which is stored in Tcl's variable structure and will not change before the next call to **Tcl_SetVar** or **Tcl_SetVar2**). **Tcl_GetVar** and **Tcl_GetVar2** use the flag bits TCL_GLOBAL_ONLY and TCL_LEAVE_ERR_MSG, both of which have the same meaning as for **Tcl_SetVar**. In addition, **Tcl_GetVar2** uses the bit TCL_PARSE_PART1, which has the same meaning as for **Tcl_SetVar2**. If an error occurs in reading the variable (e.g. the variable doesn't exist or an array element is specified for a scalar variable), then NULL is returned.

Tcl_UnsetVar and **Tcl_UnsetVar2** may be used to remove a variable, so that future calls to **Tcl_GetVar** or **Tcl_GetVar2** for the variable will return an error. The arguments to these procedures are treated in the same way as the arguments to **Tcl_GetVar** and **Tcl_GetVar2**. If the variable is successfully removed then TCL_OK is returned. If the variable cannot be removed because it doesn't exist then TCL_ERROR is returned. If an array element is specified, the given element is removed but the array remains. If an array name is specified without an index, then the entire array is removed.

SEE ALSO

Tcl_GetObjResult, Tcl_GetStringResult, Tcl_ObjGetVar2, Tcl_ObjSetVar2, Tcl_TraceVar

KEYWORDS

array, interpreter, object, scalar, set, unset, variable

NAME

Tcl_Sleep − delay execution for a given number of milliseconds

SYNOPSIS

#include <tcl.h>

Tcl_Sleep(*ms*)

ARGUMENTS

int	*ms*	(in)	Number of milliseconds to sleep.

DESCRIPTION

This procedure delays the calling process by the number of milliseconds given by the *ms* parameter and returns after that time has elapsed. It is typically used for things like flashing a button, where the delay is short and the application needn't do anything while it waits. For longer delays where the application needs to respond to other events during the delay, the procedure **Tcl_CreateTimerHandler** should be used instead of **Tcl_Sleep**.

KEYWORDS

sleep, time, wait

NAME

Tcl_SplitList, Tcl_Merge, Tcl_ScanElement, Tcl_ConvertElement − manipulate Tcl lists

SYNOPSIS

```
#include <tcl.h>

int
Tcl_SplitList(interp, list, argcPtr, argvPtr)

char *
Tcl_Merge(argc, argv)

int
Tcl_ScanElement(src, flagsPtr)

int
Tcl_ScanCountedElement(src, length, flagsPtr)

int
Tcl_ConvertElement(src, dst, flags)

int
Tcl_ConvertCountedElement(src, length, dst, flags)
```

ARGUMENTS

Tcl_Interp	**interp*	(out)	Interpreter to use for error reporting. If NULL, then no error message is left.
char	**list*	(in)	Pointer to a string with proper list structure.
int	**argcPtr*	(out)	Filled in with number of elements in *list*.
char	****argvPtr*	(out)	**argvPtr* will be filled in with the address of an array of pointers to the strings that are the extracted elements of *list*. There will be **argcPtr* valid entries in the array, followed by a NULL entry.
int	*argc*	(in)	Number of elements in *argv*.
char	***argv*	(in)	Array of strings to merge together into a single list. Each string will become a separate element of the list.
char	**src*	(in)	String that is to become an element of a list.
int	**flagsPtr*	(in)	Pointer to word to fill in with information about *src*. The value of **flagsPtr* must be passed to **Tcl_ConvertElement**.
int	*length*	(in)	Number of bytes in string *src*.
char	**dst*	(in)	Place to copy converted list element. Must contain enough characters to hold converted string.
int	*flags*	(in)	Information about *src*. Must be value returned by previous call to **Tcl_ScanElement**, possibly OR-ed with **TCL_DONT_USE_BRACES**.

DESCRIPTION

These procedures may be used to disassemble and reassemble Tcl lists. **Tcl_SplitList** breaks a list up into its constituent elements, returning an array of pointers to the elements using *argcPtr* and *argvPtr*. While extracting the arguments, **Tcl_SplitList** obeys the usual rules for backslash substitutions and braces. The area of memory pointed to by ∗*argvPtr* is dynamically allocated; in addition to the array of pointers, it also holds copies of all the list elements. It is the caller's responsibility to free up all of this storage. For example, suppose that you have called **Tcl_SplitList** with the following code:

```
int argc, code;
char *string;
char **argv;
...
code = Tcl_SplitList(interp, string, &argc, &argv);
```

Then you should eventually free the storage with a call like the following:

```
Tcl_Free((char *) argv);
```

Tcl_SplitList normally returns **TCL_OK**, which means the list was successfully parsed. If there was a syntax error in *list*, then **TCL_ERROR** is returned and *interp->result* will point to an error message describing the problem (if *interp* was not NULL). If **TCL_ERROR** is returned then no memory is allocated and ∗*argvPtr* is not modified.

Tcl_Merge is the inverse of **Tcl_SplitList**: it takes a collection of strings given by *argc* and *argv* and generates a result string that has proper list structure. This means that commands like **index** may be used to extract the original elements again. In addition, if the result of **Tcl_Merge** is passed to **Tcl_Eval**, it will be parsed into *argc* words whose values will be the same as the *argv* strings passed to **Tcl_Merge**. **Tcl_Merge** will modify the list elements with braces and/or backslashes in order to produce proper Tcl list structure. The result string is dynamically allocated using **Tcl_Alloc**; the caller must eventually release the space using **Tcl_Free**.

If the result of **Tcl_Merge** is passed to **Tcl_SplitList**, the elements returned by **Tcl_SplitList** will be identical to those passed into **Tcl_Merge**. However, the converse is not true: if **Tcl_SplitList** is passed a given string, and the resulting *argc* and *argv* are passed to **Tcl_Merge**, the resulting string may not be the same as the original string passed to **Tcl_SplitList**. This is because **Tcl_Merge** may use backslashes and braces differently than the original string.

Tcl_ScanElement and **Tcl_ConvertElement** are the procedures that do all of the real work of **Tcl_Merge**. **Tcl_ScanElement** scans its *src* argument and determines how to use backslashes and braces when converting it to a list element. It returns an overestimate of the number of characters required to represent *src* as a list element, and it stores information in ∗*flagsPtr* that is needed by **Tcl_ConvertElement**.

Tcl_ConvertElement is a companion procedure to **Tcl_ScanElement**. It does the actual work of converting a string to a list element. Its *flags* argument must be the same as the value returned by **Tcl_ScanElement**. **Tcl_ConvertElement** writes a proper list element to memory starting at ∗*dst* and returns a count of the total number of characters written, which will be no more than the result returned by **Tcl_ScanElement**. **Tcl_ConvertElement** writes out only the actual list element without any leading or trailing spaces: it is up to the caller to include spaces between adjacent list elements.

Tcl_ConvertElement uses one of two different approaches to handle the special characters in *src*. Wherever possible, it handles special characters by surrounding the string with braces. This produces clean-looking output, but can't be used in some situations, such as when *src* contains unmatched braces. In these situations, **Tcl_ConvertElement** handles special characters by generating backslash sequences for them. The caller may insist on the second approach by OR-ing the flag value returned by **Tcl_ScanElement** with **TCL_DONT_USE_BRACES**. Although this will produce an uglier result, it is useful in some special situations, such as when **Tcl_ConvertElement** is being used to generate a portion of an argument for a Tcl command. In this case, surrounding *src* with curly braces would cause the command not to be parsed correctly.

Tcl_ScanCountedElement and **Tcl_ConvertCountedElement** are the same as **Tcl_ScanElement** and **Tcl_ConvertElement**, except the length of string *src* is specified by the *length* argument, and the string may contain embedded nulls.

KEYWORDS

backslash, convert, element, list, merge, split, strings

NAME

Tcl_SplitPath, Tcl_JoinPath, Tcl_GetPathType − manipulate platform-dependent file paths

SYNOPSIS

```
#include <tcl.h>

Tcl_SplitPath(path, argcPtr, argvPtr)

char *
Tcl_JoinPath(argc, argv, resultPtr)

Tcl_PathType
Tcl_GetPathType(path)
```

ARGUMENTS

char	**path*	(in)	File path in a form appropriate for the current platform (see the **filename** manual entry for acceptable forms for path names).
int	**argcPtr*	(out)	Filled in with number of path elements in *path*.
char	****argvPtr*	(out)	**argvPtr* will be filled in with the address of an array of pointers to the strings that are the extracted elements of *path*. There will be **argcPtr* valid entries in the array, followed by a NULL entry.
int	*argc*	(in)	Number of elements in *argv*.
char	***argv*	(in)	Array of path elements to merge together into a single path.
Tcl_DString	**resultPtr*	(in/out)	A pointer to an initialized **Tcl_DString** to which the result of **Tcl_JoinPath** will be appended.

DESCRIPTION

These procedures may be used to disassemble and reassemble file paths in a platform independent manner: they provide C-level access to the same functionality as the **file split**, **file join**, and **file pathtype** commands.

Tcl_SplitPath breaks a path into its constituent elements, returning an array of pointers to the elements using *argcPtr* and *argvPtr*. The area of memory pointed to by **argvPtr* is dynamically allocated; in addition to the array of pointers, it also holds copies of all the path elements. It is the caller's responsibility to free all of this storage. For example, suppose that you have called **Tcl_SplitPath** with the following code:

```
int argc;
char *path;
char **argv;
...
Tcl_SplitPath(string, &argc, &argv);
```

Then you should eventually free the storage with a call like the following:

```
Tcl_Free((char *) argv);
```

Tcl_JoinPath is the inverse of **Tcl_SplitPath**: it takes a collection of path elements given by *argc* and *argv* and generates a result string that is a properly constructed path. The result string is appended to *resultPtr*. *ResultPtr* must refer to an initialized **Tcl_DString**.

If the result of **Tcl_SplitPath** is passed to **Tcl_JoinPath**, the result will refer to the same location, but may not be in the same form. This is because **Tcl_SplitPath** and **Tcl_JoinPath** eliminate duplicate path separators and return a normalized form for each platform.

Tcl_GetPathType returns the type of the specified *path*, where **Tcl_PathType** is one of **TCL_PATH_ABSOLUTE**, **TCL_PATH_RELATIVE**, or **TCL_PATH_VOLUME_RELATIVE**. See the **filename** manual entry for a description of the path types for each platform.

KEYWORDS

file, filename, join, path, split, type

NAME

Tcl_StaticPackage − make a statically linked package available via the **load** command

SYNOPSIS

#include <tcl.h>

Tcl_StaticPackage(*interp, pkgName, initProc, safeInitProc*)

ARGUMENTS

Tcl_Interp	∗*interp*	(in)	If not NULL, points to an interpreter into which the package has already been loaded (i.e., the caller has already invoked the appropriate initialization procedure). NULL means the package hasn't yet been incorporated into any interpreter.
char	∗*pkgName*	(in)	Name of the package; should be properly capitalized (first letter upper-case, all others lower-case).
Tcl_PackageInitProc	∗*initProc*	(in)	Procedure to invoke to incorporate this package into a trusted interpreter.
Tcl_PackageInitProc	∗*safeInitProc*	(in)	Procedure to call to incorporate this package into a safe interpreter (one that will execute untrusted scripts). NULL means the package can't be used in safe interpreters.

DESCRIPTION

This procedure may be invoked to announce that a package has been linked statically with a Tcl application and, optionally, that it has already been loaded into an interpreter. Once **Tcl_StaticPackage** has been invoked for a package, it may be loaded into interpreters using the **load** command. **Tcl_StaticPackage** is normally invoked only by the **Tcl_AppInit** procedure for the application, not by packages for themselves (**Tcl_StaticPackage** should only be invoked for statically loaded packages, and code in the package itself should not need to know whether the package is dynamically or statically loaded).

When the **load** command is used later to load the package into an interpreter, one of *initProc* and *safeInitProc* will be invoked, depending on whether the target interpreter is safe or not. *initProc* and *safeInitProc* must both match the following prototype:

typedef int Tcl_PackageInitProc(Tcl_Interp ∗*interp*);

The *interp* argument identifies the interpreter in which the package is to be loaded. The initialization procedure must return **TCL_OK** or **TCL_ERROR** to indicate whether or not it completed successfully; in the event of an error it should set *interp->result* to point to an error message. The result or error from the initialization procedure will be returned as the result of the **load** command that caused the initialization procedure to be invoked.

KEYWORDS

initialization procedure, package, static linking

NAME

Tcl_StringMatch − test whether a string matches a pattern

SYNOPSIS

```
#include <tcl.h>

int
Tcl_StringMatch(string, pattern)
```

ARGUMENTS

char	**string*	(in)	String to test.
char	**pattern*	(in)	Pattern to match against string. May contain special characters from the set *?\[].

DESCRIPTION

This utility procedure determines whether a string matches a given pattern. If it does, then **Tcl_StringMatch** returns 1. Otherwise **Tcl_StringMatch** returns 0. The algorithm used for matching is the same algorithm used in the ‘‘string match’’ Tcl command and is similar to the algorithm used by the C-shell for file name matching; see the Tcl manual entry for details.

KEYWORDS

match, pattern, string

NAME

Tcl_NewStringObj, Tcl_SetStringObj, Tcl_GetStringFromObj, Tcl_AppendToObj, Tcl_AppendStringsToObj, Tcl_SetObjLength, TclConcatObj − manipulate Tcl objects as strings

SYNOPSIS

#include <tcl.h>

Tcl_Obj *
Tcl_NewStringObj(*bytes*, *length*)

Tcl_SetStringObj(*objPtr*, *bytes*, *length*)

char *
Tcl_GetStringFromObj(*objPtr*, *lengthPtr*)

Tcl_AppendToObj(*objPtr*, *bytes*, *length*)

Tcl_AppendStringsToObj(*objPtr*, *string*, *string*, ... **(char *) NULL**)

Tcl_SetObjLength(*objPtr*, *newLength*)

Tcl_Obj *
Tcl_ConcatObj(*objc*, *objv*)

ARGUMENTS

char	**bytes*	(in)	Points to the first byte of an array of bytes used to set or append to a string object. This byte array may contain embedded null bytes unless *length* is negative.
int	*length*	(in)	The number of bytes to copy from *bytes* when initializing, setting, or appending to a string object. If negative, all bytes up to the first null are used.
Tcl_Obj	**objPtr*	(in/out)	Points to an object to manipulate.
int	**lengthPtr*	(out)	If non-NULL, the location where **Tcl_GetStringFromObj** will store the the length of an object's string representation.
char	**string*	(in)	Null-terminated string value to append to *objPtr*.
int	*newLength*	(in)	New length for the string value of *objPtr*, not including the final NULL character.
int	*objc*	(in)	The number of elements to concatenate.
Tcl_Obj	**objv[]*	(in)	The array of objects to concatenate.

DESCRIPTION

The procedures described in this manual entry allow Tcl objects to be manipulated as string values. They use the internal representation of the object to store additional information to make the string manipulations more efficient. In particular, they make a series of append operations efficient by allocating extra storage space for the string so that it doesn't have to be copied for each append.

Tcl_NewStringObj and **Tcl_SetStringObj** create a new object or modify an existing object to hold a copy of the string given by *bytes* and *length*. **Tcl_NewStringObj** returns a pointer to a newly created object with reference count zero. Both procedures set the object to hold a copy of the specified string.

Tcl_SetStringObj frees any old string representation as well as any old internal representation of the object.

Tcl_GetStringFromObj returns an object's string representation. This is given by the returned byte pointer and length, which is stored in *lengthPtr* if it is non-NULL. If the object's string representation is invalid (its byte pointer is NULL), the string representation is regenerated from the object's internal representation. The storage referenced by the returned byte pointer is owned by the object manager and should not be modified by the caller.

Tcl_AppendToObj appends the data given by *bytes* and *length* to the object specified by *objPtr*. It does this in a way that handles repeated calls relatively efficiently (it overallocates the string space to avoid repeated reallocations and copies of object's string value).

Tcl_AppendStringsToObj is similar to **Tcl_AppendToObj** except that it can be passed more than one value to append and each value must be a null-terminated string (i.e. none of the values may contain internal null characters). Any number of *string* arguments may be provided, but the last argument must be a NULL pointer to indicate the end of the list.

The **Tcl_SetObjLength** procedure changes the length of the string value of its *objPtr* argument. If the *newLength* argument is greater than the space allocated for the object's string, then the string space is reallocated and the old value is copied to the new space; the bytes between the old length of the string and the new length may have arbitrary values. If the *newLength* argument is less than the current length of the object's string, with *objPtr->length* is reduced without reallocating the string space; the original allocated size for the string is recorded in the object, so that the string length can be enlarged in a subsequent call to **Tcl_SetObjLength** without reallocating storage. In all cases **Tcl_SetObjLength** leaves a null character at *objPtr->bytes[newLength]*.

The **Tcl_ConcatObj** function returns a new string object whose value is the space-separated concatenation of the string representations of all of the objects in the *objv* array. **Tcl_ConcatObj** eliminates leading and trailing white space as it copies the string representations of the *objv* array to the result. If an element of the *objv* array consists of nothing but white space, then that object is ignored entirely. This white-space removal was added to make the output of the **concat** command cleaner-looking. **Tcl_ConcatObj** returns a pointer to a newly-created object whose ref count is zero.

SEE ALSO

Tcl_NewObj, Tcl_IncrRefCount, Tcl_DecrRefCount

KEYWORDS

append, internal representation, object, object type, string object, string type, string representation, concat, concatenate

NAME

Tcl_Main − main program for Tcl-based applications

SYNOPSIS

#include <tcl.h>

Tcl_Main(*argc, argv, appInitProc*)

ARGUMENTS

int	*argc*	(in)	Number of elements in *argv*.
char	∗*argv[]*	(in)	Array of strings containing command-line arguments.
Tcl_AppInitProc	∗*appInitProc*	(in)	Address of an application-specific initialization procedure. The value for this argument is usually **Tcl_AppInit**.

DESCRIPTION

Tcl_Main acts as the main program for most Tcl-based applications. Starting with Tcl 7.4 it is not called **main** anymore because it is part of the Tcl library and having a function **main** in a library (particularly a shared library) causes problems on many systems. Having **main** in the Tcl library would also make it hard to use Tcl in C++ programs, since C++ programs must have special C++ **main** functions.

Normally each application contains a small **main** function that does nothing but invoke **Tcl_Main**. **Tcl_Main** then does all the work of creating and running a **tclsh**-like application.

When it is has finished its own initialization, but before it processes commands, **Tcl_Main** calls the procedure given by the *appInitProc* argument. This procedure provides a ''hook'' for the application to perform its own initialization, such as defining application-specific commands. The procedure must have an interface that matches the type **Tcl_AppInitProc**:

typedef int Tcl_AppInitProc(Tcl_Interp ∗*interp*);

AppInitProc is almost always a pointer to **Tcl_AppInit**; for more details on this procedure, see the documentation for **Tcl_AppInit**.

KEYWORDS

application-specific initialization, command-line arguments, main program

NAME

Tcl_TraceVar, Tcl_TraceVar2, Tcl_UntraceVar, Tcl_UntraceVar2, Tcl_VarTraceInfo, Tcl_VarTraceInfo2 − monitor accesses to a variable

SYNOPSIS

#include <tcl.h>

int
Tcl_TraceVar(*interp, varName, flags, proc, clientData***)**

int
Tcl_TraceVar2(*interp, name1, name2, flags, proc, clientData***)**

Tcl_UntraceVar(*interp, varName, flags, proc, clientData***)**

Tcl_UntraceVar2(*interp, name1, name2, flags, proc, clientData***)**

ClientData
Tcl_VarTraceInfo(*interp, varName, flags, proc, prevClientData***)**

ClientData
Tcl_VarTraceInfo2(*interp, name1, name2, flags, proc, prevClientData***)**

ARGUMENTS

Tcl_Interp	∗*interp*	(in)	Interpreter containing variable.
char	∗*varName*	(in)	Name of variable. May refer to a scalar variable, to an array variable with no index, or to an array variable with a parenthesized index. If the name references an element of an array, then it must be in writable memory: Tcl will make temporary modifications to it while looking up the name.
int	*flags*	(in)	OR-ed combination of the values TCL_TRACE_READS, TCL_TRACE_WRITES, and TCL_TRACE_UNSETS, TCL_PARSE_PART1, and TCL_GLOBAL_ONLY. Not all flags are used by all procedures. See below for more information.
Tcl_VarTraceProc	∗*proc*	(in)	Procedure to invoke whenever one of the traced operations occurs.
ClientData	*clientData*	(in)	Arbitrary one-word value to pass to *proc*.
char	∗*name1*	(in)	Name of scalar or array variable (without array index).
char	∗*name2*	(in)	For a trace on an element of an array, gives the index of the element. For traces on scalar variables or on whole arrays, is NULL.
ClientData	*prevClientData*	(in)	If non-NULL, gives last value returned by **Tcl_VarTraceInfo** or **Tcl_VarTraceInfo2**, so this call will return information about next trace. If NULL, this call will return information about first trace.

DESCRIPTION

Tcl_TraceVar allows a C procedure to monitor and control access to a Tcl variable, so that the C procedure is invoked whenever the variable is read or written or unset. If the trace is created successfully then **Tcl_TraceVar** returns TCL_OK. If an error occurred (e.g. *varName* specifies an element of an array, but the actual variable isn't an array) then TCL_ERROR is returned and an error message is left in *interp->result*.

The *flags* argument to **Tcl_TraceVar** indicates when the trace procedure is to be invoked and provides information for setting up the trace. It consists of an OR-ed combination of any of the following values:

TCL_GLOBAL_ONLY
Normally, the variable will be looked up at the current level of procedure call; if this bit is set then the variable will be looked up at global level, ignoring any active procedures.

TCL_TRACE_READS
Invoke *proc* whenever an attempt is made to read the variable.

TCL_TRACE_WRITES
Invoke *proc* whenever an attempt is made to modify the variable.

TCL_TRACE_UNSETS
Invoke *proc* whenever the variable is unset. A variable may be unset either explicitly by an **unset** command, or implicitly when a procedure returns (its local variables are automatically unset) or when the interpreter is deleted (all variables are automatically unset).

Whenever one of the specified operations occurs on the variable, *proc* will be invoked. It should have arguments and result that match the type **Tcl_VarTraceProc**:

```
typedef char *Tcl_VarTraceProc(
    ClientData clientData,
    Tcl_Interp *interp,
    char *name1,
    char *name2,
    int flags);
```

The *clientData* and *interp* parameters will have the same values as those passed to **Tcl_TraceVar** when the trace was created. *ClientData* typically points to an application-specific data structure that describes what to do when *proc* is invoked. *Name1* and *name2* give the name of the traced variable in the normal two-part form (see the description of **Tcl_TraceVar2** below for details). *Flags* is an OR-ed combination of bits providing several pieces of information. One of the bits TCL_TRACE_READS, TCL_TRACE_WRITES, or TCL_TRACE_UNSETS will be set in *flags* to indicate which operation is being performed on the variable. The bit TCL_GLOBAL_ONLY will be set whenever the variable being accessed is a global one not accessible from the current level of procedure call: the trace procedure will need to pass this flag back to variable-related procedures like **Tcl_GetVar** if it attempts to access the variable. The bit TCL_TRACE_DESTROYED will be set in *flags* if the trace is about to be destroyed; this information may be useful to *proc* so that it can clean up its own internal data structures (see the section TCL_TRACE_DESTROYED below for more details). Lastly, the bit TCL_INTERP_DESTROYED will be set if the entire interpreter is being destroyed. When this bit is set, *proc* must be especially careful in the things it does (see the section TCL_INTERP_DESTROYED below). The trace procedure's return value should normally be NULL; see ERROR RETURNS below for information on other possibilities.

Tcl_UntraceVar may be used to remove a trace. If the variable specified by *interp*, *varName*, and *flags* has a trace set with *flags*, *proc*, and *clientData*, then the corresponding trace is removed. If no such trace exists, then the call to **Tcl_UntraceVar** has no effect. The same bits are valid for *flags* as for calls to **Tcl_TraceVar**.

Tcl_VarTraceInfo may be used to retrieve information about traces set on a given variable. The return value from **Tcl_VarTraceInfo** is the *clientData* associated with a particular trace. The trace must be on the

variable specified by the *interp*, *varName*, and *flags* arguments (only the TCL_GLOBAL_ONLY bit from *flags* is used; other bits are ignored) and its trace procedure must the same as the *proc* argument. If the *prevClientData* argument is NULL then the return value corresponds to the first (most recently created) matching trace, or NULL if there are no matching traces. If the *prevClientData* argument isn't NULL, then it should be the return value from a previous call to **Tcl_VarTraceInfo**. In this case, the new return value will correspond to the next matching trace after the one whose *clientData* matches *prevClientData*, or NULL if no trace matches *prevClientData* or if there are no more matching traces after it. This mechanism makes it possible to step through all of the traces for a given variable that have the same *proc*.

TWO-PART NAMES

The procedures **Tcl_TraceVar2**, **Tcl_UntraceVar2**, and **Tcl_VarTraceInfo2** are identical to **Tcl_TraceVar**, **Tcl_UntraceVar**, and **Tcl_VarTraceInfo**, respectively, except that the name of the variable consists of two parts. *Name1* gives the name of a scalar variable or array, and *name2* gives the name of an element within an array. If *name2* is NULL it means that either the variable is a scalar or the trace is to be set on the entire array rather than an individual element (see WHOLE-ARRAY TRACES below for more information). As a special case, if the flag TCL_PARSE_PART1 is specified, *name1* may contain both an array and an element name: if the name contains an open parenthesis and ends with a close parenthesis, then the value between the parentheses is treated as an element name (which can have any string value) and the characters before the first open parenthesis are treated as the name of an array variable. If the flag TCL_PARSE_PART1 is given, *name2* should be NULL since the array and element names are taken from *name1*.

ACCESSING VARIABLES DURING TRACES

During read and write traces, the trace procedure can read, write, or unset the traced variable using **Tcl_GetVar2**, **Tcl_SetVar2**, and other procedures. While *proc* is executing, traces are temporarily disabled for the variable, so that calls to **Tcl_GetVar2** and **Tcl_SetVar2** will not cause *proc* or other trace procedures to be invoked again. Disabling only occurs for the variable whose trace procedure is active; accesses to other variables will still be traced. However, if a variable is unset during a read or write trace then unset traces will be invoked.

During unset traces the variable has already been completely expunged. It is possible for the trace procedure to read or write the variable, but this will be a new version of the variable. Traces are not disabled during unset traces as they are for read and write traces, but existing traces have been removed from the variable before any trace procedures are invoked. If new traces are set by unset trace procedures, these traces will be invoked on accesses to the variable by the trace procedures.

CALLBACK TIMING

When read tracing has been specified for a variable, the trace procedure will be invoked whenever the variable's value is read. This includes **set** Tcl commands, **$**-notation in Tcl commands, and invocations of the **Tcl_GetVar** and **Tcl_GetVar2** procedures. *Proc* is invoked just before the variable's value is returned. It may modify the value of the variable to affect what is returned by the traced access. If it unsets the variable then the access will return an error just as if the variable never existed.

When write tracing has been specified for a variable, the trace procedure will be invoked whenever the variable's value is modified. This includes **set** commands, commands that modify variables as side effects (such as **catch** and **scan**), and calls to the **Tcl_SetVar** and **Tcl_SetVar2** procedures). *Proc* will be invoked after the variable's value has been modified, but before the new value of the variable has been returned. It may modify the value of the variable to override the change and to determine the value actually returned by the traced access. If it deletes the variable then the traced access will return an empty string.

When unset tracing has been specified, the trace procedure will be invoked whenever the variable is destroyed. The traces will be called after the variable has been completely unset.

WHOLE-ARRAY TRACES

If a call to **Tcl_TraceVar** or **Tcl_TraceVar2** specifies the name of an array variable without an index into the array, then the trace will be set on the array as a whole. This means that *proc* will be invoked whenever any element of the array is accessed in the ways specified by *flags*. When an array is unset, a whole-array trace will be invoked just once, with *name1* equal to the name of the array and *name2* NULL; it will not be invoked once for each element.

MULTIPLE TRACES

It is possible for multiple traces to exist on the same variable. When this happens, all of the trace procedures will be invoked on each access, in order from most-recently-created to least-recently-created. When there exist whole-array traces for an array as well as traces on individual elements, the whole-array traces are invoked before the individual-element traces. If a read or write trace unsets the variable then all of the unset traces will be invoked but the remainder of the read and write traces will be skipped.

ERROR RETURNS

Under normal conditions trace procedures should return NULL, indicating successful completion. If *proc* returns a non-NULL value it signifies that an error occurred. The return value must be a pointer to a static character string containing an error message. If a trace procedure returns an error, no further traces are invoked for the access and the traced access aborts with the given message. Trace procedures can use this facility to make variables read-only, for example (but note that the value of the variable will already have been modified before the trace procedure is called, so the trace procedure will have to restore the correct value).

The return value from *proc* is only used during read and write tracing. During unset traces, the return value is ignored and all relevant trace procedures will always be invoked.

RESTRICTIONS

A trace procedure can be called at any time, even when there is a partially-formed result in the interpreter's result area. If the trace procedure does anything that could damage this result (such as calling **Tcl_Eval**) then it must save the original values of the interpreter's **result** and **freeProc** fields and restore them before it returns.

UNDEFINED VARIABLES

It is legal to set a trace on an undefined variable. The variable will still appear to be undefined until the first time its value is set. If an undefined variable is traced and then unset, the unset will fail with an error (''no such variable''), but the trace procedure will still be invoked.

TCL_TRACE_DESTROYED FLAG

In an unset callback to *proc*, the TCL_TRACE_DESTROYED bit is set in *flags* if the trace is being removed as part of the deletion. Traces on a variable are always removed whenever the variable is deleted; the only time TCL_TRACE_DESTROYED isn't set is for a whole-array trace invoked when only a single element of an array is unset.

TCL_INTERP_DESTROYED

When an interpreter is destroyed, unset traces are called for all of its variables. The TCL_INTERP_DESTROYED bit will be set in the *flags* argument passed to the trace procedures. Trace

procedures must be extremely careful in what they do if the TCL_INTERP_DESTROYED bit is set. It is not safe for the procedures to invoke any Tcl procedures on the interpreter, since its state is partially deleted. All that trace procedures should do under these circumstances is to clean up and free their own internal data structures.

BUGS

Tcl doesn't do any error checking to prevent trace procedures from misusing the interpreter during traces with TCL_INTERP_DESTROYED set.

KEYWORDS

clientData, trace, variable

NAME

Tcl_TranslateFileName − convert file name to native form and replace tilde with home directory

SYNOPSIS

#include <tcl.h>

char ∗
Tcl_TranslateFileName(*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*	(in/out)	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 translates a file name to a form suitable for passing to the local operating system. It converts network names into native form and does tilde substitution.

If **Tcl_TranslateFileName** has to do tilde substitution or translate the name then it uses the dynamic string at ∗*bufferPtr* to hold the new string it generates. After **Tcl_TranslateFileName** returns a non-NULL result, the caller must eventually invoke **Tcl_DStringFree** to free any information placed in ∗*bufferPtr*. The caller need not know whether or not **Tcl_TranslateFileName** actually used the string; **Tcl_TranslateFileName** 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*. When an error occurs, **Tcl_TranslateFileName** frees the dynamic string itself so that the caller need not call **Tcl_DStringFree**.

The caller is responsible for making sure that *interp->result* has its default empty value when **Tcl_TranslateFileName** is invoked.

SEE ALSO

filename

KEYWORDS

file name, home directory, tilde, translate, user

NAME

Tcl_UpVar, Tcl_UpVar2 − link one variable to another

SYNOPSIS

#include <tcl.h>

int
Tcl_UpVar(*interp, frameName, sourceName, destName, flags***)**

int
Tcl_UpVar2(*interp, frameName, name1, name2, destName, flags***)**

ARGUMENTS

Tcl_Interp	∗*interp*	(in)	Interpreter containing variables; also used for error reporting.
char	∗*frameName*	(in)	Identifies the stack frame containing source variable. May have any of the forms accepted by the **upvar** command, such as **#0** or **1**.
char	∗*sourceName*	(in)	Name of source variable, in the frame given by *frameName*. May refer to a scalar variable or to an array variable with a parenthesized index.
char	∗*destName*	(in)	Name of destination variable, which is to be linked to source variable so that references to *destName* refer to the other variable. Must not currently exist except as an upvar-ed variable.
int	*flags*	(in)	Either TCL_GLOBAL_ONLY or 0; if non-zero, then *destName* is a global variable; otherwise it is a local to the current procedure (or global if no procedure is active).
char	∗*name1*	(in)	First part of source variable's name (scalar name, or name of array without array index).
char	∗*name2*	(in)	If source variable is an element of an array, gives the index of the element. For scalar source variables, is NULL.

DESCRIPTION

Tcl_UpVar and **Tcl_UpVar2** provide the same functionality as the **upvar** command: they make a link from a source variable to a destination variable, so that references to the destination are passed transparently through to the source. The name of the source variable may be specified either as a single string such as **xyx** or **a(24)** (by calling **Tcl_UpVar**) or in two parts where the array name has been separated from the element name (by calling **Tcl_UpVar2**). The destination variable name is specified in a single string; it may not be an array element.

Both procedures return either TCL_OK or TCL_ERROR, and they leave an error message in *interp->result* if an error occurs.

As with the **upvar** command, the source variable need not exist; if it does exist, unsetting it later does not destroy the link. The destination variable may exist at the time of the call, but if so it must exist as a linked variable.

KEYWORDS

linked variable, upvar, variable

NAME

Tcl_WrongNumArgs − generate standard error message for wrong number of arguments

SYNOPSIS

#include <tcl.h>

Tcl_WrongNumArgs(*interp, objc, objv, message*)

ARGUMENTS

Tcl_Interp	*interp*	(in)	Interpreter in which error will be reported: error message gets stored in its result object.
int	*objc*	(in)	Number of leading arguments from *objv* to include in error message.
Tcl_Obj	∗CONST *objv*[]	(in)	Arguments to command that had the wrong number of arguments.
char	∗*message*	(in)	Additional error information to print after leading arguments from *objv*. This typically gives the acceptable syntax of the command. This argument may be NULL.

DESCRIPTION

Tcl_WrongNumArgs is a utility procedure that is invoked by command procedures when they discover that they have received the wrong number of arguments. **Tcl_WrongNumArgs** generates a standard error message and stores it in the result object of *interp*. The message includes the *objc* initial elements of *objv* plus *message*. For example, if *objv* consists of the values **foo** and **bar**, *objc* is 1, and *message* is "**fileName count**" then *interp*'s result object will be set to the following string:

wrong # args: should be "foo fileName count"

If *objc* is 2, the result will be set to the following string:

wrong # args: should be "foo bar fileName count"

Objc is usually 1, but may be 2 or more for commands like **string** and the Tk widget commands, which use the first argument as a subcommand.

Some of the objects in the *objv* array may be abbreviations for a subcommand. The command **Tcl_GetIndexFromObj** will convert the abbreviated string object into an *indexObject*. If an error occurs in the parsing of the subcommand we would like to use the full subcommand name rather than the abbreviation. If the **Tcl_WrongNumArgs** command finds any *indexObjects* in the *objv* array it will use the full subcommand name in the error message instead of the abbreviated name that was origionally passed in. Using the above example, lets assume that *bar* is actually an abbreviation for *barfly* and the object is now an indexObject becasue it was passed to **Tcl_GetIndexFromObj**. In this case the error message would be:

wrong # args: should be "foo barfly fileName count"

SEE ALSO

Tcl_GetIndexFromObj

KEYWORDS

command, error message, wrong number of arguments

NAME

Tk_Get3DBorder, Tk_Draw3DRectangle, Tk_Fill3DRectangle, Tk_Draw3DPolygon, Tk_Fill3DPolygon, Tk_3DVerticalBevel, Tk_3DHorizontalBevel, Tk_SetBackgroundFromBorder, Tk_NameOf3DBorder, Tk_3DBorderColor, Tk_3DBorderGC, Tk_Free3DBorder − draw borders with three-dimensional appearance

SYNOPSIS

#include <tk.h>

Tk_3DBorder
Tk_Get3DBorder(*interp, tkwin, colorName***)**

void
Tk_Draw3DRectangle(*tkwin, drawable, border, x, y, width, height, borderWidth, relief***)**

void
Tk_Fill3DRectangle(*tkwin, drawable, border, x, y, width, height, borderWidth, relief***)**

void
Tk_Draw3DPolygon(*tkwin, drawable, border, pointPtr, numPoints, polyBorderWidth, leftRelief***)**

void
Tk_Fill3DPolygon(*tkwin, drawable, border, pointPtr, numPoints, polyBorderWidth, leftRelief***)**

void
Tk_3DVerticalBevel(*tkwin, drawable, border, x, y, width, height, leftBevel, relief***)**

void
Tk_3DHorizontalBevel(*tkwin, drawable, border, x, y, width, height, leftIn, rightIn, topBevel, relief***)**

void
Tk_SetBackgroundFromBorder(*tkwin, border***)**

char ∗
Tk_NameOf3DBorder(*border***)**

XColor ∗
Tk_3DBorderColor(*border***)**

GC ∗
Tk_3DBorderGC(*tkwin, border, which***)**

Tk_Free3DBorder(*border***)**

ARGUMENTS

Tcl_Interp	∗*interp*	(in)	Interpreter to use for error reporting.
Tk_Window	*tkwin*	(in)	Token for window (for all procedures except **Tk_Get3DBorder**, must be the window for which the border was allocated).
Tk_Uid	*colorName*	(in)	Textual description of color corresponding to background (flat areas). Illuminated edges will be brighter than this and shadowed edges will be darker than this.
Drawable	*drawable*	(in)	X token for window or pixmap; indicates where graphics are

			to be drawn. Must either be the X window for *tkwin* or a pixmap with the same screen and depth as *tkwin*.
Tk_3DBorder	*border*	(in)	Token for border previously allocated in call to **Tk_Get3DBorder**.
int	*x*	(in)	X-coordinate of upper-left corner of rectangle describing border or bevel, in pixels.
int	*y*	(in)	Y-coordinate of upper-left corner of rectangle describing border or bevel, in pixels.
int	*width*	(in)	Width of rectangle describing border or bevel, in pixels.
int	*height*	(in)	Height of rectangle describing border or bevel, in pixels.
int	*borderWidth*	(in)	Width of border in pixels. Positive means border is inside rectangle given by *x*, *y*, *width*, *height*, negative means border is outside rectangle.
int	*relief*	(in)	Indicates 3-D position of interior of object relative to exterior; should be TK_RELIEF_RAISED, TK_RELIEF_SUNKEN, TK_RELIEF_GROOVE, TK_RELIEF_SOLID, or TK_RELIEF_RIDGE (may also be TK_RELIEF_FLAT for **Tk_Fill3DRectangle**).
XPoint	*∗pointPtr*	(in)	Pointer to array of points describing the set of vertices in a polygon. The polygon need not be closed (it will be closed automatically if it isn't).
int	*numPoints*	(in)	Number of points at *∗pointPtr*.
int	*polyBorderWidth*	(in)	Width of border in pixels. If positive, border is drawn to left of trajectory given by *pointPtr*; if negative, border is drawn to right of trajectory. If *leftRelief* is TK_RELIEF_GROOVE or TK_RELIEF_RIDGE then the border is centered on the trajectory.
int	*leftRelief*	(in)	Height of left side of polygon's path relative to right. TK_RELIEF_RAISED means left side should appear higher and TK_RELIEF_SUNKEN means right side should appear higher; TK_RELIEF_GROOVE and TK_RELIEF_RIDGE mean the obvious things. For **Tk_Fill3DPolygon**, TK_RELIEF_FLAT may also be specified to indicate no difference in height.
int	*leftBevel*	(in)	Non-zero means this bevel forms the left side of the object; zero means it forms the right side.
int	*leftIn*	(in)	Non-zero means that the left edge of the horizontal bevel angles in, so that the bottom of the edge is farther to the right than the top. Zero means the edge angles out, so that the bottom is farther to the left than the top.
int	*rightIn*	(in)	Non-zero means that the right edge of the horizontal bevel angles in, so that the bottom of the edge is farther to the left than the top. Zero means the edge angles out, so that the bottom is farther to the right than the top.
int	*topBevel*	(in)	Non-zero means this bevel forms the top side of the object; zero means it forms the bottom side.

int	*which*	(in)	Specifies which of the border's graphics contexts is desired. Must be TK_3D_FLAT_GC, TK_3D_LIGHT_GC, or TK_3D_DARK_GC.

DESCRIPTION

These procedures provide facilities for drawing window borders in a way that produces a three-dimensional appearance. **Tk_Get3DBorder** allocates colors and Pixmaps needed to draw a border in the window given by the *tkwin* argument. The *colorName* argument indicates what colors should be used in the border. *ColorName* may be any value acceptable to **Tk_GetColor**. The color indicated by *colorName* will not actually be used in the border; it indicates the background color for the window (i.e. a color for flat surfaces). The illuminated portions of the border will appear brighter than indicated by *colorName*, and the shadowed portions of the border will appear darker than *colorName*.

Tk_Get3DBorder returns a token that may be used in later calls to **Tk_Draw3DRectangle**. If an error occurs in allocating information for the border (e.g. *colorName* isn't a legal color specifier), then NULL is returned and an error message is left in *interp->result*.

Once a border structure has been created, **Tk_Draw3DRectangle** may be invoked to draw the border. The *tkwin* argument specifies the window for which the border was allocated, and *drawable* specifies a window or pixmap in which the border is to be drawn. *Drawable* need not refer to the same window as *tkwin*, but it must refer to a compatible pixmap or window: one associated with the same screen and with the same depth as *tkwin*. The *x*, *y*, *width*, and *height* arguments define the bounding box of the border region within *drawable* (usually *x* and *y* are zero and *width* and *height* are the dimensions of the window), and *borderWidth* specifies the number of pixels actually occupied by the border. The *relief* argument indicates which of several three-dimensional effects is desired: TK_RELIEF_RAISED means that the interior of the rectangle should appear raised relative to the exterior of the rectangle, and TK_RELIEF_SUNKEN means that the interior should appear depressed. TK_RELIEF_GROOVE and TK_RELIEF_RIDGE mean that there should appear to be a groove or ridge around the exterior of the rectangle.

Tk_Fill3DRectangle is somewhat like **Tk_Draw3DRectangle** except that it first fills the rectangular area with the background color (one corresponding to the *colorName* used to create *border*). Then it calls **Tk_Draw3DRectangle** to draw a border just inside the outer edge of the rectangular area. The argument *relief* indicates the desired effect (TK_RELIEF_FLAT means no border should be drawn; all that happens is to fill the rectangle with the background color).

The procedure **Tk_Draw3DPolygon** may be used to draw more complex shapes with a three-dimensional appearance. The *pointPtr* and *numPoints* arguments define a trajectory, *polyBorderWidth* indicates how wide the border should be (and on which side of the trajectory to draw it), and *leftRelief* indicates which side of the trajectory should appear raised. **Tk_Draw3DPolygon** draws a border around the given trajectory using the colors from *border* to produce a three-dimensional appearance. If the trajectory is non-self-intersecting, the appearance will be a raised or sunken polygon shape. The trajectory may be self-intersecting, although it's not clear how useful this is.

Tk_Fill3DPolygon is to **Tk_Draw3DPolygon** what **Tk_Fill3DRectangle** is to **Tk_Draw3DRectangle**: it fills the polygonal area with the background color from *border*, then calls **Tk_Draw3DPolygon** to draw a border around the area (unless *leftRelief* is TK_RELIEF_FLAT; in this case no border is drawn).

The procedures **Tk_3DVerticalBevel** and **Tk_3DHorizontalBevel** provide lower-level drawing primitives that are used by procedures such as **Tk_Draw3DRectangle**. These procedures are also useful in their own right for drawing rectilinear border shapes. **Tk_3DVerticalBevel** draws a vertical beveled edge, such as the left or right side of a rectangle, and **Tk_3DHorizontalBevel** draws a horizontal beveled edge, such as the top or bottom of a rectangle. Each procedure takes *x*, *y*, *width*, and *height* arguments that describe the rectangular area of the beveled edge (e.g., *width* is the border width for **Tk_3DVerticalBevel**). The *leftBorder* and *topBorder* arguments indicate the position of the border relative to the ''inside'' of the object, and *relief*

indicates the relief of the inside of the object relative to the outside. **Tk_3DVerticalBevel** just draws a rectangular region. **Tk_3DHorizontalBevel** draws a trapezoidal region to generate mitered corners; it should be called after **Tk_3DVerticalBevel** (otherwise **Tk_3DVerticalBevel** will overwrite the mitering in the corner). The *leftIn* and *rightIn* arguments to **Tk_3DHorizontalBevel** describe the mitering at the corners; a value of 1 means that the bottom edge of the trapezoid will be shorter than the top, 0 means it will be longer. For example, to draw a rectangular border the top bevel should be drawn with 1 for both *leftIn* and *rightIn*, and the bottom bevel should be drawn with 0 for both arguments.

The procedure **Tk_SetBackgroundFromBorder** will modify the background pixel and/or pixmap of *tkwin* to produce a result compatible with *border*. For color displays, the resulting background will just be the color given by the *colorName* argument passed to **Tk_Get3DBorder** when *border* was created; for monochrome displays, the resulting background will be a light stipple pattern, in order to distinguish the background from the illuminated portion of the border.

Given a token for a border, the procedure **Tk_NameOf3DBorder** will return the *colorName* string that was passed to **Tk_Get3DBorder** to create the border.

The procedure **Tk_3DBorderColor** returns the XColor structure that will be used for flat surfaces drawn for its *border* argument by procedures like **Tk_Fill3DRectangle**. The return value corresponds to the *colorName* passed to **Tk_Get3DBorder**. The XColor, and its associated pixel value, will remain allocated as long as *border* exists.

The procedure **Tk_3DBorderGC** returns one of the X graphics contexts that are used to draw the border. The argument *which* selects which one of the three possible GC's: TK_3D_FLAT_GC returns the context used for flat surfaces, TK_3D_LIGHT_GC returns the context for light shadows, and TK_3D_DARK_GC returns the context for dark shadows.

When a border is no longer needed, **Tk_Free3DBorder** should be called to release the resources associated with the border. There should be exactly one call to **Tk_Free3DBorder** for each call to **Tk_Get3DBorder**.

KEYWORDS

3D, background, border, color, depressed, illumination, polygon, raised, shadow, three-dimensional effect

NAME

Tk_CreateBindingTable, Tk_DeleteBindingTable, Tk_CreateBinding, Tk_DeleteBinding, Tk_GetBinding, Tk_GetAllBindings, Tk_DeleteAllBindings, Tk_BindEvent − invoke scripts in response to X events

SYNOPSIS

#include <tk.h>

Tk_BindingTable
Tk_CreateBindingTable(*interp***)**

Tk_DeleteBindingTable(*bindingTable***)**

unsigned long
Tk_CreateBinding(*interp, bindingTable, object, eventString, script, append***)**

int
Tk_DeleteBinding(*interp, bindingTable, object, eventString***)**

char *
Tk_GetBinding(*interp, bindingTable, object, eventString***)**

Tk_GetAllBindings(*interp, bindingTable, object***)**

Tk_DeleteAllBindings(*bindingTable, object***)**

Tk_BindEvent(*bindingTable, eventPtr, tkwin, numObjects, objectPtr***)**

ARGUMENTS

Tcl_Interp	**interp*	(in)	Interpreter to use when invoking bindings in binding table. Also used for returning results and errors from binding procedures.
Tk_BindingTable	*bindingTable*	(in)	Token for binding table; must have been returned by some previous call to **Tk_CreateBindingTable**.
ClientData	*object*	(in)	Identifies object with which binding is associated.
char	**eventString*	(in)	String describing event sequence.
char	**script*	(in)	Tcl script to invoke when binding triggers.
int	*append*	(in)	Non-zero means append *script* to existing script for binding, if any; zero means replace existing script with new one.
XEvent	**eventPtr*	(in)	X event to match against bindings in *bindingTable*.
Tk_Window	*tkwin*	(in)	Identifier for any window on the display where the event occurred. Used to find display-related information such as key maps.
int	*numObjects*	(in)	Number of object identifiers pointed to by *objectPtr*.
ClientData	**objectPtr*	(in)	Points to an array of object identifiers: bindings will be considered for each of these objects in order from first to last.

DESCRIPTION

These procedures provide a general-purpose mechanism for creating and invoking bindings. Bindings are organized in terms of *binding tables*. A binding table consists of a collection of bindings plus a history of recent events. Within a binding table, bindings are associated with *objects*. The meaning of an object is defined by clients of the binding package. For example, Tk keeps uses one binding table to hold all of the bindings created by the **bind** command. For this table, objects are pointers to strings such as window names, class names, or other binding tags such as **all**. Tk also keeps a separate binding table for each canvas widget, which manages bindings created by the canvas's **bind** widget command; within this table, an object is either a pointer to the internal structure for a canvas item or a Tk_Uid identifying a tag.

The procedure **Tk_CreateBindingTable** creates a new binding table and associates *interp* with it (when bindings in the table are invoked, the scripts will be evaluated in *interp*). **Tk_CreateBindingTable** returns a token for the table, which must be used in calls to other procedures such as **Tk_CreateBinding** or **Tk_BindEvent**.

Tk_DeleteBindingTable frees all of the state associated with a binding table. Once it returns the caller should not use the *bindingTable* token again.

Tk_CreateBinding adds a new binding to an existing table. The *object* argument identifies the object with which the binding is to be associated, and it may be any one-word value. Typically it is a pointer to a string or data structure. The *eventString* argument identifies the event or sequence of events for the binding; see the documentation for the **bind** command for a description of its format. *script* is the Tcl script to be evaluated when the binding triggers. *append* indicates what to do if there already exists a binding for *object* and *eventString*: if *append* is zero then *script* replaces the old script; if *append* is non-zero then the new script is appended to the old one. **Tk_CreateBinding** returns an X event mask for all the events associated with the bindings. This information may be useful to invoke **XSelectInput** to select relevant events, or to disallow the use of certain events in bindings. If an error occurred while creating the binding (e.g., *eventString* refers to a non-existent event), then 0 is returned and an error message is left in *interp->result*.

Tk_DeleteBinding removes from *bindingTable* the binding given by *object* and *eventString*, if such a binding exists. **Tk_DeleteBinding** always returns TCL_OK. In some cases it may reset *interp->result* to the default empty value.

Tk_GetBinding returns a pointer to the script associated with *eventString* and *object* in *bindingTable*. If no such binding exists then NULL is returned and an error message is left in *interp->result*.

Tk_GetAllBindings returns in *interp->result* a list of all the event strings for which there are bindings in *bindingTable* associated with *object*. If there are no bindings for *object* then an empty string is returned in *interp->result*.

Tk_DeleteAllBindings deletes all of the bindings in *bindingTable* that are associated with *object*.

Tk_BindEvent is called to process an event. It makes a copy of the event in an internal history list associated with the binding table, then it checks for bindings that match the event. **Tk_BindEvent** processes each of the objects pointed to by *objectPtr* in turn. For each object, it finds all the bindings that match the current event history, selects the most specific binding using the priority mechanism described in the documentation for **bind**, and invokes the script for that binding. If there are no matching bindings for a particular object, then the object is skipped. **Tk_BindEvent** continues through all of the objects, handling exceptions such as errors, **break**, and **continue** as described in the documentation for **bind**.

KEYWORDS

binding, event, object, script

NAME

Tk_CanvasPsY, Tk_CanvasPsBitmap, Tk_CanvasPsColor, Tk_CanvasPsFont, Tk_CanvasPsPath, Tk_CanvasPsStipple − utility procedures for generating Postscript for canvases

SYNOPSIS

#include <tk.h>

double
Tk_CanvasPsY(*canvas, canvasY*)

int
Tk_CanvasPsBitmap(*interp, canvas, bitmap, x, y, width, height*)

int
Tk_CanvasPsColor(*interp, canvas, colorPtr*)

int
Tk_CanvasPsFont(*interp, canvas, fontStructPtr*)

Tk_CanvasPsPath(*interp, canvas, coordPtr, numPoints*)

int
Tk_CanvasPsStipple(*interp, canvas, bitmap*)

ARGUMENTS

Tk_Canvas	*canvas*	(in)	A token that identifies a canvas widget for which Postscript is being generated.
double	*canvasY*	(in)	Y-coordinate in the space of the canvas.
Tcl_Interp	∗*interp*	(in/out)	A Tcl interpreter; Postscript is appended to its result, or the result may be replaced with an error message.
Pixmap	*bitmap*	(in)	Bitmap to use for generating Postscript.
int	*x*	(in)	X-coordinate within *bitmap* of left edge of region to output.
int	*y*	(in)	Y-coordinate within *bitmap* of top edge of region to output.
int	*width*	(in)	Width of region of bitmap to output, in pixels.
int	*height*	(in)	Height of region of bitmap to output, in pixels.
XColor	∗*colorPtr*	(in)	Information about color value to set in Postscript.
XFontStruct	∗*fontStructPtr*	(in)	Font for which Postscript is to be generated.
double	∗*coordPtr*	(in)	Pointer to an array of coordinates for one or more points specified in canvas coordinates. The order of values in *coordPtr* is x1, y1, x2, y2, x3, y3, and so on.
int	*numPoints*	(in)	Number of points at *coordPtr*.

DESCRIPTION

These procedures are called by canvas type managers to carry out common functions related to generating Postscript. Most of the procedures take a *canvas* argument, which refers to a canvas widget for which Postscript is being generated.

Tk_CanvasY takes as argument a y-coordinate in the space of a canvas and returns the value that should be used for that point in the Postscript currently being generated for *canvas*. Y coordinates require transformation because Postscript uses an origin at the lower-left corner whereas X uses an origin at the upper-left corner. Canvas x coordinates can be used directly in Postscript without transformation.

Tk_CanvasPsBitmap generates Postscript to describe a region of a bitmap. The Postscript is generated in proper image data format for Postscript, i.e., as data between angle brackets, one bit per pixel. The Postscript is appended to *interp->result* and TCL_OK is returned unless an error occurs, in which case TCL_ERROR is returned and *interp->result* is overwritten with an error message.

Tk_CanvasPsColor generates Postscript to set the current color to correspond to its *colorPtr* argument, taking into account any color map specified in the **postscript** command. It appends the Postscript to *interp->result* and returns TCL_OK unless an error occurs, in which case TCL_ERROR is returned and *interp->result* is overwritten with an error message.

Tk_CanvasPsFont generates Postscript that sets the current font to match *fontStructPtr* as closely as possible. **Tk_CanvasPsFont** takes into account any font map specified in the **postscript** command, and it does the best it can at mapping X fonts to Postscript fonts. It appends the Postscript to *interp->result* and returns TCL_OK unless an error occurs, in which case TCL_ERROR is returned and *interp->result* is overwritten with an error message.

Tk_CanvasPsPath generates Postscript to set the current path to the set of points given by *coordPtr* and *numPoints*. It appends the resulting Postscript to *interp->result*.

Tk_CanvasPsStipple generates Postscript that will fill the current path in stippled fashion. It uses *bitmap* as the stipple pattern and the current Postscript color; ones in the stipple bitmap are drawn in the current color, and zeroes are not drawn at all. The Postscript is appended to *interp->result* and TCL_OK is returned, unless an error occurs, in which case TCL_ERROR is returned and *interp->result* is overwritten with an error message.

KEYWORDS

bitmap, canvas, color, font, path, Postscript, stipple

NAME

Tk_CanvasTkwin, Tk_CanvasGetCoord, Tk_CanvasDrawableCoords, Tk_CanvasSetStippleOrigin, Tk_CanvasWindowCoords, Tk_CanvasEventuallyRedraw, Tk_CanvasTagsOption − utility procedures for canvas type managers

SYNOPSIS

#include <tk.h>

Tk_Window
Tk_CanvasTkwin(*canvas*)

int
Tk_CanvasGetCoord(*interp, canvas, string, doublePtr*)

Tk_CanvasDrawableCoords(*canvas, x, y, drawableXPtr, drawableYPtr*)

Tk_CanvasSetStippleOrigin(*canvas, gc*)

Tk_CanvasWindowCoords(*canvas, x, y, screenXPtr, screenYPtr*)

Tk_CanvasEventuallyRedraw(*canvas, x1, y1, x2, y2*)

Tk_OptionParseProc ∗**Tk_CanvasTagsParseProc**;

Tk_OptionPrintProc ∗**Tk_CanvasTagsPrintProc**;

ARGUMENTS

Tk_Canvas	*canvas*	(in)	A token that identifies a canvas widget.
Tcl_Interp	∗*interp*	(in/out)	Interpreter to use for error reporting.
char	∗*string*	(in)	Textual description of a canvas coordinate.
double	∗*doublePtr*	(out)	Points to place to store a converted coordinate.
double	*x*	(in)	An x coordinate in the space of the canvas.
double	*y*	(in)	A y coordinate in the space of the canvas.
short	∗*drawableXPtr*	(out)	Pointer to a location in which to store an x coordinate in the space of the drawable currently being used to redisplay the canvas.
short	∗*drawableYPtr*	(out)	Pointer to a location in which to store a y coordinate in the space of the drawable currently being used to redisplay the canvas.
GC	*gc*	(out)	Graphics context to modify.
short	∗*screenXPtr*	(out)	Points to a location in which to store the screen coordinate in the canvas window that corresponds to *x*.
short	∗*screenYPtr*	(out)	Points to a location in which to store the screen coordinate in the canvas window that corresponds to *y*.
int	*x1*	(in)	Left edge of the region that needs redisplay. Only pixels at or to the right of this coordinate need to be redisplayed.
int	*y1*	(in)	Top edge of the region that needs redisplay. Only pixels at or below this coordinate need to be redisplayed.

int	*x2*	(in)	Right edge of the region that needs redisplay. Only pixels to the left of this coordinate need to be redisplayed.
int	*y2*	(in)	Bottom edge of the region that needs redisplay. Only pixels above this coordinate need to be redisplayed.

DESCRIPTION

These procedures are called by canvas type managers to perform various utility functions.

Tk_CanvasTkwin returns the Tk_Window associated with a particular canvas.

Tk_CanvasGetCoord translates a string specification of a coordinate (such as **2p** or **1.6c**) into a double-precision canvas coordinate. If *string* is a valid coordinate description then **Tk_CanvasGetCoord** stores the corresponding canvas coordinate at ∗*doublePtr* and returns TCL_OK. Otherwise it stores an error message in *interp->result* and returns TCL_ERROR.

Tk_CanvasDrawableCoords is called by type managers during redisplay to compute where to draw things. Given *x* and *y* coordinates in the space of the canvas, **Tk_CanvasDrawableCoords** computes the corresponding pixel in the drawable that is currently being used for redisplay; it returns those coordinates in ∗*drawableXPtr* and ∗*drawableYPtr*. This procedure should not be invoked except during redisplay.

Tk_CanvasSetStippleOrigin is also used during redisplay. It sets the stipple origin in *gc* so that stipples drawn with *gc* in the current offscreen pixmap will line up with stipples drawn with origin (0,0) in the canvas's actual window. **Tk_CanvasSetStippleOrigin** is needed in order to guarantee that stipple patterns line up properly when the canvas is redisplayed in small pieces. Redisplays are carried out in double-buffered fashion where a piece of the canvas is redrawn in an offscreen pixmap and then copied back onto the screen. In this approach the stipple origins in graphics contexts need to be adjusted during each redisplay to compensate for the position of the off-screen pixmap relative to the window. If an item is being drawn with stipples, its type manager typically calls **Tk_CanvasSetStippleOrigin** just before using *gc* to draw something; after it is finished drawing, the type manager calls **XSetTSOrigin** to restore the origin in *gc* back to (0,0) (the restore is needed because graphics contexts are shared, so they cannot be modified permanently).

Tk_CanvasWindowCoords is similar to **Tk_CanvasDrawableCoords** except that it returns coordinates in the canvas's window on the screen, instead of coordinates in an off-screen pixmap.

Tk_CanvasEventuallyRedraw may be invoked by a type manager to inform Tk that a portion of a canvas needs to be redrawn. The *x1*, *y1*, *x2*, and *y2* arguments specify the region that needs to be redrawn, in canvas coordinates. Type managers rarely need to invoke **Tk_CanvasEventuallyRedraw**, since Tk can normally figure out when an item has changed and make the redisplay request on its behalf (this happens, for example whenever Tk calls a *configureProc* or *scaleProc*). The only time that a type manager needs to call **Tk_CanvasEventuallyRedraw** is if an item has changed on its own without being invoked through one of the procedures in its Tk_ItemType; this could happen, for example, in an image item if the image is modified using image commands.

Tk_CanvasTagsParseProc and **Tk_CanvasTagsPrintProc** are procedures that handle the **−tags** option for canvas items. The code of a canvas type manager won't call these procedures directly, but will use their addresses to create a **Tk_CustomOption** structure for the **−tags** option. The code typically looks like this:

```
static Tk_CustomOption tagsOption = {Tk_CanvasTagsParseProc,
    Tk_CanvasTagsPrintProc, (ClientData) NULL
};

static Tk_ConfigSpec configSpecs[] = {
    ...
    {TK_CONFIG_CUSTOM, "−tags", (char ∗) NULL, (char ∗) NULL,
```

```
        (char *) NULL, 0, TK_CONFIG_NULL_OK, &tagsOption},
    ...
};
```

KEYWORDS

canvas, focus, item type, redisplay, selection, type manager

NAME

Tk_CanvasTextInfo − additional information for managing text items in canvases

SYNOPSIS

#include <tk.h>

Tk_CanvasTextInfo ∗
Tk_CanvasGetTextInfo(*canvas*)

ARGUMENTS

Tk_Canvas	*canvas*	(in)	A token that identifies a particular canvas widget.

DESCRIPTION

Textual canvas items are somewhat more complicated to manage than other items, due to things like the selection and the input focus. **Tk_CanvasGetTextInfo** may be invoked by a type manager to obtain additional information needed for items that display text. The return value from **Tk_CanvasGetTextInfo** is a pointer to a structure that is shared between Tk and all the items that display text. The structure has the following form:

```
typedef struct Tk_CanvasTextInfo {
    Tk_3DBorder selBorder;
    int selBorderWidth;
    XColor *selFgColorPtr;
    Tk_Item *selItemPtr;
    int selectFirst;
    int selectLast;
    Tk_Item *anchorItemPtr;
    int selectAnchor;
    Tk_3DBorder insertBorder;
    int insertWidth;
    int insertBorderWidth;
    Tk_Item *focusItemPtr;
    int gotFocus;
    int cursorOn;
} Tk_CanvasTextInfo;
```

The **selBorder** field identifies a Tk_3DBorder that should be used for drawing the background under selected text. *selBorderWidth* gives the width of the raised border around selected text, in pixels. *selFgColorPtr* points to an XColor that describes the foreground color to be used when drawing selected text. *selItemPtr* points to the item that is currently selected, or NULL if there is no item selected or if the canvas doesn't have the selection. *selectFirst* and *selectLast* give the indices of the first and last selected characters in *selItemPtr*, as returned by the *indexProc* for that item. *anchorItemPtr* points to the item that currently has the selection anchor; this is not necessarily the same as *selItemPtr*. *selectAnchor* is an index that identifies the anchor position within *anchorItemPtr*. *insertBorder* contains a Tk_3DBorder to use when drawing the insertion cursor; *insertWidth* gives the total width of the insertion cursor in pixels, and *insertBorderWidth* gives the width of the raised border around the insertion cursor. *focusItemPtr* identifies the item that currently has the input focus, or NULL if there is no such item. *gotFocus* is 1 if the canvas widget has the input focus and 0 otherwise. *cursorOn* is 1 if the insertion cursor should be drawn in *focusItemPtr* and 0 if it should not be drawn; this field is toggled on and off by Tk to make the cursor blink.

The structure returned by **Tk_CanvasGetTextInfo** is shared between Tk and the type managers; typically the type manager calls **Tk_CanvasGetTextInfo** once when an item is created and then saves the pointer in the item's record. Tk will update information in the Tk_CanvasTextInfo; for example, a **configure** widget command might change the *selBorder* field, or a **select** widget command might change the *selectFirst* field,

or Tk might change *cursorOn* in order to make the insertion cursor flash on and off during successive redisplays.

Type managers should treat all of the fields of the Tk_CanvasTextInfo structure as read-only, except for *selItemPtr*, *selectFirst*, *selectLast*, and *selectAnchor*. Type managers may change *selectFirst*, *selectLast*, and *selectAnchor* to adjust for insertions and deletions in the item (but only if the item is the current owner of the selection or anchor, as determined by *selItemPtr* or *anchorItemPtr*). If all of the selected text in the item is deleted, the item should set *selItemPtr* to NULL to indicate that there is no longer a selection.

KEYWORDS

canvas, focus, insertion cursor, selection, selection anchor, text

NAME

Tk_ClipboardClear, Tk_ClipboardAppend − Manage the clipboard

SYNOPSIS

#include <tk.h>

int
Tk_ClipboardClear(*interp, tkwin*)

int
Tk_ClipboardAppend(*interp, tkwin, target, format, buffer*)

ARGUMENTS

Tcl_Interp	∗*interp*	(in)	Interpreter to use for reporting errors.
Tk_Window	*tkwin*	(in)	Window that determines which display's clipboard to manipulate.
Atom	*target*	(in)	Conversion type for this clipboard item; has same meaning as *target* argument to **Tk_CreateSelHandler**.
Atom	*format*	(in)	Representation to use when data is retrieved; has same meaning as *format* argument to **Tk_CreateSelHandler**.
char	∗*buffer*	(in)	Null terminated string containing the data to be appended to the clipboard.

DESCRIPTION

These two procedures manage the clipboard for Tk. The clipboard is typically managed by calling **Tk_ClipboardClear** once, then calling **Tk_ClipboardAppend** to add data for any number of targets.

Tk_ClipboardClear claims the CLIPBOARD selection and frees any data items previously stored on the clipboard in this application. It normally returns TCL_OK, but if an error occurs it returns TCL_ERROR and leaves an error message in *interp->result*. **Tk_ClipboardClear** must be called before a sequence of **Tk_ClipboardAppend** calls can be issued.

Tk_ClipboardAppend appends a buffer of data to the clipboard. The first buffer for a given *target* determines the *format* for that *target*. Any successive appends for that *target* must have the same format or an error will be returned. **Tk_ClipboardAppend** returns TCL_OK if the buffer is successfully copied onto the clipboard. If the clipboard is not currently owned by the application, either because **Tk_ClipboardClear** has not been called or because ownership of the clipboard has changed since the last call to **Tk_ClipboardClear**, **Tk_ClipboardAppend** returns TCL_ERROR and leaves an error message in *interp->result*.

In order to guarantee atomicity, no event handling should occur between **Tk_ClipboardClear** and the following **Tk_ClipboardAppend** calls (otherwise someone could retrieve a partially completed clipboard or claim ownership away from this application).

Tk_ClipboardClear may invoke callbacks, including arbitrary Tcl scripts, as a result of losing the CLIPBOARD selection, so any calling function should take care to be reentrant at the point **Tk_ClipboardClear** is invoked.

KEYWORDS

append, clipboard, clear, format, type

NAME

Tk_ClearSelection − Deselect a selection

SYNOPSIS

#include <tk.h>

Tk_ClearSelection(*tkwin, selection*)

ARGUMENTS

Tk_Window	*tkwin*	(in)	The selection will be cleared from the display containing this window.
Atom	*selection*	(in)	The name of selection to be cleared.

DESCRIPTION

Tk_ClearSelection cancels the selection specified by the atom *selection* for the display containing *tkwin*. The selection need not be in *tkwin* itself or even in *tkwin*'s application. If there is a window anywhere on *tkwin*'s display that owns *selection*, the window will be notified and the selection will be cleared. If there is no owner for *selection* on the display, then the procedure has no effect.

KEYWORDS

clear, selection

NAME

Tk_ConfigureWidget, Tk_Offset, Tk_ConfigureInfo, Tk_ConfigureValue, Tk_FreeOptions − process configuration options for widgets

SYNOPSIS

#include <tk.h>

int
Tk_ConfigureWidget(*interp, tkwin, specs, argc, argv, widgRec, flags***)**

int
Tk_Offset(*type, field***)**

int
Tk_ConfigureInfo(*interp, tkwin, specs, widgRec, argvName, flags***)**

int

Tk_FreeOptions(*specs, widgRec, display, flags***)**

ARGUMENTS

Tcl_Interp	∗*interp*	(in)	Interpreter to use for returning error messages.
Tk_Window	*tkwin*	(in)	Window used to represent widget (needed to set up X resources).
Tk_ConfigSpec	∗*specs*	(in)	Pointer to table specifying legal configuration options for this widget.
int	*argc*	(in)	Number of arguments in *argv*.
char	∗∗*argv*	(in)	Command-line options for configuring widget.
char	∗*widgRec*	(in/out)	Points to widget record structure. Fields in this structure get modified by **Tk_ConfigureWidget** to hold configuration information.
int	*flags*	(in)	If non-zero, then it specifies an OR-ed combination of flags that control the processing of configuration information. TK_CONFIG_ARGV_ONLY causes the option database and defaults to be ignored, and flag bits TK_CONFIG_USER_BIT and higher are used to selectively disable entries in *specs*.
type name	*type*	(in)	The name of the type of a widget record.
field name	*field*	(in)	The name of a field in records of type *type*.
char	∗*argvName*	(in)	The name used on Tcl command lines to refer to a particular option (e.g. when creating a widget or invoking the **configure** widget command). If non-NULL, then information is returned only for this option. If NULL, then information is returned for all available options.
Display	∗*display*	(in)	Display containing widget whose record is being freed; needed in order to free up resources.

DESCRIPTION

Tk_ConfigureWidget is called to configure various aspects of a widget, such as colors, fonts, border width, etc. It is intended as a convenience procedure to reduce the amount of code that must be written in individual widget managers to handle configuration information. It is typically invoked when widgets are created, and again when the **configure** command is invoked for a widget. Although intended primarily for widgets, **Tk_ConfigureWidget** can be used in other situations where *argc*-*argv* information is to be used to fill in a record structure, such as configuring graphical elements for a canvas widget or entries of a menu.

Tk_ConfigureWidget processes a table specifying the configuration options that are supported (*specs*) and a collection of command-line arguments (*argc* and *argv*) to fill in fields of a record (*widgRec*). It uses the option database and defaults specified in *specs* to fill in fields of *widgRec* that are not specified in *argv*. **Tk_ConfigureWidget** normally returns the value TCL_OK; in this case it does not modify *interp*. If an error occurs then TCL_ERROR is returned and **Tk_ConfigureWidget** will leave an error message in *interp->result* in the standard Tcl fashion. In the event of an error return, some of the fields of *widgRec* could already have been set, if configuration information for them was successfully processed before the error occurred. The other fields will be set to reasonable initial values so that **Tk_FreeOptions** can be called for cleanup.

The *specs* array specifies the kinds of configuration options expected by the widget. Each of its entries specifies one configuration option and has the following structure:

```
typedef struct {
    int type;
    char *argvName;
    char *dbName;
    char *dbClass;
    char *defValue;
    int offset;
    int specFlags;
    Tk_CustomOption *customPtr;
} Tk_ConfigSpec;
```

The *type* field indicates what type of configuration option this is (e.g. TK_CONFIG_COLOR for a color value, or TK_CONFIG_INT for an integer value). The *type* field indicates how to use the value of the option (more on this below). The *argvName* field is a string such as ‘‘−font’’ or ‘‘−bg’’, which is compared with the values in *argv* (if *argvName* is NULL it means this is a grouped entry; see GROUPED ENTRIES below). The *dbName* and *dbClass* fields are used to look up a value for this option in the option database. The *defValue* field specifies a default value for this configuration option if no value is specified in either *argv* or the option database. *Offset* indicates where in *widgRec* to store information about this option, and *specFlags* contains additional information to control the processing of this configuration option (see FLAGS below). The last field, *customPtr*, is only used if *type* is TK_CONFIG_CUSTOM; see CUSTOM OPTION TYPES below.

Tk_ConfigureWidget first processes *argv* to see which (if any) configuration options are specified there. *Argv* must contain an even number of fields; the first of each pair of fields must match the *argvName* of some entry in *specs* (unique abbreviations are acceptable), and the second field of the pair contains the value for that configuration option. If there are entries in *spec* for which there were no matching entries in *argv*, **Tk_ConfigureWidget** uses the *dbName* and *dbClass* fields of the *specs* entry to probe the option database; if a value is found, then it is used as the value for the option. Finally, if no entry is found in the option database, the *defValue* field of the *specs* entry is used as the value for the configuration option. If the *defValue* is NULL, or if the TK_CONFIG_DONT_SET_DEFAULT bit is set in *flags*, then there is no default value and this *specs* entry will be ignored if no value is specified in *argv* or the option database.

Once a string value has been determined for a configuration option, **Tk_ConfigureWidget** translates the string value into a more useful form, such as a color if *type* is TK_CONFIG_COLOR or an integer if *type* is TK_CONFIG_INT. This value is then stored in the record pointed to by *widgRec*. This record is assumed to contain information relevant to the manager of the widget; its exact type is unknown to

Tk_ConfigureWidget. The *offset* field of each *specs* entry indicates where in *widgRec* to store the information about this configuration option. You should use the **Tk_Offset** macro to generate *offset* values (see below for a description of **Tk_Offset**). The location indicated by *widgRec* and *offset* will be referred to as the ''target'' in the descriptions below.

The *type* field of each entry in *specs* determines what to do with the string value of that configuration option. The legal values for *type*, and the corresponding actions, are:

TK_CONFIG_ACTIVE_CURSOR
The value must be an ASCII string identifying a cursor in a form suitable for passing to **Tk_GetCursor**. The value is converted to a **Tk_Cursor** by calling **Tk_GetCursor** and the result is stored in the target. In addition, the resulting cursor is made the active cursor for *tkwin* by calling **XDefineCursor**. If TK_CONFIG_NULL_OK is specified in *specFlags* then the value may be an empty string, in which case the target and *tkwin*'s active cursor will be set to **None**. If the previous value of the target wasn't **None**, then it is freed by passing it to **Tk_FreeCursor**.

TK_CONFIG_ANCHOR
The value must be an ASCII string identifying an anchor point in one of the ways accepted by **Tk_GetAnchor**. The string is converted to a **Tk_Anchor** by calling **Tk_GetAnchor** and the result is stored in the target.

TK_CONFIG_BITMAP
The value must be an ASCII string identifying a bitmap in a form suitable for passing to **Tk_GetBitmap**. The value is converted to a **Pixmap** by calling **Tk_GetBitmap** and the result is stored in the target. If TK_CONFIG_NULL_OK is specified in *specFlags* then the value may be an empty string, in which case the target is set to **None**. If the previous value of the target wasn't **None**, then it is freed by passing it to **Tk_FreeBitmap**.

TK_CONFIG_BOOLEAN
The value must be an ASCII string specifying a boolean value. Any of the values ''true'', ''yes'', ''on'', or ''1'', or an abbreviation of one of these values, means true; any of the values ''false'', ''no'', ''off'', or ''0'', or an abbreviation of one of these values, means false. The target is expected to be an integer; for true values it will be set to 1 and for false values it will be set to 0.

TK_CONFIG_BORDER
The value must be an ASCII string identifying a border color in a form suitable for passing to **Tk_Get3DBorder**. The value is converted to a (**Tk_3DBorder** *) by calling **Tk_Get3DBorder** and the result is stored in the target. If TK_CONFIG_NULL_OK is specified in *specFlags* then the value may be an empty string, in which case the target will be set to NULL. If the previous value of the target wasn't NULL, then it is freed by passing it to **Tk_Free3DBorder**.

TK_CONFIG_CAP_STYLE
The value must be an ASCII string identifying a cap style in one of the ways accepted by **Tk_GetCapStyle**. The string is converted to an integer value corresponding to the cap style by calling **Tk_GetCapStyle** and the result is stored in the target.

TK_CONFIG_COLOR
The value must be an ASCII string identifying a color in a form suitable for passing to **Tk_GetColor**. The value is converted to an (**XColor** *) by calling **Tk_GetColor** and the result is stored in the target. If TK_CONFIG_NULL_OK is specified in *specFlags* then the value may be an empty string, in which case the target will be set to **None**. If the previous value of the target wasn't NULL, then it is freed by passing it to **Tk_FreeColor**.

TK_CONFIG_CURSOR
This option is identical to **TK_CONFIG_ACTIVE_CURSOR** except that the new cursor is not made the active one for *tkwin*.

TK_CONFIG_CUSTOM

This option allows applications to define new option types. The *customPtr* field of the entry points to a structure defining the new option type. See the section CUSTOM OPTION TYPES below for details.

TK_CONFIG_DOUBLE
The value must be an ASCII floating-point number in the format accepted by **strtol**. The string is converted to a **double** value, and the value is stored in the target.

TK_CONFIG_END
Marks the end of the table. The last entry in *specs* must have this type; all of its other fields are ignored and it will never match any arguments.

TK_CONFIG_FONT
The value must be an ASCII string identifying a font in a form suitable for passing to **Tk_GetFontStruct**. The value is converted to an (**XFontStruct** *) by calling **Tk_GetFontStruct** and the result is stored in the target. If TK_CONFIG_NULL_OK is specified in *specFlags* then the value may be an empty string, in which case the target will be set to NULL. If the previous value of the target wasn't NULL, then it is freed by passing it to **Tk_FreeFontStruct**.

TK_CONFIG_INT
The value must be an ASCII integer string in the format accepted by **strtol** (e.g. ''0'' and ''0x'' prefixes may be used to specify octal or hexadecimal numbers, respectively). The string is converted to an integer value and the integer is stored in the target.

TK_CONFIG_JOIN_STYLE
The value must be an ASCII string identifying a join style in one of the ways accepted by **Tk_GetJoinStyle**. The string is converted to an integer value corresponding to the join style by calling **Tk_GetJoinStyle** and the result is stored in the target.

TK_CONFIG_JUSTIFY
The value must be an ASCII string identifying a justification method in one of the ways accepted by **Tk_GetJustify**. The string is converted to a **Tk_Justify** by calling **Tk_GetJustify** and the result is stored in the target.

TK_CONFIG_MM
The value must specify a screen distance in one of the forms acceptable to **Tk_GetScreenMM**. The string is converted to double-precision floating-point distance in millimeters and the value is stored in the target.

TK_CONFIG_PIXELS
The value must specify screen units in one of the forms acceptable to **Tk_GetPixels**. The string is converted to an integer distance in pixels and the value is stored in the target.

TK_CONFIG_RELIEF
The value must be an ASCII string identifying a relief in a form suitable for passing to **Tk_GetRelief**. The value is converted to an integer relief value by calling **Tk_GetRelief** and the result is stored in the target.

TK_CONFIG_STRING
A copy of the value is made by allocating memory space with **malloc** and copying the value into the dynamically-allocated space. A pointer to the new string is stored in the target. If TK_CONFIG_NULL_OK is specified in *specFlags* then the value may be an empty string, in which case the target will be set to NULL. If the previous value of the target wasn't NULL, then it is freed by passing it to **free**.

TK_CONFIG_SYNONYM
This *type* value identifies special entries in *specs* that are synonyms for other entries. If an *argv* value matches the *argvName* of a TK_CONFIG_SYNONYM entry, the entry isn't used directly.

Instead, **Tk_ConfigureWidget** searches *specs* for another entry whose *argvName* is the same as the *dbName* field in the TK_CONFIG_SYNONYM entry; this new entry is used just as if its *argvName* had matched the *argv* value. The synonym mechanism allows multiple *argv* values to be used for a single configuration option, such as ‘‘−background’’ and ‘‘−bg’’.

TK_CONFIG_UID
The value is translated to a **Tk_Uid** (by passing it to **Tk_GetUid**). The resulting value is stored in the target. If TK_CONFIG_NULL_OK is specified in *specFlags* and the value is an empty string then the target will be set to NULL.

TK_CONFIG_WINDOW
The value must be a window path name. It is translated to a **Tk_Window** token and the token is stored in the target.

GROUPED ENTRIES

In some cases it is useful to generate multiple resources from a single configuration value. For example, a color name might be used both to generate the background color for a widget (using TK_CONFIG_COLOR) and to generate a 3-D border to draw around the widget (using TK_CONFIG_BORDER). In cases like this it is possible to specify that several consecutive entries in *specs* are to be treated as a group. The first entry is used to determine a value (using its *argvName*, *dbName*, *dbClass*, and *defValue* fields). The value will be processed several times (one for each entry in the group), generating multiple different resources and modifying multiple targets within *widgRec*. Each of the entries after the first must have a NULL value in its *argvName* field; this indicates that the entry is to be grouped with the entry that precedes it. Only the *type* and *offset* fields are used from these follow-on entries.

FLAGS

The *flags* argument passed to **Tk_ConfigureWidget** is used in conjunction with the *specFlags* fields in the entries of *specs* to provide additional control over the processing of configuration options. These values are used in three different ways as described below.

First, if the *flags* argument to **Tk_ConfigureWidget** has the TK_CONFIG_ARGV_ONLY bit set (i.e., *flags* | TK_CONFIG_ARGV_ONLY != 0), then the option database and *defValue* fields are not used. In this case, if an entry in *specs* doesn’t match a field in *argv* then nothing happens: the corresponding target isn’t modified. This feature is useful when the goal is to modify certain configuration options while leaving others in their current state, such as when a **configure** widget command is being processed.

Second, the *specFlags* field of an entry in *specs* may be used to control the processing of that entry. Each *specFlags* field may consists of an OR-ed combination of the following values:

TK_CONFIG_COLOR_ONLY
If this bit is set then the entry will only be considered if the display for *tkwin* has more than one bit plane. If the display is monochromatic then this *specs* entry will be ignored.

TK_CONFIG_MONO_ONLY
If this bit is set then the entry will only be considered if the display for *tkwin* has exactly one bit plane. If the display is not monochromatic then this *specs* entry will be ignored.

TK_CONFIG_NULL_OK
This bit is only relevant for some types of entries (see the descriptions of the various entry types above). If this bit is set, it indicates that an empty string value for the field is acceptable and if it occurs then the target should be set to NULL or **None**, depending on the type of the target. This flag is typically used to allow a feature to be turned off entirely, e.g. set a cursor value to **None** so that a window simply inherits its parent’s cursor. If this bit isn’t set then empty strings are processed as strings, which generally results in an error.

TK_CONFIG_DONT_SET_DEFAULT

If this bit is one, it means that the *defValue* field of the entry should only be used for returning the default value in **Tk_ConfigureInfo**. In calls to **Tk_ConfigureWidget** no default will be supplied for entries with this flag set; it is assumed that the caller has already supplied a default value in the target location. This flag provides a performance optimization where it is expensive to process the default string: the client can compute the default once, save the value, and provide it before calling **Tk_ConfigureWidget**.

TK_CONFIG_OPTION_SPECIFIED

This bit is set and cleared by **Tk_ConfigureWidget**. Whenever **Tk_ConfigureWidget** returns, this bit will be set in all the entries where a value was specified in *argv*. It will be zero in all other entries. This bit provides a way for clients to determine which values actually changed in a call to **Tk_ConfigureWidget**.

The TK_CONFIG_MONO_ONLY and TK_CONFIG_COLOR_ONLY flags are typically used to specify different default values for monochrome and color displays. This is done by creating two entries in *specs* that are identical except for their *defValue* and *specFlags* fields. One entry should have the value TK_CONFIG_MONO_ONLY in its *specFlags* and the default value for monochrome displays in its *defValue*; the other entry entry should have the value TK_CONFIG_COLOR_ONLY in its *specFlags* and the appropriate *defValue* for color displays.

Third, it is possible to use *flags* and *specFlags* together to selectively disable some entries. This feature is not needed very often. It is useful in cases where several similar kinds of widgets are implemented in one place. It allows a single *specs* table to be created with all the configuration options for all the widget types. When processing a particular widget type, only entries relevant to that type will be used. This effect is achieved by setting the high-order bits (those in positions equal to or greater than TK_CONFIG_USER_BIT) in *specFlags* values or in *flags*. In order for a particular entry in *specs* to be used, its high-order bits must match exactly the high-order bits of the *flags* value passed to **Tk_ConfigureWidget**. If a *specs* table is being used for N different widget types, then N of the high-order bits will be used. Each *specs* entry will have one of more of those bits set in its *specFlags* field to indicate the widget types for which this entry is valid. When calling **Tk_ConfigureWidget**, *flags* will have a single one of these bits set to select the entries for the desired widget type. For a working example of this feature, see the code in tkButton.c.

TK_OFFSET

The **Tk_Offset** macro is provided as a safe way of generating the *offset* values for entries in Tk_ConfigSpec structures. It takes two arguments: the name of a type of record, and the name of a field in that record. It returns the byte offset of the named field in records of the given type.

TK_CONFIGUREINFO

The **Tk_ConfigureInfo** procedure may be used to obtain information about one or all of the options for a given widget. Given a token for a window (*tkwin*), a table describing the configuration options for a class of widgets (*specs*), a pointer to a widget record containing the current information for a widget (*widgRec*), and a NULL *argvName* argument, **Tk_ConfigureInfo** generates a string describing all of the configuration options for the window. The string is placed in *interp->result*. Under normal circumstances it returns TCL_OK; if an error occurs then it returns TCL_ERROR and *interp->result* contains an error message.

If *argvName* is NULL, then the value left in *interp->result* by **Tk_ConfigureInfo** consists of a list of one or more entries, each of which describes one configuration option (i.e. one entry in *specs*). Each entry in the list will contain either two or five values. If the corresponding entry in *specs* has type TK_CONFIG_SYNONYM, then the list will contain two values: the *argvName* for the entry and the *dbName* (synonym name). Otherwise the list will contain five values: *argvName*, *dbName*, *dbClass*, *defValue*, and current value. The current value is computed from the appropriate field of *widgRec* by calling procedures like **Tk_NameOfColor**.

If the *argvName* argument to **Tk_ConfigureInfo** is non-NULL, then it indicates a single option, and information is returned only for that option. The string placed in *interp->result* will be a list containing two or five values as described above; this will be identical to the corresponding sublist that would have been returned if *argvName* had been NULL.

The *flags* argument to **Tk_ConfigureInfo** is used to restrict the *specs* entries to consider, just as for **Tk_ConfigureWidget**.

TK_CONFIGUREVALUE

Tk_ConfigureValue takes arguments similar to **Tk_ConfigureInfo**; instead of returning a list of values, it just returns the current value of the option given by *argvName* (*argvName* must not be NULL). The value is returned in *interp->result* and TCL_OK is normally returned as the procedure's result. If an error occurs in **Tk_ConfigureValue** (e.g., *argvName* is not a valid option name), TCL_ERROR is returned and an error message is left in *interp->result*. This procedure is typically called to implement **cget** widget commands.

TK_FREEOPTIONS

The **Tk_FreeOptions** procedure may be invoked during widget cleanup to release all of the resources associated with configuration options. It scans through *specs* and for each entry corresponding to a resource that must be explicitly freed (e.g. those with type TK_CONFIG_COLOR), it frees the resource in the widget record. If the field in the widget record doesn't refer to a resource (e.g. it contains a null pointer) then no resource is freed for that entry. After freeing a resource, **Tk_FreeOptions** sets the corresponding field of the widget record to null.

CUSTOM OPTION TYPES

Applications can extend the built-in configuration types with additional configuration types by writing procedures to parse and print options of the a type and creating a structure pointing to those procedures:

```
typedef struct Tk_CustomOption {
    Tk_OptionParseProc *parseProc;
    Tk_OptionPrintProc *printProc;
    ClientData clientData;
} Tk_CustomOption;

typedef int Tk_OptionParseProc(
    ClientData clientData,
    Tcl_Interp *interp,
    Tk_Window tkwin,
    char *value,
    char *widgRec,
    int offset);

typedef char *Tk_OptionPrintProc(
    ClientData clientData,
    Tk_Window tkwin,
    char *widgRec,
    int offset,
    Tcl_FreeProc **freeProcPtr);
```

The Tk_CustomOption structure contains three fields, which are pointers to the two procedures and a *clientData* value to be passed to those procedures when they are invoked. The *clientData* value typically points to a structure containing information that is needed by the procedures when they are parsing and printing options.

The *parseProc* procedure is invoked by **Tk_ConfigureWidget** to parse a string and store the resulting value in the widget record. The *clientData* argument is a copy of the *clientData* field in the Tk_CustomOption structure. The *interp* argument points to a Tcl interpreter used for error reporting. *Tkwin* is a copy of the *tkwin* argument to **Tk_ConfigureWidget**. The *value* argument is a string describing the value for the option; it could have been specified explicitly in the call to **Tk_ConfigureWidget** or it could come from the option database or a default. *Value* will never be a null pointer but it may point to an empty string. *RecordPtr* is the same as the *widgRec* argument to **Tk_ConfigureWidget**; it points to the start of the widget record to modify. The last argument, *offset*, gives the offset in bytes from the start of the widget record to the location where the option value is to be placed. The procedure should translate the string to whatever form is appropriate for the option and store the value in the widget record. It should normally return TCL_OK, but if an error occurs in translating the string to a value then it should return TCL_ERROR and store an error message in *interp->result*.

The *printProc* procedure is called by **Tk_ConfigureInfo** to produce a string value describing an existing option. Its *clientData*, *tkwin*, *widgRec*, and *offset* arguments all have the same meaning as for Tk_OptionParseProc procedures. The *printProc* procedure should examine the option whose value is stored at *offset* in *widgRec*, produce a string describing that option, and return a pointer to the string. If the string is stored in dynamically-allocated memory, then the procedure must set **freeProcPtr* to the address of a procedure to call to free the string's memory; **Tk_ConfigureInfo** will call this procedure when it is finished with the string. If the result string is stored in static memory then *printProc* need not do anything with the *freeProcPtr* argument.

Once *parseProc* and *printProc* have been defined and a Tk_CustomOption structure has been created for them, options of this new type may be manipulated with Tk_ConfigSpec entries whose *type* fields are TK_CONFIG_CUSTOM and whose *customPtr* fields point to the Tk_CustomOption structure.

EXAMPLES

Although the explanation of **Tk_ConfigureWidget** is fairly complicated, its actual use is pretty straightforward. The easiest way to get started is to copy the code from an existing widget. The library implementation of frames (tkFrame.c) has a simple configuration table, and the library implementation of buttons (tkButton.c) has a much more complex table that uses many of the fancy *specFlags* mechanisms.

KEYWORDS

anchor, bitmap, boolean, border, cap style, color, configuration options, cursor, custom, double, font, integer, join style, justify, millimeters, pixels, relief, synonym, uid

NAME

Tk_ConfigureWindow, Tk_MoveWindow, Tk_ResizeWindow, Tk_MoveResizeWindow, Tk_SetWindowBorderWidth, Tk_ChangeWindowAttributes, Tk_SetWindowBackground, Tk_SetWindowBackgroundPixmap, Tk_SetWindowBorder, Tk_SetWindowBorderPixmap, Tk_SetWindowColormap, Tk_DefineCursor, Tk_UndefineCursor − change window configuration or attributes

SYNOPSIS

#include <tk.h>

Tk_ConfigureWindow(*tkwin, valueMask, valuePtr*)

Tk_MoveWindow(*tkwin, x, y*)

Tk_ResizeWindow(*tkwin, width, height*)

Tk_MoveResizeWindow(*tkwin, x, y, width, height*)

Tk_SetWindowBorderWidth(*tkwin, borderWidth*)

Tk_ChangeWindowAttributes(*tkwin, valueMask, attsPtr*)

Tk_SetWindowBackground(*tkwin, pixel*)

Tk_SetWindowBackgroundPixmap(*tkwin, pixmap*)

Tk_SetWindowBorder(*tkwin, pixel*)

Tk_SetWindowBorderPixmap(*tkwin, pixmap*)

Tk_SetWindowColormap(*tkwin, colormap*)

Tk_DefineCursor(*tkwin, cursor*)

Tk_UndefineCursor(*tkwin*)

ARGUMENTS

Tk_Window	*tkwin*	(in)	Token for window.
unsigned int	*valueMask*	(in)	OR-ed mask of values like **CWX** or **CWBorderPixel**, indicating which fields of ∗*valuePtr* or ∗*attsPtr* to use.
XWindowChanges	∗*valuePtr*	(in)	Points to a structure containing new values for the configuration parameters selected by *valueMask*. Fields not selected by *valueMask* are ignored.
int	*x*	(in)	New x-coordinate for *tkwin*'s top left pixel (including border, if any) within tkwin's parent.
int	*y*	(in)	New y-coordinate for *tkwin*'s top left pixel (including border, if any) within tkwin's parent.
int	*width*	(in)	New width for *tkwin* (interior, not including border).
int	*height*	(in)	New height for *tkwin* (interior, not including border).
int	*borderWidth*	(in)	New width for *tkwin*'s border.

XSetWindowAttributes	**attsPtr*	(in)	Points to a structure containing new values for the attributes given by the *valueMask* argument. Attributes not selected by *valueMask* are ignored.
unsigned long	*pixel*	(in)	New background or border color for window.
Pixmap	*pixmap*	(in)	New pixmap to use for background or border of *tkwin*. WARNING: cannot necessarily be deleted immediately, as for Xlib calls. See note below.
Colormap	*colormap*	(in)	New colormap to use for *tkwin*.
Tk_Cursor	*cursor*	(in)	New cursor to use for *tkwin*. If **None** is specified, then *tkwin* will not have its own cursor; it will use the cursor of its parent.

DESCRIPTION

These procedures are analogous to the X library procedures with similar names, such as **XConfigureWindow**. Each one of the above procedures calls the corresponding X procedure and also saves the configuration information in Tk's local structure for the window. This allows the information to be retrieved quickly by the application (using macros such as **Tk_X** and **Tk_Height**) without having to contact the X server. In addition, if no X window has actually been created for *tkwin* yet, these procedures do not issue X operations or cause event handlers to be invoked; they save the information in Tk's local structure for the window; when the window is created later, the saved information will be used to configure the window.

See the X library documentation for details on what these procedures do and how they use their arguments.

In the procedures **Tk_ConfigureWindow**, **Tk_MoveWindow**, **Tk_ResizeWindow**, **Tk_MoveResizeWindow**, and **Tk_SetWindowBorderWidth**, if *tkwin* is an internal window then event handlers interested in configure events are invoked immediately, before the procedure returns. If *tkwin* is a top-level window then the event handlers will be invoked later, after X has seen the request and returned an event for it.

Applications using Tk should never call procedures like **XConfigureWindow** directly; they should always use the corresponding Tk procedures.

The size and location of a window should only be modified by the appropriate geometry manager for that window and never by a window itself (but see **Tk_MoveToplevelWindow** for moving a top-level window).

You may not use **Tk_ConfigureWindow** to change the stacking order of a window (*valueMask* may not contain the **CWSibling** or **CWStackMode** bits). To change the stacking order, use the procedure **Tk_RestackWindow**.

The procedure **Tk_SetWindowColormap** will automatically add *tkwin* to the **TK_COLORMAP_WINDOWS** property of its nearest top-level ancestor if the new colormap is different from that of *tkwin*'s parent and *tkwin* isn't already in the **TK_COLORMAP_WINDOWS** property.

BUGS

Tk_SetWindowBackgroundPixmap and **Tk_SetWindowBorderPixmap** differ slightly from their Xlib counterparts in that the *pixmap* argument may not necessarily be deleted immediately after calling one of these procedures. This is because *tkwin*'s window may not exist yet at the time of the call, in which case *pixmap* is merely saved and used later when *tkwin*'s window is actually created. If you wish to delete *pixmap*, then call **Tk_MakeWindowExist** first to be sure that *tkwin*'s window exists and *pixmap* has been passed to the X server.

A similar problem occurs for the *cursor* argument passed to **Tk_DefineCursor**. The solution is the same as for pixmaps above: call **Tk_MakeWindowExist** before freeing the cursor.

SEE ALSO

Tk_MoveToplevelWindow, Tk_RestackWindow

KEYWORDS

attributes, border, color, configure, height, pixel, pixmap, width, window, x, y

NAME

Tk_CoordsToWindow − Find window containing a point

SYNOPSIS

#include <tk.h>

Tk_Window
Tk_CoordsToWindow(*rootX, rootY, tkwin*)

ARGUMENTS

int	*rootX*	(in)	X-coordinate (in root window coordinates).
int	*rootY*	(in)	Y-coordinate (in root window coordinates).
Tk_Window	*tkwin*	(in)	Token for window that identifies application.

DESCRIPTION

Tk_CoordsToWindow locates the window that contains a given point. The point is specified in root coordinates with *rootX* and *rootY* (if a virtual-root window manager is in use then *rootX* and *rootY* are in the coordinate system of the virtual root window). The return value from the procedure is a token for the window that contains the given point. If the point is not in any window, or if the containing window is not in the same application as *tkwin*, then NULL is returned.

The containing window is decided using the same rules that determine which window contains the mouse cursor: if a parent and a child both contain the point then the child gets preference, and if two siblings both contain the point then the highest one in the stacking order (i.e. the one that's visible on the screen) gets preference.

KEYWORDS

containing, coordinates, root window

NAME

Tk_CreateErrorHandler, Tk_DeleteErrorHandler − handle X protocol errors

SYNOPSIS

#include <tk.h>

Tk_ErrorHandler
Tk_CreateErrorHandler(*display, error, request, minor, proc, clientData*)

Tk_DeleteErrorHandler(*handler*)

ARGUMENTS

Display	∗*display*	(in)	Display whose errors are to be handled.
int	*error*	(in)	Match only error events with this value in the *error_code* field. If -1, then match any *error_code* value.
int	*request*	(in)	Match only error events with this value in the *request_code* field. If -1, then match any *request_code* value.
int	*minor*	(in)	Match only error events with this value in the *minor_code* field. If -1, then match any *minor_code* value.
Tk_ErrorProc	∗*proc*	(in)	Procedure to invoke whenever an error event is received for *display* and matches *error*, *request*, and *minor*. NULL means ignore any matching errors.
ClientData	*clientData*	(in)	Arbitrary one-word value to pass to *proc*.
Tk_ErrorHandler	*handler*	(in)	Token for error handler to delete (return value from a previous call to **Tk_CreateErrorHandler**).

DESCRIPTION

Tk_CreateErrorHandler arranges for a particular procedure (*proc*) to be called whenever certain protocol errors occur on a particular display (*display*). Protocol errors occur when the X protocol is used incorrectly, such as attempting to map a window that doesn't exist. See the Xlib documentation for **XSetErrorHandler** for more information on the kinds of errors that can occur. For *proc* to be invoked to handle a particular error, five things must occur:

[1] The error must pertain to *display*.

[2] Either the *error* argument to **Tk_CreateErrorHandler** must have been -1, or the *error* argument must match the *error_code* field from the error event.

[3] Either the *request* argument to **Tk_CreateErrorHandler** must have been -1, or the *request* argument must match the *request_code* field from the error event.

[4] Either the *minor* argument to **Tk_CreateErrorHandler** must have been -1, or the *minor* argument must match the *minor_code* field from the error event.

[5] The protocol request to which the error pertains must have been made when the handler was active (see below for more information).

Proc should have arguments and result that match the following type:

```
typedef int Tk_ErrorProc(
    ClientData clientData,
    XErrorEvent *errEventPtr);
```

The *clientData* parameter to *proc* is a copy of the *clientData* argument given to **Tcl_CreateErrorHandler** when the callback was created. Typically, *clientData* points to a data structure containing application-

specific information that is needed to deal with the error. *ErrEventPtr* is a pointer to the X error event. The procedure *proc* should return an integer value. If it returns 0 it means that *proc* handled the error completely and there is no need to take any other action for the error. If it returns non-zero it means *proc* was unable to handle the error.

If a value of NULL is specified for *proc*, all matching errors will be ignored: this will produce the same result as if a procedure had been specified that always returns 0.

If more than more than one handler matches a particular error, then they are invoked in turn. The handlers will be invoked in reverse order of creation: most recently declared handler first. If any handler returns 0, then subsequent (older) handlers will not be invoked. If no handler returns 0, then Tk invokes X'es default error handler, which prints an error message and aborts the program. If you wish to have a default handler that deals with errors that no other handler can deal with, then declare it first.

The X documentation states that ''the error handler should not call any functions (directly or indirectly) on the display that will generate protocol requests or that will look for input events.'' This restriction applies to handlers declared by **Tk_CreateErrorHandler**; disobey it at your own risk.

Tk_DeleteErrorHandler may be called to delete a previously-created error handler. The *handler* argument identifies the error handler, and should be a value returned by a previous call to **Tk_CreateEventHandler**.

A particular error handler applies to errors resulting from protocol requests generated between the call to **Tk_CreateErrorHandler** and the call to **Tk_DeleteErrorHandler**. However, the actual callback to *proc* may not occur until after the **Tk_DeleteErrorHandler** call, due to buffering in the client and server. If an error event pertains to a protocol request made just before calling **Tk_DeleteErrorHandler**, then the error event may not have been processed before the **Tk_DeleteErrorHandler** call. When this situation arises, Tk will save information about the handler and invoke the handler's *proc* later when the error event finally arrives. If an application wishes to delete an error handler and know for certain that all relevant errors have been processed, it should first call **Tk_DeleteErrorHandler** and then call **XSync**; this will flush out any buffered requests and errors, but will result in a performance penalty because it requires communication to and from the X server. After the **XSync** call Tk is guaranteed not to call any error handlers deleted before the **XSync** call.

For the Tk error handling mechanism to work properly, it is essential that application code never calls **XSetErrorHandler** directly; applications should use only **Tk_CreateErrorHandler**.

KEYWORDS

callback, error, event, handler

NAME

Tk_CreateGenericHandler, Tk_DeleteGenericHandler − associate procedure callback with all X events

SYNOPSIS

#include <tk.h>

Tk_CreateGenericHandler(*proc, clientData*)

Tk_DeleteGenericHandler(*proc, clientData*)

ARGUMENTS

Tk_GenericProc	∗*proc*	(in)	Procedure to invoke whenever any X event occurs on any display.
ClientData	*clientData*	(in)	Arbitrary one-word value to pass to *proc*.

DESCRIPTION

Tk_CreateGenericHandler arranges for *proc* to be invoked in the future whenever any X event occurs. This mechanism is *not* intended for dispatching X events on windows managed by Tk (you should use **Tk_CreateEventHandler** for this purpose). **Tk_CreateGenericHandler** is intended for other purposes, such as tracing X events, monitoring events on windows not owned by Tk, accessing X-related libraries that were not originally designed for use with Tk, and so on.

The callback to *proc* will be made by **Tk_HandleEvent**; this mechanism only works in programs that dispatch events through **Tk_HandleEvent** (or through other Tk procedures that call **Tk_HandleEvent**, such as **Tk_DoOneEvent** or **Tk_MainLoop**).

Proc should have arguments and result that match the type **Tk_GenericProc**:

```
typedef int Tk_GenericProc(
    ClientData clientData,
    XEvent *eventPtr);
```

The *clientData* parameter to *proc* is a copy of the *clientData* argument given to **Tk_CreateGenericHandler** when the callback was created. Typically, *clientData* points to a data structure containing application-specific information about how to handle events. *EventPtr* is a pointer to the X event.

Whenever an X event is processed by **Tk_HandleEvent**, *proc* is called. The return value from *proc* is normally 0. A non-zero return value indicates that the event is not to be handled further; that is, *proc* has done all processing that is to be allowed for the event.

If there are multiple generic event handlers, each one is called for each event, in the order in which they were established.

Tk_DeleteGenericHandler may be called to delete a previously-created generic event handler: it deletes each handler it finds that matches the *proc* and *clientData* arguments. If no such handler exists, then **Tk_DeleteGenericHandler** returns without doing anything. Although Tk supports it, it's probably a bad idea to have more than one callback with the same *proc* and *clientData* arguments.

Establishing a generic event handler does nothing to ensure that the process will actually receive the X events that the handler wants to process. For example, it is the caller's responsibility to invoke **XSelectInput** to select the desired events, if that is necessary.

KEYWORDS

bind, callback, event, handler

NAME

Tk_CreateImageType, Tk_GetImageMasterData − define new kind of image

SYNOPSIS

#include <tk.h>

Tk_CreateImageType(*typePtr*)
ClientData

Tk_GetImageMasterData(*interp, name, typePtrPtr*)

ARGUMENTS

Tk_ImageType	∗*typePtr*	(in)	Structure that defines the new type of image. Must be static: a pointer to this structure is retained by the image code.
Tcl_Interp	∗*interp*	(in)	Interpreter in which image was created.
char	∗*name*	(in)	Name of existing image.
Tk_ImageType	∗∗*typePtrPtr*	(out)	Points to word in which to store a pointer to type information for the given image, if it exists.

DESCRIPTION

Tk_CreateImageType is invoked to define a new kind of image. An image type corresponds to a particular value of the *type* argument for the **image create** command. There may exist any number of different image types, and new types may be defined dynamically by calling **Tk_CreateImageType**. For example, there might be one type for 2-color bitmaps, another for multi-color images, another for dithered images, another for video, and so on.

The code that implements a new image type is called an *image manager*. It consists of a collection of procedures plus three different kinds of data structures. The first data structure is a Tk_ImageType structure, which contains the name of the image type and pointers to five procedures provided by the image manager to deal with images of this type:

```
typedef struct Tk_ImageType {
    char ∗name;
    Tk_ImageCreateProc ∗createProc;
    Tk_ImageGetProc ∗getProc;
    Tk_ImageDisplayProc ∗displayProc;
    Tk_ImageFreeProc ∗freeProc;
    Tk_ImageDeleteProc ∗deleteProc;
} Tk_ImageType;
```

The fields of this structure will be described in later subsections of this entry.

The second major data structure manipulated by an image manager is called an *image master*; it contains overall information about a particular image, such as the values of the configuration options specified in an **image create** command. There will usually be one of these structures for each invocation of the **image create** command.

The third data structure related to images is an *image instance*. There will usually be one of these structures for each usage of an image in a particular widget. It is possible for a single image to appear simultaneously in multiple widgets, or even multiple times in the same widget. Furthermore, different instances may be on different screens or displays. The image instance data structure describes things that may vary from instance to instance, such as colors and graphics contexts for redisplay. There is usually one instance structure for each **−image** option specified for a widget or canvas item.

The following subsections describe the fields of a Tk_ImageType in more detail.

NAME

typePtr->name provides a name for the image type. Once **Tk_CreateImageType** returns, this name may be used in **image create** commands to create images of the new type. If there already existed an image type by this name then the new image type replaces the old one.

CREATEPROC

typePtr->createProc provides the address of a procedure for Tk to call whenever **image create** is invoked to create an image of the new type. *typePtr->createProc* must match the following prototype:

```
typedef int Tk_ImageCreateProc(
    Tcl_Interp *interp,
    char *name,
    int argc,
    char **argv,
    Tk_ImageType *typePtr,
    Tk_ImageMaster master,
    ClientData *masterDataPtr);
```

The *interp* argument is the interpreter in which the **image** command was invoked, and *name* is the name for the new image, which was either specified explicitly in the **image** command or generated automatically by the **image** command. The *argc* and *argv* arguments describe all the configuration options for the new image (everything after the name argument to **image**). The *master* argument is a token that refers to Tk's information about this image; the image manager must return this token to Tk when invoking the **Tk_ImageChanged** procedure. Typically *createProc* will parse *argc* and *argv* and create an image master data structure for the new image. *createProc* may store an arbitrary one-word value at *masterDataPtr*, which will be passed back to the image manager when other callbacks are invoked. Typically the value is a pointer to the master data structure for the image.

If *createProc* encounters an error, it should leave an error message in *interp->result* and return **TCL_ERROR**; otherwise it should return **TCL_OK**.

createProc should call **Tk_ImageChanged** in order to set the size of the image and request an initial redisplay.

GETPROC

typePtr->getProc is invoked by Tk whenever a widget calls **Tk_GetImage** to use a particular image. This procedure must match the following prototype:

```
typedef ClientData Tk_ImageGetProc(
    Tk_Window tkwin,
    ClientData masterData);
```

The *tkwin* argument identifies the window in which the image will be used and *masterData* is the value returned by *createProc* when the image master was created. *getProc* will usually create a data structure for the new instance, including such things as the resources needed to display the image in the given window. *getProc* returns a one-word token for the instance, which is typically the address of the instance data structure. Tk will pass this value back to the image manager when invoking its *displayProc* and *freeProc* procedures.

DISPLAYPROC

typePtr->displayProc is invoked by Tk whenever an image needs to be displayed (i.e., whenever a widget calls **Tk_RedrawImage**). *displayProc* must match the following prototype:

```
typedef void Tk_ImageDisplayProc(
```

ClientData *instanceData*,
Display **display*,
Drawable *drawable*,
int *imageX*,
int *imageY*,
int *width*,
int *height*,
int *drawableX*,
int *drawableY*);

The *instanceData* will be the same as the value returned by *getProc* when the instance was created. *display* and *drawable* indicate where to display the image; *drawable* may be a pixmap rather than the window specified to *getProc* (this is usually the case, since most widgets double-buffer their redisplay to get smoother visual effects). *imageX*, *imageY*, *width*, and *height* identify the region of the image that must be redisplayed. This region will always be within the size of the image as specified in the most recent call to **Tk_ImageChanged**. *drawableX* and *drawableY* indicate where in *drawable* the image should be displayed; *displayProc* should display the given region of the image so that point (*imageX*, *imageY*) in the image appears at (*drawableX*, *drawableY*) in *drawable*.

FREEPROC

typePtr->freeProc contains the address of a procedure that Tk will invoke when an image instance is released (i.e., when **Tk_FreeImage** is invoked). This can happen, for example, when a widget is deleted or a image item in a canvas is deleted, or when the image displayed in a widget or canvas item is changed. *freeProc* must match the following prototype:

typedef void Tk_ImageFreeProc(
ClientData *instanceData*,
Display **display*);

The *instanceData* will be the same as the value returned by *getProc* when the instance was created, and *display* is the display containing the window for the instance. *freeProc* should release any resources associated with the image instance, since the instance will never be used again.

DELETEPROC

typePtr->deleteProc is a procedure that Tk invokes when an image is being deleted (i.e. when the **image delete** command is invoked). Before invoking *deleteProc* Tk will invoke *freeProc* for each of the image's instances. *deleteProc* must match the following prototype:

typedef void Tk_ImageDeleteProc(
ClientData *masterData*);

The *masterData* argument will be the same as the value stored in **masterDataPtr* by *createProc* when the image was created. *deleteProc* should release any resources associated with the image.

TK_GETIMAGEMASTERDATA

The procedure **Tk_GetImageMasterData** may be invoked to retrieve information about an image. For example, an image manager can use this procedure to locate its image master data for an image. If there exists an image named *name* in the interpreter given by *interp*, then **typePtrPtr* is filled in with type information for the image (the *typePtr* value passed to **Tk_CreateImageType** when the image type was registered) and the return value is the ClientData value returned by the *createProc* when the image was created (this is typically a pointer to the image master data structure). If no such image exists then NULL is returned and NULL is stored at **typePtrPtr*.

SEE ALSO

Tk_ImageChanged, Tk_GetImage, Tk_FreeImage, Tk_RedrawImage, Tk_SizeOfImage

KEYWORDS

image manager, image type, instance, master

NAME

Tk_CreateItemType, Tk_GetItemTypes − define new kind of canvas item

SYNOPSIS

#include <tk.h>

Tk_CreateItemType(*typePtr*)

Tk_ItemType ∗
Tk_GetItemTypes()

ARGUMENTS

Tk_ItemType	∗*typePtr*	(in)	Structure that defines the new type of canvas item.

INTRODUCTION

Tk_CreateItemType is invoked to define a new kind of canvas item described by the *typePtr* argument. An item type corresponds to a particular value of the *type* argument to the **create** widget command for canvases, and the code that implements a canvas item type is called a *type manager*. Tk defines several built-in item types, such as **rectangle** and **text** and **image**, but **Tk_CreateItemType** allows additional item types to be defined. Once **Tk_CreateItemType** returns, the new item type may be used in new or existing canvas widgets just like the built-in item types.

Tk_GetItemTypes returns a pointer to the first in the list of all item types currently defined for canvases. The entries in the list are linked together through their *nextPtr* fields, with the end of the list marked by a NULL *nextPtr*.

You may find it easier to understand the rest of this manual entry by looking at the code for an existing canvas item type such as bitmap (file tkCanvBmap.c) or text (tkCanvText.c). The easiest way to create a new type manager is to copy the code for an existing type and modify it for the new type.

Tk provides a number of utility procedures for the use of canvas type managers, such as **Tk_CanvasCoords** and **Tk_CanvasPsColor**; these are described in separate manual entries.

DATA STRUCTURES

A type manager consists of a collection of procedures that provide a standard set of operations on items of that type. The type manager deals with three kinds of data structures. The first data structure is a Tk_ItemType; it contains information such as the name of the type and pointers to the standard procedures implemented by the type manager:

```
typedef struct Tk_ItemType {
    char *name;
    int itemSize;
    Tk_ItemCreateProc *createProc;
    Tk_ConfigSpec *configSpecs;
    Tk_ItemConfigureProc *configProc;
    Tk_ItemCoordProc *coordProc;
    Tk_ItemDeleteProc *deleteProc;
    Tk_ItemDisplayProc *displayProc;
    int alwaysRedraw;
    Tk_ItemPointProc *pointProc;
    Tk_ItemAreaProc *areaProc;
    Tk_ItemPostscriptProc *postscriptProc;
    Tk_ItemScaleProc *scaleProc;
    Tk_ItemTranslateProc *translateProc;
```

```
    Tk_ItemIndexProc *indexProc;
    Tk_ItemCursorProc *icursorProc;
    Tk_ItemSelectionProc *selectionProc;
    Tk_ItemInsertProc *insertProc;
    Tk_ItemDCharsProc *dCharsProc;
    Tk_ItemType *nextPtr;
} Tk_ItemType;
```

The fields of a Tk_ItemType structure are described in more detail later in this manual entry. When **Tk_CreateItemType** is called, its *typePtr* argument must point to a structure with all of the fields initialized except *nextPtr*, which Tk sets to link all the types together into a list. The structure must be in permanent memory (either statically allocated or dynamically allocated but never freed); Tk retains a pointer to this structure.

The second data structure manipulated by a type manager is an *item record*. For each item in a canvas there exists one item record. All of the items of a given type generally have item records with the same structure, but different types usually have different formats for their item records. The first part of each item record is a header with a standard structure defined by Tk via the type Tk_Item; the rest of the item record is defined by the type manager. A type manager must define its item records with a Tk_Item as the first field. For example, the item record for bitmap items is defined as follows:

```
typedef struct BitmapItem {
    Tk_Item header;
    double x, y;
    Tk_Anchor anchor;
    Pixmap bitmap;
    XColor *fgColor;
    XColor *bgColor;
    GC gc;
} BitmapItem;
```

The *header* substructure contains information used by Tk to manage the item, such as its identifier, its tags, its type, and its bounding box. The fields starting with *x* belong to the type manager: Tk will never read or write them. The type manager should not need to read or write any of the fields in the header except for four fields whose names are *x1*, *y1*, *x2*, and *y2*. These fields give a bounding box for the items using integer canvas coordinates: the item should not cover any pixels with x-coordinate lower than *x1* or y-coordinate lower than *y1*, nor should it cover any pixels with x-coordinate greater than or equal to *x2* or y-coordinate greater than or equal to *y2*. It is up to the type manager to keep the bounding box up to date as the item is moved and reconfigured.

Whenever Tk calls a procedure in a type manager it passes in a pointer to an item record. The argument is always passed as a pointer to a Tk_Item; the type manager will typically cast this into a pointer to its own specific type, such as BitmapItem.

The third data structure used by type managers has type Tk_Canvas; it serves as an opaque handle for the canvas widget as a whole. Type managers need not know anything about the contents of this structure. A Tk_Canvas handle is typically passed in to the procedures of a type manager, and the type manager can pass the handle back to library procedures such as Tk_CanvasTkwin to fetch information about the canvas.

NAME

This section and the ones that follow describe each of the fields in a Tk_ItemType structure in detail. The *name* field provides a string name for the item type. Once **Tk_CreateImageType** returns, this name may be used in **create** widget commands to create items of the new type. If there already existed an item type by this name then the new item type replaces the old one.

ITEMSIZE

typePtr->itemSize gives the size in bytes of item records of this type, including the Tk_Item header. Tk uses this size to allocate memory space for items of the type. All of the item records for a given type must have the same size. If variable length fields are needed for an item (such as a list of points for a polygon), the type manager can allocate a separate object of variable length and keep a pointer to it in the item record.

CREATEPROC

typePtr->createProc points to a procedure for Tk to call whenever a new item of this type is created. *typePtr->createProc* must match the following prototype:

```
typedef int Tk_ItemCreateProc(
    Tcl_Interp *interp,
    Tk_Canvas canvas,
    Tk_Item *itemPtr,
    int argc,
    char **argv);
```

The *interp* argument is the interpreter in which the canvas's **create** widget command was invoked, and *canvas* is a handle for the canvas widget. *itemPtr* is a pointer to a newly-allocated item of size *typePtr->itemSize*. Tk has already initialized the item's header (the first **sizeof(Tk_ItemType)** bytes). The *argc* and *argv* arguments describe all of the arguments to the **create** command after the *type* argument. For example, in the widget command

.c create rectangle 10 20 50 50 −fill black

argc will be **6** and *argv*[0] will contain the string **10**.

createProc should use *argc* and *argv* to initialize the type-specific parts of the item record and set an initial value for the bounding box in the item's header. It should return a standard Tcl completion code and leave an error message in *interp->result* if an error occurs. If an error occurs Tk will free the item record, so *createProc* must be sure to leave the item record in a clean state if it returns an error (e.g., it must free any additional memory that it allocated for the item).

CONFIGSPECS

Each type manager must provide a standard table describing its configuration options, in a form suitable for use with **Tk_ConfigureWidget**. This table will normally be used by *typePtr->createProc* and *typePtr->configProc*, but Tk also uses it directly to retrieve option information in the **itemcget** and **itemconfigure** widget commands. *typePtr->configSpecs* must point to the configuration table for this type. Note: Tk provides a custom option type **tk_CanvasTagsOption** for implementing the **−tags** option; see an existing type manager for an example of how to use it in *configSpecs*.

CONFIGPROC

typePtr->configProc is called by Tk whenever the **itemconfigure** widget command is invoked to change the configuration options for a canvas item. This procedure must match the following prototype:

```
typedef int Tk_ItemConfigureProc(
    Tcl_Interp *interp,
    Tk_Canvas canvas,
    Tk_Item *itemPtr,
    int argc,
    char **argv,
    int flags);
```

The *interp* argument identifies the interpreter in which the widget command was invoked, *canvas* is a handle for the canvas widget, and *itemPtr* is a pointer to the item being configured. *argc* and *argv* contain the configuration options. For example, if the following command is invoked:

.c itemconfigure 2 −fill red −outline black

argc is **4** and *argv* contains the strings **−fill** through **black**. *argc* will always be an even value. The *flags* argument contains flags to pass to **Tk_ConfigureWidget**; currently this value is always TK_CONFIG_ARGV_ONLY when Tk invokes *typePtr->configProc*, but the type manager's *createProc* procedure will usually invoke *configProc* with different flag values.

typePtr->configProc returns a standard Tcl completion code and leaves an error message in *interp->result* if an error occurs. It must update the item's bounding box to reflect the new configuration options.

COORDPROC

typePtr->coordProc is invoked by Tk to implement the **coords** widget command for an item. It must match the following prototype:

```
typedef int Tk_ItemCoordProc(
    Tcl_Interp *interp,
    Tk_Canvas canvas,
    Tk_Item *itemPtr,
    int argc,
    char **argv);
```

The arguments *interp*, *canvas*, and *itemPtr* all have the standard meanings, and *argc* and *argv* describe the coordinate arguments. For example, if the following widget command is invoked:

```
.c coords 2 30 90
```

argc will be **2** and **argv** will contain the string values **30** and **90**.

The *coordProc* procedure should process the new coordinates, update the item appropriately (e.g., it must reset the bounding box in the item's header), and return a standard Tcl completion code. If an error occurs, *coordProc* must leave an error message in *interp->result*.

DELETEPROC

typePtr->deleteProc is invoked by Tk to delete an item and free any resources allocated to it. It must match the following prototype:

```
typedef void Tk_ItemDeleteProc(
    Tk_Canvas canvas,
    Tk_Item *itemPtr,
    Display *display);
```

The *canvas* and *itemPtr* arguments have the usual interpretations, and *display* identifies the X display containing the canvas. *deleteProc* must free up any resources allocated for the item, so that Tk can free the item record. *deleteProc* should not actually free the item record; this will be done by Tk when *deleteProc* returns.

DISPLAYPROC AND ALWAYSREDRAW

typePtr->displayProc is invoked by Tk to redraw an item on the screen. It must match the following prototype:

```
typedef void Tk_ItemDisplayProc(
    Tk_Canvas canvas,
    Tk_Item *itemPtr,
    Display *display,
    Drawable dst,
    int x,
    int y,
    int width,
    int height);
```

The *canvas* and *itemPtr* arguments have the usual meaning. *display* identifies the display containing the canvas, and *dst* specifies a drawable in which the item should be rendered; typically this is an off-screen

pixmap, which Tk will copy into the canvas's window once all relevant items have been drawn. *x*, *y*, *width*, and *height* specify a rectangular region in canvas coordinates, which is the area to be redrawn; only information that overlaps this area needs to be redrawn. Tk will not call *displayProc* unless the item's bounding box overlaps the redraw area, but the type manager may wish to use the redraw area to optimize the redisplay of the item.

Because of scrolling and the use of off-screen pixmaps for double-buffered redisplay, the item's coordinates in *dst* will not necessarily be the same as those in the canvas. *displayProc* should call **Tk_CanvasDrawableCoords** to transform coordinates from those of the canvas to those of *dst*.

Normally an item's *displayProc* is only invoked if the item overlaps the area being displayed. However, if *typePtr->alwaysRedraw* has a non-zero value, then *displayProc* is invoked during every redisplay operation, even if the item doesn't overlap the area of redisplay. *alwaysRedraw* should normally be set to 0; it is only set to 1 in special cases such as window items that need to be unmapped when they are off-screen.

POINTPROC

typePtr->pointProc is invoked by Tk to find out how close a given point is to a canvas item. Tk uses this procedure for purposes such as locating the item under the mouse or finding the closest item to a given point. The procedure must match the following prototype:

```
typedef double Tk_ItemPointProc(
    Tk_Canvas canvas,
    Tk_Item *itemPtr,
    double *pointPtr);
```

canvas and *itemPtr* have the usual meaning. *pointPtr* points to an array of two numbers giving the x and y coordinates of a point. *pointProc* must return a real value giving the distance from the point to the item, or 0 if the point lies inside the item.

AREAPROC

typePtr->areaProc is invoked by Tk to find out the relationship between an item and a rectangular area. It must match the following prototype:

```
typedef int Tk_ItemAreaProc(
    Tk_Canvas canvas,
    Tk_Item *itemPtr,
    double *rectPtr);
```

canvas and *itemPtr* have the usual meaning. *rectPtr* points to an array of four real numbers; the first two give the x and y coordinates of the upper left corner of a rectangle, and the second two give the x and y coordinates of the lower right corner. *areaProc* must return −1 if the item lies entirely outside the given area, 0 if it lies partially inside and partially outside the area, and 1 if it lies entirely inside the area.

POSTSCRIPTPROC

typePtr->postscriptProc is invoked by Tk to generate Postcript for an item during the **postscript** widget command. If the type manager is not capable of generating Postscript then *typePtr->postscriptProc* should be NULL. The procedure must match the following prototype:

```
typedef int Tk_ItemPostscriptProc(
    Tcl_Interp *interp,
    Tk_Canvas canvas,
    Tk_Item *itemPtr,
    int prepass);
```

The *interp*, *canvas*, and *itemPtr* arguments all have standard meanings; *prepass* will be described below. If *postscriptProc* completes successfully, it should append Postscript for the item to the information in *interp->result* (e.g. by calling **Tcl_AppendResult**, not **Tcl_SetResult**) and return TCL_OK. If an error occurs, *postscriptProc* should clear the result and replace its contents with an error message; then it should

return TCL_ERROR.

Tk provides a collection of utility procedures to simplify *postscriptProc*. For example, **Tk_CanvasPsColor** will generate Postscript to set the current color to a given Tk color and **Tk_CanvasPsFont** will set up font information. When generating Postscript, the type manager is free to change the graphics state of the Postscript interpreter, since Tk places **gsave** and **grestore** commands around the Postscript for the item. The type manager can use canvas x coordinates directly in its Postscript, but it must call **Tk_CanvasPsY** to convert y coordinates from the space of the canvas (where the origin is at the upper left) to the space of Postscript (where the origin is at the lower left).

In order to generate Postscript that complies with the Adobe Document Structuring Conventions, Tk actually generates Postscript in two passes. It calls each item's *postscriptProc* in each pass. The only purpose of the first pass is to collect font information (which is done by **Tk_CanvPsFont**); the actual Postscript is discarded. Tk sets the *prepass* argument to *postscriptProc* to 1 during the first pass; the type manager can use *prepass* to skip all Postscript generation except for calls to **Tk_CanvasPsFont**. During the second pass *prepass* will be 0, so the type manager must generate complete Postscript.

SCALEPROC

typePtr->scaleProc is invoked by Tk to rescale a canvas item during the **scale** widget command. The procedure must match the following prototype:

```
typedef void Tk_ItemScaleProc(
    Tk_Canvas canvas,
    Tk_Item *itemPtr,
    double originX,
    double originY,
    double scaleX,
    double scaleY);
```

The *canvas* and *itemPtr* arguments have the usual meaning. *originX* and *originY* specify an origin relative to which the item is to be scaled, and *scaleX* and *scaleY* give the x and y scale factors. The item should adjust its coordinates so that a point in the item that used to have coordinates *x* and *y* will have new coordinates *x'* and *y'*, where

x' = originX + scaleX(x-originX)*
y' = originY + scaleY(y-originY)*

scaleProc must also update the bounding box in the item's header.

TRANSLATEPROC

typePtr->translateProc is invoked by Tk to translate a canvas item during the **move** widget command. The procedure must match the following prototype:

```
typedef void Tk_ItemTranslateProc(
    Tk_Canvas canvas,
    Tk_Item *itemPtr,
    double deltaX,
    double deltaY);
```

The *canvas* and *itemPtr* arguments have the usual meaning, and *deltaX* and *deltaY* give the amounts that should be added to each x and y coordinate within the item. The type manager should adjust the item's coordinates and update the bounding box in the item's header.

INDEXPROC

typePtr->indexProc is invoked by Tk to translate a string index specification into a numerical index, for example during the **index** widget command. It is only relevant for item types that support indexable text; *typePtr->indexProc* may be specified as NULL for non-textual item types. The procedure must match the following prototype:

```
typedef int Tk_ItemIndexProc(
    Tcl_Interp *interp,
    Tk_Canvas canvas,
    Tk_Item *itemPtr,
    char indexString,
    int *indexPtr);
```

The *interp*, *canvas*, and *itemPtr* arguments all have the usual meaning. *indexString* contains a textual description of an index, and *indexPtr* points to an integer value that should be filled in with a numerical index. It is up to the type manager to decide what forms of index are supported (e.g., numbers, **insert**, **sel.first**, **end**, etc.). *indexProc* should return a Tcl completion code and set *interp->result* in the event of an error.

ICURSORPROC

typePtr->icursorProc is invoked by Tk during the **icursor** widget command to set the position of the insertion cursor in a textual item. It is only relevant for item types that support an insertion cursor; *typePtr->icursorProc* may be specified as NULL for item types that don't support an insertion cursor. The procedure must match the following prototype:

```
typedef void Tk_ItemIndexProc(
    Tk_Canvas canvas,
    Tk_Item *itemPtr,
    int index);
```

canvas and *itemPtr* have the usual meanings, and *index* is an index into the item's text, as returned by a previous call to *typePtr->insertProc*. The type manager should position the insertion cursor in the item just before the character given by *index*. Whether or not to actually display the insertion cursor is determined by other information provided by **Tk_CanvasGetTextInfo**.

SELECTIONPROC

typePtr->selectionProc is invoked by Tk during selection retrievals; it must return part or all of the selected text in the item (if any). It is only relevant for item types that support text; *typePtr->selectionProc* may be specified as NULL for non-textual item types. The procedure must match the following prototype:

```
typedef int Tk_ItemSelectionProc(
    Tk_Canvas canvas,
    Tk_Item *itemPtr,
    int offset,
    char *buffer,
    int maxBytes);
```

canvas and *itemPtr* have the usual meanings. *offset* is an offset in bytes into the selection where 0 refers to the first byte of the selection; it identifies the first character that is to be returned in this call. *buffer* points to an area of memory in which to store the requested bytes, and *maxBytes* specifies the maximum number of bytes to return. *selectionProc* should extract up to *maxBytes* characters from the selection and copy them to *maxBytes*; it should return a count of the number of bytes actually copied, which may be less than *maxBytes* if there aren't *offset+maxBytes* bytes in the selection.

INSERTPROC

typePtr->insertProc is invoked by Tk during the **insert** widget command to insert new text into a canvas item. It is only relevant for item types that support text; *typePtr->insertProc* may be specified as NULL for non-textual item types. The procedure must match the following prototype:

```
typedef void Tk_ItemInsertProc(
    Tk_Canvas canvas,
    Tk_Item *itemPtr,
```

```
int index,
char *string);
```

canvas and *itemPtr* have the usual meanings. *index* is an index into the item's text, as returned by a previous call to *typePtr->insertProc*, and *string* contains new text to insert just before the character given by *index*. The type manager should insert the text and recompute the bounding box in the item's header.

DCHARSPROC

typePtr->dCharsProc is invoked by Tk during the **dchars** widget command to delete a range of text from a canvas item. It is only relevant for item types that support text; *typePtr->dCharsProc* may be specified as NULL for non-textual item types. The procedure must match the following prototype:

```
typedef void Tk_ItemDCharsProc(
    Tk_Canvas canvas,
    Tk_Item *itemPtr,
    int first,
    int last);
```

canvas and *itemPtr* have the usual meanings. *first* and *last* give the indices of the first and last bytes to be deleted, as returned by previous calls to *typePtr->indexProc*. The type manager should delete the specified characters and update the bounding box in the item's header.

SEE ALSO

Tk_CanvasPsY, Tk_CanvasTextInfo, Tk_CanvasTkwin

KEYWORDS

canvas, focus, item type, selection, type manager

NAME

Tk_CreatePhotoImageFormat − define new file format for photo images

SYNOPSIS

#include <tk.h>
#include <tkPhoto.h>

Tk_CreatePhotoImageFormat(*formatPtr*)

ARGUMENTS

Tk_PhotoImageFormat	∗*formatPtr*	(in)	Structure that defines the new file format.

DESCRIPTION

Tk_CreatePhotoImageFormat is invoked to define a new file format for image data for use with photo images. The code that implements an image file format is called an image file format handler, or handler for short. The photo image code maintains a list of handlers that can be used to read and write data to or from a file. Some handlers may also support reading image data from a string or converting image data to a string format. The user can specify which handler to use with the **−format** image configuration option or the **−format** option to the **read** and **write** photo image subcommands.

An image file format handler consists of a collection of procedures plus a Tk_PhotoImageFormat structure, which contains the name of the image file format and pointers to six procedures provided by the handler to deal with files and strings in this format. The Tk_PhotoImageFormat structure contains the following fields:

```
typedef struct Tk_PhotoImageFormat {
    char *name;
    Tk_ImageFileMatchProc *fileMatchProc;
    Tk_ImageStringMatchProc *stringMatchProc;
    Tk_ImageFileReadProc *fileReadProc;
    Tk_ImageStringReadProc *stringReadProc;
    Tk_ImageFileWriteProc *fileWriteProc;
    Tk_ImageStringWriteProc *stringWriteProc;
} Tk_PhotoImageFormat;
```

The handler need not provide implementations of all six procedures. For example, the procedures that handle string data would not be provided for a format in which the image data are stored in binary, and could therefore contain null characters. If any procedure is not implemented, the corresponding pointer in the Tk_PhotoImageFormat structure should be set to NULL. The handler must provide the *fileMatchProc* procedure if it provides the *fileReadProc* procedure, and the *stringMatchProc* procedure if it provides the *stringReadProc* procedure.

NAME

formatPtr->name provides a name for the image type. Once **Tk_CreatePhotoImageFormat** returns, this name may be used in the **−format** photo image configuration and subcommand option. The manual page for the photo image (photo(n)) describes how image file formats are chosen based on their names and the value given to the **−format** option.

FILEMATCHPROC

formatPtr->fileMatchProc provides the address of a procedure for Tk to call when it is searching for an image file format handler suitable for reading data in a given file. *formatPtr->fileMatchProc* must match the following prototype:

```
typedef int Tk_ImageFileMatchProc(
```

Tcl_Channel *chan*,
char ∗*fileName*,
char ∗*formatString*,
int ∗*widthPtr*,
int ∗*heightPtr*);

The *fileName* argument is the name of the file containing the image data, which is open for reading as *chan*. The *formatString* argument contains the value given for the **−format** option, or NULL if the option was not specified. If the data in the file appears to be in the format supported by this handler, the *formatPtr->fileMatchProc* procedure should store the width and height of the image in ∗*widthPtr* and ∗*heightPtr* respectively, and return 1. Otherwise it should return 0.

STRINGMATCHPROC

formatPtr->stringMatchProc provides the address of a procedure for Tk to call when it is searching for an image file format handler for suitable for reading data from a given string. *formatPtr->stringMatchProc* must match the following prototype:

typedef int Tk_ImageStringMatchProc(
char ∗*string*,
char ∗*formatString*,
int ∗*widthPtr*,
int ∗*heightPtr*);

The *string* argument points to the string containing the image data. The *formatString* argument contains the value given for the **−format** option, or NULL if the option was not specified. If the data in the string appears to be in the format supported by this handler, the *formatPtr->stringMatchProc* procedure should store the width and height of the image in ∗*widthPtr* and ∗*heightPtr* respectively, and return 1. Otherwise it should return 0.

FILEREADPROC

formatPtr->fileReadProc provides the address of a procedure for Tk to call to read data from an image file into a photo image. *formatPtr->fileReadProc* must match the following prototype:

typedef int Tk_ImageFileReadProc(
Tcl_Interp ∗*interp*,
Tcl_Channel *chan*,
char ∗*fileName*,
char ∗*formatString*,
PhotoHandle *imageHandle*,
int *destX*, int *destY*,
int *width*, int *height*,
int *srcX*, int *srcY*);

The *interp* argument is the interpreter in which the command was invoked to read the image; it should be used for reporting errors. The image data is in the file named *fileName*, which is open for reading as *chan*. The *formatString* argument contains the value given for the **−format** option, or NULL if the option was not specified. The image data in the file, or a subimage of it, is to be read into the photo image identified by the handle *imageHandle*. The subimage of the data in the file is of dimensions *width* x *height* and has its top-left corner at coordinates (*srcX*,*srcY*). It is to be stored in the photo image with its top-left corner at coordinates (*destX*,*destY*) using the **Tk_PhotoPutBlock** procedure. The return value is a standard Tcl return value.

STRINGREADPROC

formatPtr->stringReadProc provides the address of a procedure for Tk to call to read data from a string into a photo image. *formatPtr->stringReadProc* must match the following prototype:

```
typedef int Tk_ImageStringReadProc(
    Tcl_Interp *interp,
    char *string,
    char *formatString,
    PhotoHandle imageHandle,
    int destX, int destY,
    int width, int height,
    int srcX, int srcY);
```

The *interp* argument is the interpreter in which the command was invoked to read the image; it should be used for reporting errors. The *string* argument points to the image data in string form. The *formatString* argument contains the value given for the **−format** option, or NULL if the option was not specified. The image data in the string, or a subimage of it, is to be read into the photo image identified by the handle *imageHandle*. The subimage of the data in the string is of dimensions *width* x *height* and has its top-left corner at coordinates (*srcX*,*srcY*). It is to be stored in the photo image with its top-left corner at coordinates (*destX*,*destY*) using the **Tk_PhotoPutBlock** procedure. The return value is a standard Tcl return value.

FILEWRITEPROC

formatPtr->fileWriteProc provides the address of a procedure for Tk to call to write data from a photo image to a file. *formatPtr->fileWriteProc* must match the following prototype:

```
typedef int Tk_ImageFileWriteProc(
    Tcl_Interp *interp,
    char *fileName,
    char *formatString,
    Tk_PhotoImageBlock *blockPtr);
```

The *interp* argument is the interpreter in which the command was invoked to write the image; it should be used for reporting errors. The image data to be written are in memory and are described by the Tk_PhotoImageBlock structure pointed to by *blockPtr*; see the manual page FindPhoto(3) for details. The *fileName* argument points to the string giving the name of the file in which to write the image data. The *formatString* argument contains the value given for the **−format** option, or NULL if the option was not specified. The format string can contain extra characters after the name of the format. If appropriate, the *formatPtr->fileWriteProc* procedure may interpret these characters to specify further details about the image file. The return value is a standard Tcl return value.

STRINGWRITEPROC

formatPtr->stringWriteProc provides the address of a procedure for Tk to call to translate image data from a photo image into a string. *formatPtr->stringWriteProc* must match the following prototype:

```
typedef int Tk_ImageStringWriteProc(
    Tcl_Interp *interp,
    Tcl_DString *dataPtr,
    char *formatString,
    Tk_PhotoImageBlock *blockPtr);
```

The *interp* argument is the interpreter in which the command was invoked to convert the image; it should be used for reporting errors. The image data to be converted are in memory and are described by the Tk_PhotoImageBlock structure pointed to by *blockPtr*; see the manual page FindPhoto(3) for details. The data for the string should be appended to the dynamic string given by *dataPtr*. The *formatString* argument contains the value given for the **−format** option, or NULL if the option was not specified. The format string can contain extra characters after the name of the format. If appropriate, the *formatPtr->stringWriteProc* procedure may interpret these characters to specify further details about the image file. The return value is a standard Tcl return value.

SEE ALSO

Tk_FindPhoto, Tk_PhotoPutBlock

KEYWORDS

photo image, image file

NAME

Tk_CreateSelHandler, Tk_DeleteSelHandler − arrange to handle requests for a selection

SYNOPSIS

#include <tk.h>

Tk_CreateSelHandler(*tkwin, selection, target, proc, clientData, format*)

Tk_DeleteSelHandler(*tkwin, selection, target*)

ARGUMENTS

Tk_Window	*tkwin*	(in)	Window for which *proc* will provide selection information.
Atom	*selection*	(in)	The name of the selection for which *proc* will provide selection information.
Atom	*target*	(in)	Form in which *proc* can provide the selection (e.g. STRING or FILE_NAME). Corresponds to *type* arguments in **selection** commands.
Tk_SelectionProc	∗*proc*	(in)	Procedure to invoke whenever the selection is owned by *tkwin* and the selection contents are requested in the format given by *target*.
ClientData	*clientData*	(in)	Arbitrary one-word value to pass to *proc*.
Atom	*format*	(in)	If the selection requestor isn't in this process, *format* determines the representation used to transmit the selection to its requestor.

DESCRIPTION

Tk_CreateSelHandler arranges for a particular procedure (*proc*) to be called whenever *selection* is owned by *tkwin* and the selection contents are requested in the form given by *target*. *Target* should be one of the entries defined in the left column of Table 2 of the X Inter-Client Communication Conventions Manual (ICCCM) or any other form in which an application is willing to present the selection. The most common form is STRING.

Proc should have arguments and result that match the type **Tk_SelectionProc**:

```
typedef int Tk_SelectionProc(
    ClientData clientData,
    int offset,
    char *buffer,
    int maxBytes);
```

The *clientData* parameter to *proc* is a copy of the *clientData* argument given to **Tk_CreateSelHandler**. Typically, *clientData* points to a data structure containing application-specific information that is needed to retrieve the selection. *Offset* specifies an offset position into the selection, *buffer* specifies a location at which to copy information about the selection, and *maxBytes* specifies the amount of space available at *buffer*. *Proc* should place a NULL-terminated string at *buffer* containing *maxBytes* or fewer characters (not including the terminating NULL), and it should return a count of the number of non-NULL characters stored at *buffer*. If the selection no longer exists (e.g. it once existed but the user deleted the range of characters containing it), then *proc* should return -1.

When transferring large selections, Tk will break them up into smaller pieces (typically a few thousand bytes each) for more efficient transmission. It will do this by calling *proc* one or more times, using successively higher values of *offset* to retrieve successive portions of the selection. If *proc* returns a count less than *maxBytes* it means that the entire remainder of the selection has been returned. If *proc*'s return value is

maxBytes it means there may be additional information in the selection, so Tk must make another call to *proc* to retrieve the next portion.

Proc always returns selection information in the form of a character string. However, the ICCCM allows for information to be transmitted from the selection owner to the selection requestor in any of several formats, such as a string, an array of atoms, an array of integers, etc. The *format* argument to **Tk_CreateSelHandler** indicates what format should be used to transmit the selection to its requestor (see the middle column of Table 2 of the ICCCM for examples). If *format* is not STRING, then Tk will take the value returned by *proc* and divided it into fields separated by white space. If *format* is ATOM, then Tk will return the selection as an array of atoms, with each field in *proc*'s result treated as the name of one atom. For any other value of *format*, Tk will return the selection as an array of 32-bit values where each field of *proc*'s result is treated as a number and translated to a 32-bit value. In any event, the *format* atom is returned to the selection requestor along with the contents of the selection.

If **Tk_CreateSelHandler** is called when there already exists a handler for *selection* and *target* on *tkwin*, then the existing handler is replaced with a new one.

Tk_DeleteSelHandler removes the handler given by *tkwin*, *selection*, and *target*, if such a handler exists. If there is no such handler then it has no effect.

KEYWORDS

format, handler, selection, target

NAME

Tk_CreateWindow, Tk_CreateWindowFromPath, Tk_DestroyWindow, Tk_MakeWindowExist − create or delete window

SYNOPSIS

#include <tk.h>

Tk_Window
Tk_CreateWindow(*interp, parent, name, topLevScreen*)

Tk_Window
Tk_CreateWindowFromPath(*interp, tkwin, pathName, topLevScreen*)

Tk_DestroyWindow(*tkwin*)

Tk_MakeWindowExist(*tkwin*)

ARGUMENTS

Tcl_Interp	∗*interp*	(out)	Tcl interpreter to use for error reporting. If no error occurs, then ∗*interp* isn't modified.
Tk_Window	*parent*	(in)	Token for the window that is to serve as the logical parent of the new window.
char	∗*name*	(in)	Name to use for this window. Must be unique among all children of the same *parent*.
char	∗*topLevScreen*	(in)	Has same format as *screenName*. If NULL, then new window is created as an internal window. If non-NULL, new window is created as a top-level window on screen *topLevScreen*. If *topLevScreen* is an empty string ("") then new window is created as top-level window of *parent*'s screen.
Tk_Window	*tkwin*	(in)	Token for window.
char	∗*pathName*	(in)	Name of new window, specified as path name within application (e.g. **.a.b.c**).

DESCRIPTION

The procedures **Tk_CreateWindow** and **Tk_CreateWindowFromPath** are used to create new windows for use in Tk-based applications. Each of the procedures returns a token that can be used to manipulate the window in other calls to the Tk library. If the window couldn't be created successfully, then NULL is returned and *interp->result* is modified to hold an error message.

Tk supports two different kinds of windows: internal windows and top-level windows. An internal window is an interior window of a Tk application, such as a scrollbar or menu bar or button. A top-level window is one that is created as a child of a screen's root window, rather than as an interior window, but which is logically part of some existing main window. Examples of top-level windows are pop-up menus and dialog boxes.

New windows may be created by calling **Tk_CreateWindow**. If the *topLevScreen* argument is NULL, then the new window will be an internal window. If *topLevScreen* is non-NULL, then the new window will be a top-level window: *topLevScreen* indicates the name of a screen and the new window will be created as a child of the root window of *topLevScreen*. In either case Tk will consider the new window to be the logical child of *parent*: the new window's path name will reflect this fact, options may be specified for the new window under this assumption, and so on. The only difference is that new X window for a top-level

window will not be a child of *parent*'s X window. For example, a pull-down menu's *parent* would be the button-like window used to invoke it, which would in turn be a child of the menu bar window. A dialog box might have the application's main window as its parent.

Tk_CreateWindowFromPath offers an alternate way of specifying new windows. In **Tk_CreateWindowFromPath** the new window is specified with a token for any window in the target application (*tkwin*), plus a path name for the new window. It produces the same effect as **Tk_CreateWindow** and allows both top-level and internal windows to be created, depending on the value of *topLevScreen*. In calls to **Tk_CreateWindowFromPath**, as in calls to **Tk_CreateWindow**, the parent of the new window must exist at the time of the call, but the new window must not already exist.

The window creation procedures don't actually issue the command to X to create a window. Instead, they create a local data structure associated with the window and defer the creation of the X window. The window will actually be created by the first call to **Tk_MapWindow**. Deferred window creation allows various aspects of the window (such as its size, background color, etc.) to be modified after its creation without incurring any overhead in the X server. When the window is finally mapped all of the window attributes can be set while creating the window.

The value returned by a window-creation procedure is not the X token for the window (it can't be, since X hasn't been asked to create the window yet). Instead, it is a token for Tk's local data structure for the window. Most of the Tk library procedures take Tk_Window tokens, rather than X identifiers. The actual X window identifier can be retrieved from the local data structure using the **Tk_WindowId** macro; see the manual entry for **Tk_WindowId** for details.

Tk_DestroyWindow deletes a window and all the data structures associated with it, including any event handlers created with **Tk_CreateEventHandler**. In addition, **Tk_DestroyWindow** will delete any children of *tkwin* recursively (where children are defined in the Tk sense, consisting of all windows that were created with the given window as *parent*). If *tkwin* was created by **Tk_CreateInternalWindow** then event handlers interested in destroy events are invoked immediately. If *tkwin* is a top-level or main window, then the event handlers will be invoked later, after X has seen the request and returned an event for it.

If a window has been created but hasn't been mapped, so no X window exists, it is possible to force the creation of the X window by calling **Tk_MakeWindowExist**. This procedure issues the X commands to instantiate the window given by *tkwin*.

KEYWORDS

create, deferred creation, destroy, display, internal window, screen, top-level window, window

NAME

Tk_DeleteImage − Destroy an image.

SYNOPSIS

#include <tk.h>

Tk_DeleteImage(*interp, name*)

ARGUMENTS

Tcl_Interp	∗*interp*	(in)	Interpreter for which the image was created.
char	∗*name*	(in)	Name of the image.

DESCRIPTION

Tk_DeleteImage deletes the image given by *interp* and *name*, if there is one. All instances of that image will redisplay as empty regions. If the given image does not exist then the procedure has no effect.

KEYWORDS

delete image, image manager

NAME

Tk_DrawFocusHighlight − draw the traversal highlight ring for a widget

SYNOPSIS

#include <tk.h>

Tk_GetPixels(*tkwin, gc, width, drawable***)**

ARGUMENTS

Tk_Window	*tkwin*	(in)	Window for which the highlight is being drawn. Used to retrieve the window's dimensions, among other things.
GC	*gc*	(in)	Graphics context to use for drawing the highlight.
int	*width*	(in)	Width of the highlight ring, in pixels.
Drawable	*drawable*	(in)	Drawable in which to draw the highlight; usually an offscreen pixmap for double buffering.

DESCRIPTION

Tk_DrawFocusHighlight is a utility procedure that draws the traversal highlight ring for a widget. It is typically invoked by widgets during redisplay.

KEYWORDS

focus, traversal highlight

NAME

Tk_CreateEventHandler, Tk_DeleteEventHandler − associate procedure callback with an X event

SYNOPSIS

#include <tk.h>

Tk_CreateEventHandler(*tkwin, mask, proc, clientData*)

Tk_DeleteEventHandler(*tkwin, mask, proc, clientData*)

ARGUMENTS

Tk_Window	*tkwin*	(in)	Token for window in which events may occur.
unsigned long	*mask*	(in)	Bit-mask of events (such as **ButtonPressMask**) for which *proc* should be called.
Tk_EventProc	∗*proc*	(in)	Procedure to invoke whenever an event in *mask* occurs in the window given by *tkwin*.
ClientData	*clientData*	(in)	Arbitrary one-word value to pass to *proc*.

DESCRIPTION

Tk_CreateEventHandler arranges for *proc* to be invoked in the future whenever one of the event types specified by *mask* occurs in the window specified by *tkwin*. The callback to *proc* will be made by **Tk_HandleEvent**; this mechanism only works in programs that dispatch events through **Tk_HandleEvent** (or through other Tk procedures that call **Tk_HandleEvent**, such as **Tk_DoOneEvent** or **Tk_MainLoop**).

Proc should have arguments and result that match the type **Tk_EventProc**:

```
typedef void Tk_EventProc(
    ClientData clientData,
    XEvent *eventPtr);
```

The *clientData* parameter to *proc* is a copy of the *clientData* argument given to **Tk_CreateEventHandler** when the callback was created. Typically, *clientData* points to a data structure containing application-specific information about the window in which the event occurred. *EventPtr* is a pointer to the X event, which will be one of the ones specified in the *mask* argument to **Tk_CreateEventHandler**.

Tk_DeleteEventHandler may be called to delete a previously-created event handler: it deletes the first handler it finds that is associated with *tkwin* and matches the *mask*, *proc*, and *clientData* arguments. If no such handler exists, then **Tk_EventHandler** returns without doing anything. Although Tk supports it, it's probably a bad idea to have more than one callback with the same *mask*, *proc*, and *clientData* arguments. When a window is deleted all of its handlers will be deleted automatically; in this case there is no need to call **Tk_DeleteEventHandler**.

If multiple handlers are declared for the same type of X event on the same window, then the handlers will be invoked in the order they were created.

KEYWORDS

bind, callback, event, handler

NAME

Tk_FindPhoto, Tk_PhotoPutBlock, Tk_PhotoPutZoomedBlock, Tk_PhotoGetImage, Tk_PhotoBlank, Tk_PhotoExpand, Tk_PhotoGetSize, Tk_PhotoSetSize − manipulate the image data stored in a photo image.

SYNOPSIS

#include <tk.h>
#include <tkPhoto.h>

Tk_PhotoHandle
Tk_FindPhoto(*interp, imageName*)

void
Tk_PhotoPutBlock(*handle, blockPtr, x, y, width, height*)

void
Tk_PhotoPutZoomedBlock(*handle, blockPtr, x, y, width, height,zoomX, zoomY, subsampleX, subsampleY*)

int
Tk_PhotoGetImage(*handle, blockPtr*)

void
Tk_PhotoBlank(*handle*)

void
Tk_PhotoExpand(*handle, width, height*)

void
Tk_PhotoGetSize(*handle, widthPtr, heightPtr*)

void
Tk_PhotoSetSize(*handle, width, height*)

ARGUMENTS

Tcl_Interp	∗*interp*	(in)	Interpreter in which image was created.
char	∗*imageName*	(in)	Name of the photo image.
Tk_PhotoHandle	*handle*	(in)	Opaque handle identifying the photo image to be affected.
Tk_PhotoImageBlock	∗*blockPtr*	(in)	Specifies the address and storage layout of image data.
int	*x*	(in)	Specifies the X coordinate where the top-left corner of the block is to be placed within the image.
int	*y*	(in)	Specifies the Y coordinate where the top-left corner of the block is to be placed within the image.
int	*width*	(in)	Specifies the width of the image area to be affected (for **Tk_PhotoPutBlock**) or the desired image width (for **Tk_PhotoExpand** and **Tk_PhotoSetSize**).
int	*height*	(in)	Specifies the height of the image area to be affected (for **Tk_PhotoPutBlock**) or the desired image height (for **Tk_PhotoExpand** and **Tk_PhotoSetSize**).
int	∗*widthPtr*	(out)	Pointer to location in which to store the image width.

int	**heightPtr*	(out)	Pointer to location in which to store the image height.
int	*subsampleX*	(in)	Specifies the subsampling factor in the X direction for input image data.
int	*subsampleY*	(in)	Specifies the subsampling factor in the Y direction for input image data.
int	*zoomX*	(in)	Specifies the zoom factor to be applied in the X direction to pixels being written to the photo image.
int	*zoomY*	(in)	Specifies the zoom factor to be applied in the Y direction to pixels being written to the photo image.

DESCRIPTION

Tk_FindPhoto returns an opaque handle that is used to identify a particular photo image to the other procedures. The parameter is the name of the image, that is, the name specified to the **image create photo** command, or assigned by that command if no name was specified.

Tk_PhotoPutBlock is used to supply blocks of image data to be displayed. The call affects an area of the image of size *width* x *height* pixels, with its top-left corner at coordinates (*x*,*y*). All of *width*, *height*, *x*, and *y* must be non-negative. If part of this area lies outside the current bounds of the image, the image will be expanded to include the area, unless the user has specified an explicit image size with the **−width** and/or **−height** widget configuration options (see photo(n)); in that case the area is silently clipped to the image boundaries.

The *block* parameter is a pointer to a **Tk_PhotoImageBlock** structure, defined as follows:

```
typedef struct {
    unsigned char *pixelPtr;
    int width;
    int height;
    int pitch;
    int pixelSize;
    int offset[3];
} Tk_PhotoImageBlock;
```

The *pixelPtr* field points to the first pixel, that is, the top-left pixel in the block. The *width* and *height* fields specify the dimensions of the block of pixels. The *pixelSize* field specifies the address difference between two horizontally adjacent pixels. Often it is 3 or 4, but it can have any value. The *pitch* field specifies the address difference between two vertically adjacent pixels. The *offset* array contains the offsets from the address of a pixel to the addresses of the bytes containing the red, green and blue components. These are normally 0, 1 and 2, but can have other values, e.g., for images that are stored as separate red, green and blue planes.

The value given for the *width* and *height* parameters to **Tk_PhotoPutBlock** do not have to correspond to the values specified in *block*. If they are smaller, **Tk_PhotoPutBlock** extracts a sub-block from the image data supplied. If they are larger, the data given are replicated (in a tiled fashion) to fill the specified area. These rules operate independently in the horizontal and vertical directions.

Tk_PhotoPutZoomedBlock works like **Tk_PhotoPutBlock** except that the image can be reduced or enlarged for display. The *subsampleX* and *subsampleY* parameters allow the size of the image to be reduced by subsampling. **Tk_PhotoPutZoomedBlock** will use only pixels from the input image whose X coordinates are multiples of *subsampleX*, and whose Y coordinates are multiples of *subsampleY*. For example, an image of 512x512 pixels can be reduced to 256x256 by setting *subsampleX* and *subsampleY* to 2.

The *zoomX* and *zoomY* parameters allow the image to be enlarged by pixel replication. Each pixel of the (possibly subsampled) input image will be written to a block *zoomX* pixels wide and *zoomY* pixels high of the displayed image. Subsampling and zooming can be used together for special effects.

Tk_PhotoGetImage can be used to retrieve image data from a photo image. **Tk_PhotoGetImage** fills in the structure pointed to by the *blockPtr* parameter with values that describe the address and layout of the image data that the photo image has stored internally. The values are valid until the image is destroyed or its size is changed. **Tk_PhotoGetImage** returns 1 for compatibility with the corresponding procedure in the old photo widget.

Tk_PhotoBlank blanks the entire area of the photo image. Blank areas of a photo image are transparent.

Tk_PhotoExpand requests that the widget's image be expanded to be at least *width* x *height* pixels in size. The width and/or height are unchanged if the user has specified an explicit image width or height with the **−width** and/or **−height** configuration options, respectively. If the image data are being supplied in many small blocks, it is more efficient to use **Tk_PhotoExpand** or **Tk_PhotoSetSize** at the beginning rather than allowing the image to expand in many small increments as image blocks are supplied.

Tk_PhotoSetSize specifies the size of the image, as if the user had specified the given *width* and *height* values to the **−width** and **−height** configuration options. A value of zero for *width* or *height* does not change the image's width or height, but allows the width or height to be changed by subsequent calls to **Tk_PhotoPutBlock**, **Tk_PhotoPutZoomedBlock** or **Tk_PhotoExpand**.

Tk_PhotoGetSize returns the dimensions of the image in ∗*widthPtr* and ∗*heightPtr*.

CREDITS

The code for the photo image type was developed by Paul Mackerras, based on his earlier photo widget code.

KEYWORDS

photo, image

NAME

Tk_FontId, Tk_FontMetrics, Tk_PostscriptFontName − accessor functions for fonts

SYNOPSIS

#include <tk.h>

Font
Tk_FontId(*tkfont***)**

void
Tk_GetFontMetrics(*tkfont, fmPtr***)**

int
Tk_PostscriptFontName(*tkfont, dsPtr***)**

ARGUMENTS

Tk_Font	*tkfont*	(in)	Opaque font token being queried. Must have been returned by a previous call to **Tk_GetFont**.
Tk_FontMetrics	∗*fmPtr*	(out)	Pointer to structure in which the font metrics for *tkfont* will be stored.
Tcl_DString	∗*dsPtr*	(out)	Pointer to an initialized **Tcl_DString** to which the name of the Postscript font that corresponds to *tkfont* will be appended.

DESCRIPTION

Given a *tkfont*, **Tk_FontId** returns the token that should be selected into an XGCValues structure in order to construct a graphics context that can be used to draw text in the specified font.

Tk_GetFontMetrics computes the ascent, descent, and linespace of the *tkfont* in pixels and stores those values in the structure pointer to by *fmPtr*. These values can be used in computations such as to space multiple lines of text, to align the baselines of text in different fonts, and to vertically align text in a given region. See the documentation for the **font** command for definitions of the terms ascent, descent, and linespace, used in font metrics.

Tk_PostscriptFontName maps a *tkfont* to the corresponding Postcript font name that should be used when printing. The return value is the size in points of the *tkfont* and the Postscript font name is appended to *dsPtr*. *DsPtr* must refer to an initialized **Tcl_DString**. Given a ''reasonable'' Postscript printer, the following screen font families should print correctly:

> **Avant Garde**, **Arial**, **Bookman**, **Courier**, **Courier New**, **Geneva**, **Helvetica**, **Monaco**, **New Century Schoolbook**, **New York**, **Palatino**, **Symbol**, **Times**, **Times New Roman**, **Zapf Chancery**, and **Zapf Dingbats**.

Any other font families may not print correctly because the computed Postscript font name may be incorrect or not exist on the printer.

DATA STRUCTURES

The Tk_FontMetrics data structure is used by Tk_GetFontMetrics to return information about a font and is defined as follows:

```
typedef struct Tk_FontMetrics {
    int ascent;
    int descent;
    int linespace;
} Tk_FontMetrics;
```

The *linespace* field is the amount in pixels that the tallest letter sticks up above the baseline, plus any extra blank space added by the designer of the font.

The *descent* is the largest amount in pixels that any letter sticks below the baseline, plus any extra blank space added by the designer of the font.

The *linespace* is the sum of the ascent and descent. How far apart two lines of text in the same font should be placed so that none of the characters in one line overlap any of the characters in the other line.

KEYWORDS

font

NAME

Tk_FreeXId − make X resource identifier available for reuse

SYNOPSIS

#include <tk.h>

Tk_FreeXId(*display, id***)**

ARGUMENTS

Display	∗*display*	(in)	Display for which *id* was allocated.
XID	*id*	(in)	Identifier of X resource (window, font, pixmap, cursor, graphics context, or colormap) that is no longer in use.

DESCRIPTION

The default allocator for resource identifiers provided by Xlib is very simple-minded and does not allow resource identifiers to be re-used. If a long-running application reaches the end of the resource id space, it will generate an X protocol error and crash. Tk replaces the default id allocator with its own allocator, which allows identifiers to be reused. In order for this to work, **Tk_FreeXId** must be called to tell the allocator about resources that have been freed. Tk automatically calls **Tk_FreeXId** whenever it frees a resource, so if you use procedures like **Tk_GetFontStruct**, **Tk_GetGC**, and **Tk_GetPixmap** then you need not call **Tk_FreeXId**. However, if you allocate resources directly from Xlib, for example by calling **XCreatePixmap**, then you should call **Tk_FreeXId** when you call the corresponding Xlib free procedure, such as **XFreePixmap**. If you don't call **Tk_FreeXId** then the resource identifier will be lost, which could cause problems if the application runs long enough to lose all of the available identifiers.

KEYWORDS

resource identifier

NAME

Tk_GeometryRequest, Tk_SetInternalBorder − specify desired geometry or internal border for a window

SYNOPSIS

#include <tk.h>

Tk_GeometryRequest(*tkwin, reqWidth, reqHeight*)

Tk_SetInternalBorder(*tkwin, width*)

ARGUMENTS

Tk_Window	*tkwin*	(in)	Window for which geometry is being requested.
int	*reqWidth*	(in)	Desired width for *tkwin*, in pixel units.
int	*reqHeight*	(in)	Desired height for *tkwin*, in pixel units.
int	*width*	(in)	Space to leave for internal border for *tkwin*, in pixel units.

DESCRIPTION

Tk_GeometryRequest is called by widget code to indicate its preference for the dimensions of a particular window. The arguments to **Tk_GeometryRequest** are made available to the geometry manager for the window, which then decides on the actual geometry for the window. Although geometry managers generally try to satisfy requests made to **Tk_GeometryRequest**, there is no guarantee that this will always be possible. Widget code should not assume that a geometry request will be satisfied until it receives a **ConfigureNotify** event indicating that the geometry change has occurred. Widget code should never call procedures like **Tk_ResizeWindow** directly. Instead, it should invoke **Tk_GeometryRequest** and leave the final geometry decisions to the geometry manager.

If *tkwin* is a top-level window, then the geometry information will be passed to the window manager using the standard ICCCM protocol.

Tk_SetInternalBorder is called by widget code to indicate that the widget has an internal border. This means that the widget draws a decorative border inside the window instead of using the standard X borders, which are external to the window's area. For example, internal borders are used to draw 3-D effects. *Width* specifies the width of the border in pixels. Geometry managers will use this information to avoid placing any children of *tkwin* overlapping the outermost *width* pixels of *tkwin*'s area.

The information specified in calls to **Tk_GeometryRequest** and **Tk_SetInternalBorder** can be retrieved using the macros **Tk_ReqWidth**, **Tk_ReqHeight**, and **Tk_InternalBorderWidth**. See the **Tk_WindowId** manual entry for details.

KEYWORDS

geometry, request

NAME

Tk_GetAnchor, Tk_NameOfAnchor − translate between strings and anchor positions

SYNOPSIS

```
#include <tk.h>

int
Tk_GetAnchor(interp, string, anchorPtr)

char *
Tk_NameOfAnchor(anchor)
```

ARGUMENTS

Tcl_Interp	**interp*	(in)	Interpreter to use for error reporting.
char	**string*	(in)	String containing name of anchor point: one of ‘‘n’’, ‘‘ne’’, ‘‘e’’, ‘‘se’’, ‘‘s’’, ‘‘sw’’, ‘‘w’’, ‘‘nw’’, or ‘‘center’’.
int	**anchorPtr*	(out)	Pointer to location in which to store anchor position corresponding to *string*.
Tk_Anchor	*anchor*	(in)	Anchor position, e.g. **TCL_ANCHOR_CENTER**.

DESCRIPTION

Tk_GetAnchor places in **anchorPtr* an anchor position (enumerated type **Tk_Anchor**) corresponding to *string*, which will be one of **TK_ANCHOR_N**, **TK_ANCHOR_NE**, **TK_ANCHOR_E**, **TK_ANCHOR_SE**, **TK_ANCHOR_S**, **TK_ANCHOR_SW**, **TK_ANCHOR_W**, **TK_ANCHOR_NW**, or **TK_ANCHOR_CENTER**. Anchor positions are typically used for indicating a point on an object that will be used to position that object, e.g. **TK_ANCHOR_N** means position the top center point of the object at a particular place.

Under normal circumstances the return value is **TCL_OK** and *interp* is unused. If *string* doesn’t contain a valid anchor position or an abbreviation of one of these names, then an error message is stored in *interp->result*, **TCL_ERROR** is returned, and **anchorPtr* is unmodified.

Tk_NameOfAnchor is the logical inverse of **Tk_GetAnchor**. Given an anchor position such as **TK_ANCHOR_N** it returns a statically-allocated string corresponding to *anchor*. If *anchor* isn’t a legal anchor value, then ‘‘unknown anchor position’’ is returned.

KEYWORDS

anchor position

NAME

Tk_GetBitmap, Tk_DefineBitmap, Tk_NameOfBitmap, Tk_SizeOfBitmap, Tk_FreeBitmap, Tk_GetBitmapFromData − maintain database of single-plane pixmaps

SYNOPSIS

#include <tk.h>

Pixmap
Tk_GetBitmap(*interp, tkwin, id***)**

int
Tk_DefineBitmap(*interp, nameId, source, width, height***)**

Tk_Uid
Tk_NameOfBitmap(*display, bitmap***)**

Tk_SizeOfBitmap(*display, bitmap, widthPtr, heightPtr***)**

Tk_FreeBitmap(*display, bitmap***)**

ARGUMENTS

Tcl_Interp	∗*interp*	(in)	Interpreter to use for error reporting.
Tk_Window	*tkwin*	(in)	Token for window in which the bitmap will be used.
Tk_Uid	*id*	(in)	Description of bitmap; see below for possible values.
Tk_Uid	*nameId*	(in)	Name for new bitmap to be defined.
char	∗*source*	(in)	Data for bitmap, in standard bitmap format. Must be stored in static memory whose value will never change.
int	*width*	(in)	Width of bitmap.
int	*height*	(in)	Height of bitmap.
int	∗*widthPtr*	(out)	Pointer to word to fill in with *bitmap*'s width.
int	∗*heightPtr*	(out)	Pointer to word to fill in with *bitmap*'s height.
Display	∗*display*	(in)	Display for which *bitmap* was allocated.
Pixmap	*bitmap*	(in)	Identifier for a bitmap allocated by **Tk_GetBitmap**.

DESCRIPTION

These procedures manage a collection of bitmaps (one-plane pixmaps) being used by an application. The procedures allow bitmaps to be re-used efficiently, thereby avoiding server overhead, and also allow bitmaps to be named with character strings.

Tk_GetBitmap takes as argument a Tk_Uid describing a bitmap. It returns a Pixmap identifier for a bitmap corresponding to the description. It re-uses an existing bitmap, if possible, and creates a new one otherwise. At present, *id* must have one of the following forms:

@fileName *FileName* must be the name of a file containing a bitmap description in the standard X11 or X10 format.

name *Name* must be the name of a bitmap defined previously with a call to **Tk_DefineBitmap**. The following names are pre-defined by Tk:

error The international "don't" symbol: a circle with a diagonal line

	across it.
gray75	75% gray: a checkerboard pattern where three out of four bits are on.
gray50	50% gray: a checkerboard pattern where every other bit is on.
gray25	25% gray: a checkerboard pattern where one out of every four bits is on.
gray12	12.5% gray: a pattern where one-eighth of the bits are on, consisting of every fourth pixel in every other row.
hourglass	An hourglass symbol.
info	A large letter ‘‘i’’.
questhead	The silhouette of a human head, with a question mark in it.
question	A large question-mark.
warning	A large exclamation point.

In addition, the following pre-defined names are available only on the **Macintosh** platform:

document	A generic document.
stationery	Document stationery.
edition	The *edition* symbol.
application	Generic application icon.
accessory	A desk accessory.
folder	Generic folder icon.
pfolder	A locked folder.
trash	A trash can.
floppy	A floppy disk.
ramdisk	A floppy disk with chip.
cdrom	A cd disk icon.
preferences	A folder with prefs symbol.
querydoc	A database document icon.
stop	A stop sign.
note	A face with ballon words.
caution	A triangle with an exclamation point.

Under normal conditions, **Tk_GetBitmap** returns an identifier for the requested bitmap. If an error occurs in creating the bitmap, such as when *id* refers to a non-existent file, then **None** is returned and an error message is left in *interp->result*.

Tk_DefineBitmap associates a name with in-memory bitmap data so that the name can be used in later calls to **Tk_GetBitmap**. The *nameId* argument gives a name for the bitmap; it must not previously have been used in a call to **Tk_DefineBitmap**. The arguments *source*, *width*, and *height* describe the bitmap. **Tk_DefineBitmap** normally returns TCL_OK; if an error occurs (e.g. a bitmap named *nameId* has already been defined) then TCL_ERROR is returned and an error message is left in *interp->result*. Note: **Tk_DefineBitmap** expects the memory pointed to by *source* to be static: **Tk_DefineBitmap** doesn’t make

a private copy of this memory, but uses the bytes pointed to by *source* later in calls to **Tk_GetBitmap**.

Typically **Tk_DefineBitmap** is used by **#include**-ing a bitmap file directly into a C program and then referencing the variables defined by the file. For example, suppose there exists a file **stip.bitmap**, which was created by the **bitmap** program and contains a stipple pattern. The following code uses **Tk_DefineBitmap** to define a new bitmap named **foo**:

```
Pixmap bitmap;
#include "stip.bitmap"
Tk_DefineBitmap(interp, Tk_GetUid("foo"), stip_bits,
    stip_width, stip_height);
...
bitmap = Tk_GetBitmap(interp, tkwin, Tk_GetUid("foo"));
```

This code causes the bitmap file to be read at compile-time and incorporates the bitmap information into the program's executable image. The same bitmap file could be read at run-time using **Tk_GetBitmap**:

```
Pixmap bitmap;
bitmap = Tk_GetBitmap(interp, tkwin, Tk_GetUid("@stip.bitmap"));
```

The second form is a bit more flexible (the file could be modified after the program has been compiled, or a different string could be provided to read a different file), but it is a little slower and requires the bitmap file to exist separately from the program.

Tk_GetBitmap maintains a database of all the bitmaps that are currently in use. Whenever possible, it will return an existing bitmap rather than creating a new one. This approach can substantially reduce server overhead, so **Tk_GetBitmap** should generally be used in preference to Xlib procedures like **XReadBitmapFile**.

The bitmaps returned by **Tk_GetBitmap** are shared, so callers should never modify them. If a bitmap must be modified dynamically, then it should be created by calling Xlib procedures such as **XReadBitmapFile** or **XCreatePixmap** directly.

The procedure **Tk_NameOfBitmap** is roughly the inverse of **Tk_GetBitmap**. Given an X Pixmap argument, it returns the *id* that was passed to **Tk_GetBitmap** when the bitmap was created. *Bitmap* must have been the return value from a previous call to **Tk_GetBitmap**.

Tk_SizeOfBitmap returns the dimensions of its *bitmap* argument in the words pointed to by the *widthPtr* and *heightPtr* arguments. As with **Tk_NameOfBitmap**, *bitmap* must have been created by **Tk_GetBitmap**.

When a bitmap returned by **Tk_GetBitmap** is no longer needed, **Tk_FreeBitmap** should be called to release it. There should be exactly one call to **Tk_FreeBitmap** for each call to **Tk_GetBitmap**. When a bitmap is no longer in use anywhere (i.e. it has been freed as many times as it has been gotten) **Tk_FreeBitmap** will release it to the X server and delete it from the database.

BUGS

In determining whether an existing bitmap can be used to satisfy a new request, **Tk_GetBitmap** considers only the immediate value of its *id* argument. For example, when a file name is passed to **Tk_GetBitmap**, **Tk_GetBitmap** will assume it is safe to re-use an existing bitmap created from the same file name: it will not check to see whether the file itself has changed, or whether the current directory has changed, thereby causing the name to refer to a different file.

KEYWORDS

bitmap, pixmap

NAME

Tk_GetCapStyle, Tk_NameOfCapStyle − translate between strings and cap styles

SYNOPSIS

#include <tk.h>

int
Tk_GetCapStyle(*interp, string, capPtr***)**

char ∗
Tk_NameOfCapStyle(*cap***)**

ARGUMENTS

Tcl_Interp	∗*interp*	(in)	Interpreter to use for error reporting.
char	∗*string*	(in)	String containing name of cap style: one of ‘‘butt’’, ‘‘projecting’’, or ‘‘round’’.
int	∗*capPtr*	(out)	Pointer to location in which to store X cap style corresponding to *string*.
int	*cap*	(in)	Cap style: one of **CapButt**, **CapProjecting**, or **CapRound**.

DESCRIPTION

Tk_GetCapStyle places in ∗*capPtr* the X cap style corresponding to *string*. This will be one of the values **CapButt**, **CapProjecting**, or **CapRound**. Cap styles are typically used in X graphics contexts to indicate how the end-points of lines should be capped. See the X documentation for information on what each style implies.

Under normal circumstances the return value is **TCL_OK** and *interp* is unused. If *string* doesn’t contain a valid cap style or an abbreviation of one of these names, then an error message is stored in *interp->result*, **TCL_ERROR** is returned, and ∗*capPtr* is unmodified.

Tk_NameOfCapStyle is the logical inverse of **Tk_GetCapStyle**. Given a cap style such as **CapButt** it returns a statically-allocated string corresponding to *cap*. If *cap* isn’t a legal cap style, then ‘‘unknown cap style’’ is returned.

KEYWORDS

butt, cap style, projecting, round

NAME

Tk_GetColormap, Tk_FreeColormap − allocate and free colormaps

SYNOPSIS

#include <tk.h>

Colormap
Tk_GetColormap(*interp, tkwin, string***)**

Tk_FreeColormap(*display, colormap***)**

ARGUMENTS

Tcl_Interp	∗*interp*	(in)	Interpreter to use for error reporting.
Tk_Window	*tkwin*	(in)	Token for window in which colormap will be used.
char	∗*string*	(in)	Selects a colormap: either **new** or the name of a window with the same screen and visual as *tkwin*.
Display	∗*display*	(in)	Display for which *colormap* was allocated.
Colormap	*colormap*	(in)	Colormap to free; must have been returned by a previous call to **Tk_GetColormap** or **Tk_GetVisual**.

DESCRIPTION

These procedures are used to manage colormaps. **Tk_GetColormap** returns a colormap suitable for use in *tkwin*. If its *string* argument is **new** then a new colormap is created; otherwise *string* must be the name of another window with the same screen and visual as *tkwin*, and the colormap from that window is returned. If *string* doesn't make sense, or if it refers to a window on a different screen from *tkwin* or with a different visual than *tkwin*, then **Tk_GetColormap** returns **None** and leaves an error message in *interp->result*.

Tk_FreeColormap should be called when a colormap returned by **Tk_GetColormap** is no longer needed. Tk maintains a reference count for each colormap returned by **Tk_GetColormap**, so there should eventually be one call to **Tk_FreeColormap** for each call to **Tk_GetColormap**. When a colormap's reference count becomes zero, Tk releases the X colormap.

Tk_GetVisual and **Tk_GetColormap** work together, in that a new colormap created by **Tk_GetVisual** may later be returned by **Tk_GetColormap**. The reference counting mechanism for colormaps includes both procedures, so callers of **Tk_GetVisual** must also call **Tk_FreeColormap** to release the colormap. If **Tk_GetColormap** is called with a *string* value of **new** then the resulting colormap will never be returned by **Tk_GetVisual**; however, it can be used in other windows by calling **Tk_GetColormap** with the original window's name as *string*.

KEYWORDS

colormap

NAME

Tk_GetColor, Tk_GetColorByValue, Tk_NameOfColor, Tk_FreeColor − maintain database of colors

SYNOPSIS

#include <tk.h>

XColor ∗
Tk_GetColor(*interp, tkwin, nameId***)**

XColor ∗
Tk_GetColorByValue(*tkwin, prefPtr***)**

char ∗
Tk_NameOfColor(*colorPtr***)**

GC
Tk_GCForColor(*colorPtr, drawable*)

Tk_FreeColor(*colorPtr***)**

ARGUMENTS

Tcl_Interp	∗*interp*	(in)	Interpreter to use for error reporting.
Tk_Window	*tkwin*	(in)	Token for window in which color will be used.
Tk_Uid	*nameId*	(in)	Textual description of desired color.
XColor	∗*prefPtr*	(in)	Indicates red, green, and blue intensities of desired color.
XColor	∗*colorPtr*	(in)	Pointer to X color information. Must have been allocated by previous call to **Tk_GetColor** or **Tk_GetColorByValue**, except when passed to **Tk_NameOfColor**.
Drawable	*drawable*	(in)	Drawable in which the result graphics context will be used. Must have same screen and depth as the window for which the color was allocated.

DESCRIPTION

The **Tk_GetColor** and **Tk_GetColorByValue** procedures locate pixel values that may be used to render particular colors in the window given by *tkwin*. In **Tk_GetColor** the desired color is specified with a Tk_Uid (*nameId*), which may have any of the following forms:

colorname — Any of the valid textual names for a color defined in the server's color database file, such as **red** or **PeachPuff**.

#RGB

#RRGGBB

#RRRGGGBBB

#RRRRGGGGBBBB — A numeric specification of the red, green, and blue intensities to use to display the color. Each *R*, *G*, or *B* represents a single hexadecimal digit. The four forms permit colors to be specified with 4-bit, 8-bit, 12-bit or 16-bit values. When fewer than 16 bits are provided for each color, they represent the most significant bits of the color. For example, #3a7 is the same as #3000a0007000.

In **Tk_GetColorByValue**, the desired color is indicated with the *red*, *green*, and *blue* fields of the structure pointed to by *colorPtr*.

If **Tk_GetColor** or **Tk_GetColorByValue** is successful in allocating the desired color, then it returns a pointer to an XColor structure; the structure indicates the exact intensities of the allocated color (which may differ slightly from those requested, depending on the limitations of the screen) and a pixel value that may be used to draw in the color. If the colormap for *tkwin* is full, **Tk_GetColor** and **Tk_GetColorByValue** will use the closest existing color in the colormap. If **Tk_GetColor** encounters an error while allocating the color (such as an unknown color name) then NULL is returned and an error message is stored in *interp->result*; **Tk_GetColorByValue** never returns an error.

Tk_GetColor and **Tk_GetColorByValue** maintain a database of all the colors currently in use. If the same *nameId* is requested multiple times from **Tk_GetColor** (e.g. by different windows), or if the same intensities are requested multiple times from **Tk_GetColorByValue**, then existing pixel values will be reused. Re-using an existing pixel avoids any interaction with the X server, which makes the allocation much more efficient. For this reason, you should generally use **Tk_GetColor** or **Tk_GetColorByValue** instead of Xlib procedures like **XAllocColor**, **XAllocNamedColor**, or **XParseColor**.

Since different calls to **Tk_GetColor** or **Tk_GetColorByValue** may return the same shared pixel value, callers should never change the color of a pixel returned by the procedures. If you need to change a color value dynamically, you should use **XAllocColorCells** to allocate the pixel value for the color.

The procedure **Tk_NameOfColor** is roughly the inverse of **Tk_GetColor**. If its *colorPtr* argument was created by **Tk_GetColor**, then the return value is the *nameId* string that was passed to **Tk_GetColor** to create the color. If *colorPtr* was created by a call to **Tk_GetColorByValue**, or by any other mechanism, then the return value is a string that could be passed to **Tk_GetColor** to return the same color. Note: the string returned by **Tk_NameOfColor** is only guaranteed to persist until the next call to **Tk_NameOfColor**.

Tk_GCForColor returns a graphics context whose **Foreground** field is the pixel allocated for *colorPtr* and whose other fields all have default values. This provides an easy way to do basic drawing with a color. The graphics context is cached with the color and will exist only as long as *colorPtr* exists; it is freed when the last reference to *colorPtr* is freed by calling **Tk_FreeColor**.

When a pixel value returned by **Tk_GetColor** or **Tk_GetColorByValue** is no longer needed, **Tk_FreeColor** should be called to release the color. There should be exactly one call to **Tk_FreeColor** for each call to **Tk_GetColor** or **Tk_GetColorByValue**. When a pixel value is no longer in use anywhere (i.e. it has been freed as many times as it has been gotten) **Tk_FreeColor** will release it to the X server and delete it from the database.

KEYWORDS

color, intensity, pixel value

NAME

Tk_GetCursor, Tk_GetCursorFromData, Tk_NameOfCursor, Tk_FreeCursor − maintain database of cursors

SYNOPSIS

#include <tk.h>

Tk_Cursor
Tk_GetCursor(*interp, tkwin, nameId***)**

Tk_Cursor
Tk_GetCursorFromData(*interp, tkwin, source, mask, width, height, xHot, yHot, fg, bg***)**

char ∗
Tk_NameOfCursor(*display, cursor***)**

Tk_FreeCursor(*display, cursor***)**

ARGUMENTS

Tcl_Interp	∗*interp*	(in)	Interpreter to use for error reporting.
Tk_Window	*tkwin*	(in)	Token for window in which the cursor will be used.
Tk_Uid	*nameId*	(in)	Description of cursor; see below for possible values.
char	∗*source*	(in)	Data for cursor bitmap, in standard bitmap format.
char	∗*mask*	(in)	Data for mask bitmap, in standard bitmap format.
int	*width*	(in)	Width of *source* and *mask*.
int	*height*	(in)	Height of *source* and *mask*.
int	*xHot*	(in)	X-location of cursor hot-spot.
int	*yHot*	(in)	Y-location of cursor hot-spot.
Tk_Uid	*fg*	(in)	Textual description of foreground color for cursor.
Tk_Uid	*bg*	(in)	Textual description of background color for cursor.
Display	∗*display*	(in)	Display for which *cursor* was allocated.
Tk_Cursor	*cursor*	(in)	Opaque Tk identifier for cursor. If passed to**Tk_FreeCursor**, must have been returned by some previous call to **Tk_GetCursor** or **Tk_GetCursorFromData**.

DESCRIPTION

These procedures manage a collection of cursors being used by an application. The procedures allow cursors to be re-used efficiently, thereby avoiding server overhead, and also allow cursors to be named with character strings (actually Tk_Uids).

Tk_GetCursor takes as argument a Tk_Uid describing a cursor, and returns an opaque Tk identifier for a cursor corresponding to the description. It re-uses an existing cursor if possible and creates a new one otherwise. *NameId* must be a standard Tcl list with one of the following forms:

name [*fgColor* [*bgColor*]]

Name is the name of a cursor in the standard X cursor font, i.e., any of the names defined in **cursorfont.h**, without the **XC_**. Some example values are **X_cursor**, **hand2**, or **left_ptr**. Appendix B of ‘‘The X Window System’’ by Scheifler & Gettys has illustrations showing what each of these

cursors looks like. If *fgColor* and *bgColor* are both specified, they give the foreground and background colors to use for the cursor (any of the forms acceptable to **Tk_GetColor** may be used). If only *fgColor* is specified, then there will be no background color: the background will be transparent. If no colors are specified, then the cursor will use black for its foreground color and white for its background color.

The Macintosh version of Tk also supports all of the X cursors. Tk on the Mac will also accept any of the standard Mac cursors including **ibeam**, **crosshair**, **watch**, **plus**, and **arrow**. In addition, Tk will load Macintosh cursor resources of the types **crsr** (color) and **CURS** (black and white) by the name of the of the resource. The application and all its open dynamic library's resource files will be searched for the named cursor. If there are conflicts color cursors will always be loaded in preference to black and white cursors.

@*sourceName maskName fgColor bgColor*

In this form, *sourceName* and *maskName* are the names of files describing bitmaps for the cursor's source bits and mask. Each file must be in standard X11 or X10 bitmap format. *FgColor* and *bgColor* indicate the colors to use for the cursor, in any of the forms acceptable to **Tk_GetColor**. This form of the command will not work on Macintosh or Windows computers.

@*sourceName fgColor*

This form is similar to the one above, except that the source is used as mask also. This means that the cursor's background is transparent. This form of the command will not work on Macintosh or Windows computers.

Tk_GetCursorFromData allows cursors to be created from in-memory descriptions of their source and mask bitmaps. *Source* points to standard bitmap data for the cursor's source bits, and *mask* points to standard bitmap data describing which pixels of *source* are to be drawn and which are to be considered transparent. *Width* and *height* give the dimensions of the cursor, *xHot* and *yHot* indicate the location of the cursor's hot-spot (the point that is reported when an event occurs), and *fg* and *bg* describe the cursor's foreground and background colors textually (any of the forms suitable for **Tk_GetColor** may be used). Typically, the arguments to **Tk_GetCursorFromData** are created by including a cursor file directly into the source code for a program, as in the following example:

```
Tk_Cursor cursor;
#include "source.cursor"
#include "mask.cursor"
cursor = Tk_GetCursorFromData(interp, tkwin, source_bits,
    mask_bits, source_width, source_height, source_x_hot,
    source_y_hot, Tk_GetUid("red"), Tk_GetUid("blue"));
```

Under normal conditions, **Tk_GetCursor** and **Tk_GetCursorFromData** will return an identifier for the requested cursor. If an error occurs in creating the cursor, such as when *nameId* refers to a non-existent file, then **None** is returned and an error message will be stored in *interp->result*.

Tk_GetCursor and **Tk_GetCursorFromData** maintain a database of all the cursors they have created. Whenever possible, a call to **Tk_GetCursor** or **Tk_GetCursorFromData** will return an existing cursor rather than creating a new one. This approach can substantially reduce server overhead, so the Tk procedures should generally be used in preference to Xlib procedures like **XCreateFontCursor** or **XCreatePixmapCursor**, which create a new cursor on each call.

The procedure **Tk_NameOfCursor** is roughly the inverse of **Tk_GetCursor**. If its *cursor* argument was created by **Tk_GetCursor**, then the return value is the *nameId* argument that was passed to **Tk_GetCursor** to create the cursor. If *cursor* was created by a call to **Tk_GetCursorFromData**, or by any other mechanism, then the return value is a hexadecimal string giving the X identifier for the cursor. Note: the string returned by **Tk_NameOfCursor** is only guaranteed to persist until the next call to **Tk_NameOfCursor**. Also, this call is not portable except for cursors returned by **Tk_GetCursor**.

When a cursor returned by **Tk_GetCursor** or **Tk_GetCursorFromData** is no longer needed, **Tk_FreeCursor** should be called to release it. There should be exactly one call to **Tk_FreeCursor** for each call to **Tk_GetCursor** or **Tk_GetCursorFromData**. When a cursor is no longer in use anywhere (i.e. it has been freed as many times as it has been gotten) **Tk_FreeCursor** will release it to the X server and remove it from the database.

BUGS

In determining whether an existing cursor can be used to satisfy a new request, **Tk_GetCursor** and **Tk_GetCursorFromData** consider only the immediate values of their arguments. For example, when a file name is passed to **Tk_GetCursor**, **Tk_GetCursor** will assume it is safe to re-use an existing cursor created from the same file name: it will not check to see whether the file itself has changed, or whether the current directory has changed, thereby causing the name to refer to a different file. Similarly, **Tk_GetCursorFromData** assumes that if the same *source* pointer is used in two different calls, then the pointers refer to the same data; it does not check to see if the actual data values have changed.

KEYWORDS

cursor

NAME

Tk_GetFont, Tk_NameOfFont, Tk_FreeFont − maintain database of fonts

SYNOPSIS

```
#include <tk.h>

Tk_Font
Tk_GetFont(interp, tkwin, string)

char *
Tk_NameOfFont(tkfont)

void
Tk_FreeFont(tkfont)
```

ARGUMENTS

Tcl_Interp	**interp*	(in)	Interpreter to use for error reporting.
Tk_Window	*tkwin*	(in)	Token for window on the display in which font will be used.
const char	**string*	(in)	Name or description of desired font. See documentation for the **font** command for details on acceptable formats.
Tk_Font	*tkfont*	(in)	Opaque font token.

DESCRIPTION

Tk_GetFont finds the font indicated by *string* and returns a token that represents the font. The return value can be used in subsequent calls to procedures such as **Tk_FontMetrics**, **Tk_MeasureChars**, and **Tk_FreeFont**. The token returned by **Tk_GetFont** will remain valid until **Tk_FreeFont** is called to release it. *String* can be either a symbolic name or a font description; see the documentation for the **font** command for a description of the valid formats. If **Tk_GetFont** is unsuccessful (because, for example, *string* was not a valid font specification) then it returns **NULL** and stores an error message in *interp->result*.

Tk_GetFont maintains a database of all fonts it has allocated. If the same *string* is requested multiple times (e.g. by different windows or for different purposes), then additional calls for the same *string* will be handled without involving the platform-specific graphics server.

The procedure **Tk_NameOfFont** is roughly the inverse of **Tk_GetFont**. Given a *tkfont* that was created by **Tk_GetFont**, the return value is the *string* argument that was passed to **Tk_GetFont** to create the font. The string returned by **Tk_NameOfFont** is only guaranteed to persist until the *tkfont* is deleted. The caller must not modify this string.

When a font returned by **Tk_GetFont** is no longer needed, **Tk_FreeFont** should be called to release it. There should be exactly one call to **Tk_FreeFont** for each call to **Tk_GetFont**. When a font is no longer in use anywhere (i.e. it has been freed as many times as it has been gotten) **Tk_FreeFont** will release any platform-specific storage and delete it from the database.

KEYWORDS

font

NAME

Tk_GetGC, Tk_FreeGC − maintain database of read-only graphics contexts

SYNOPSIS

#include <tk.h>

GC
Tk_GetGC(*tkwin, valueMask, valuePtr*)

Tk_FreeGC(*display, gc*)

ARGUMENTS

Tk_Window	*tkwin*	(in)	Token for window in which the graphics context will be used.
unsigned long	*valueMask*	(in)	Mask of bits (such as **GCForeground** or **GCStipple**) indicating which fields of ∗*valuePtr* are valid.
XGCValues	∗*valuePtr*	(in)	Pointer to structure describing the desired values for the graphics context.
Display	∗*display*	(in)	Display for which *gc* was allocated.
GC	*gc*	(in)	X identifier for graphics context that is no longer needed. Must have been allocated by **Tk_GetGC**.

DESCRIPTION

Tk_GetGC and **Tk_FreeGC** manage a collection of graphics contexts being used by an application. The procedures allow graphics contexts to be shared, thereby avoiding the server overhead that would be incurred if a separate GC were created for each use. **Tk_GetGC** takes arguments describing the desired graphics context and returns an X identifier for a GC that fits the description. The graphics context that is returned will have default values in all of the fields not specified explicitly by *valueMask* and *valuePtr*.

Tk_GetGC maintains a database of all the graphics contexts it has created. Whenever possible, a call to **Tk_GetGC** will return an existing graphics context rather than creating a new one. This approach can substantially reduce server overhead, so **Tk_GetGC** should generally be used in preference to the Xlib procedure **XCreateGC**, which creates a new graphics context on each call.

Since the return values of **Tk_GetGC** are shared, callers should never modify the graphics contexts returned by **Tk_GetGC**. If a graphics context must be modified dynamically, then it should be created by calling **XCreateGC** instead of **Tk_GetGC**.

When a graphics context is no longer needed, **Tk_FreeGC** should be called to release it. There should be exactly one call to **Tk_FreeGC** for each call to **Tk_GetGC**. When a graphics context is no longer in use anywhere (i.e. it has been freed as many times as it has been gotten) **Tk_FreeGC** will release it to the X server and delete it from the database.

KEYWORDS

graphics context

NAME

Tk_GetImage, Tk_RedrawImage, Tk_SizeOfImage, Tk_FreeImage − use an image in a widget

SYNOPSIS

#include <tk.h>

Tk_Image
Tk_GetImage(*interp, tkwin, name, changeProc, clientData*)

Tk_RedrawImage(*image, imageX, imageY, width, height, drawable, drawableX, drawableY*)

Tk_SizeOfImage(*image, widthPtr, heightPtr*)

Tk_FreeImage(*image*)

ARGUMENTS

Tcl_Interp	∗*interp*	(in)	Place to leave error message.
Tk_Window	*tkwin*	(in)	Window in which image will be used.
char	∗*name*	(in)	Name of image.
Tk_ImageChangedProc	∗*changeProc*	(in)	Procedure for Tk to invoke whenever image content or size changes.
ClientData	*clientData*	(in)	One-word value for Tk to pass to *changeProc*.
Tk_Image	*image*	(in)	Token for image instance; must have been returned by a previous call to **Tk_GetImage**.
int	*imageX*	(in)	X-coordinate of upper-left corner of region of image to redisplay (measured in pixels from the image's upper-left corner).
int	*imageY*	(in)	Y-coordinate of upper-left corner of region of image to redisplay (measured in pixels from the image's upper-left corner).
int	*width*	((in))	Width of region of image to redisplay.
int	*height*	((in))	Height of region of image to redisplay.
Drawable	*drawable*	(in)	Where to display image. Must either be window specified to **Tk_GetImage** or a pixmap compatible with that window.
int	*drawableX*	(in)	Where to display image in *drawable*: this is the x-coordinate in *drawable* where x-coordinate *imageX* of the image should be displayed.
int	*drawableY*	(in)	Where to display image in *drawable*: this is the y-coordinate in *drawable* where y-coordinate *imageY* of the image should be displayed.
int	*widthPtr*	(out)	Store width of *image* (in pixels) here.
int	*heightPtr*	(out)	Store height of *image* (in pixels) here.

DESCRIPTION

These procedures are invoked by widgets that wish to display images. **Tk_GetImage** is invoked by a widget when it first decides to display an image. *name* gives the name of the desired image and *tkwin* identifies the window where the image will be displayed. **Tk_GetImage** looks up the image in the table of existing images and returns a token for a new instance of the image. If the image doesn't exist then **Tk_GetImage** returns NULL and leaves an error message in *interp->result*.

When a widget wishes to actually display an image it must call **Tk_RedrawWidget**, identifying the image (*image*), a region within the image to redisplay (*imageX*, *imageY*, *width*, and *height*), and a place to display the image (*drawable*, *drawableX*, and *drawableY*). Tk will then invoke the appropriate image manager, which will display the requested portion of the image before returning.

A widget can find out the dimensions of an image by calling **Tk_SizeOfImage**: the width and height will be stored in the locations given by *widthPtr* and *heightPtr*, respectively.

When a widget is finished with an image (e.g., the widget is being deleted or it is going to use a different image instead of the current one), it must call **Tk_FreeImage** to release the image instance. The widget should never again use the image token after passing it to **Tk_FreeImage**. There must be exactly one call to **Tk_FreeImage** for each call to **Tk_GetImage**.

If the contents or size of an image changes, then any widgets using the image will need to find out about the changes so that they can redisplay themselves. The *changeProc* and *clientData* arguments to **Tk_GetImage** are used for this purpose. *changeProc* will be called by Tk whenever a change occurs in the image; it must match the following prototype:

```
typedef void Tk_ImageChangedProc(
    ClientData clientData,
    int x,
    int y,
    int width,
    int height,
    int imageWidth,
    int imageHeight);
```

The *clientData* argument to *changeProc* is the same as the *clientData* argument to **Tk_GetImage**. It is usually a pointer to the widget record for the widget or some other data structure managed by the widget. The arguments *x*, *y*, *width*, and *height* identify a region within the image that must be redisplayed; they are specified in pixels measured from the upper-left corner of the image. The arguments *imageWidth* and *imageHeight* give the image's (new) size.

SEE ALSO

Tk_CreateImageType

KEYWORDS

images, redisplay

NAME

Tk_GetJoinStyle, Tk_NameOfJoinStyle − translate between strings and join styles

SYNOPSIS

#include <tk.h>

int
Tk_GetJoinStyle(*interp, string, joinPtr***)**

char ∗
Tk_NameOfJoinStyle(*join***)**

ARGUMENTS

Tcl_Interp	∗*interp*	(in)	Interpreter to use for error reporting.
char	∗*string*	(in)	String containing name of join style: one of ‘‘bevel’’, ‘‘miter’’, or ‘‘round’’.
int	∗*joinPtr*	(out)	Pointer to location in which to store X join style corresponding to *string*.
int	*join*	(in)	Join style: one of **JoinBevel**, **JoinMiter**, **JoinRound**.

DESCRIPTION

Tk_GetJoinStyle places in ∗*joinPtr* the X join style corresponding to *string*, which will be one of **JoinBevel**, **JoinMiter**, or **JoinRound**. Join styles are typically used in X graphics contexts to indicate how adjacent line segments should be joined together. See the X documentation for information on what each style implies.

Under normal circumstances the return value is **TCL_OK** and *interp* is unused. If *string* doesn’t contain a valid join style or an abbreviation of one of these names, then an error message is stored in *interp->result*, **TCL_ERROR** is returned, and ∗*joinPtr* is unmodified.

Tk_NameOfJoinStyle is the logical inverse of **Tk_GetJoinStyle**. Given a join style such as **JoinBevel** it returns a statically-allocated string corresponding to *join*. If *join* isn’t a legal join style, then ‘‘unknown join style’’ is returned.

KEYWORDS

bevel, join style, miter, round

NAME

Tk_GetJustify, Tk_NameOfJustify − translate between strings and justification styles

SYNOPSIS

#include <tk.h>

Tk_Justify
Tk_GetJustify(*interp, string, justifyPtr***)**

char ∗
Tk_NameOfJustify(*justify***)**

ARGUMENTS

Tcl_Interp	∗*interp*	(in)	Interpreter to use for error reporting.
char	∗*string*	(in)	String containing name of justification style (‘‘left’’, ‘‘right’’, or ‘‘center’’).
int	∗*justifyPtr*	(out)	Pointer to location in which to store justify value corresponding to *string*.
Tk_Justify	*justify*	(in)	Justification style (one of the values listed below).

DESCRIPTION

Tk_GetJustify places in ∗*justifyPtr* the justify value corresponding to *string*. This value will be one of the following:

TK_JUSTIFY_LEFT
Means that the text on each line should start at the left edge of the line; as a result, the right edges of lines may be ragged.

TK_JUSTIFY_RIGHT
Means that the text on each line should end at the right edge of the line; as a result, the left edges of lines may be ragged.

TK_JUSTIFY_CENTER
Means that the text on each line should be centered; as a result, both the left and right edges of lines may be ragged.

Under normal circumstances the return value is **TCL_OK** and *interp* is unused. If *string* doesn’t contain a valid justification style or an abbreviation of one of these names, then an error message is stored in *interp->result*, **TCL_ERROR** is returned, and ∗*justifyPtr* is unmodified.

Tk_NameOfJustify is the logical inverse of **Tk_GetJustify**. Given a justify value it returns a statically-allocated string corresponding to *justify*. If *justify* isn’t a legal justify value, then ‘‘unknown justification style’’ is returned.

KEYWORDS

center, fill, justification, string

NAME

Tk_GetOption − retrieve an option from the option database

SYNOPSIS

#include <tk.h>

Tk_Uid
Tk_GetOption(*tkwin, name, class*)

ARGUMENTS

Tk_Window	*tkwin*	(in)	Token for window.
char	∗*name*	(in)	Name of desired option.
char	∗*class*	(in)	Class of desired option. Null means there is no class for this option; do lookup based on name only.

DESCRIPTION

This procedure is invoked to retrieve an option from the database associated with *tkwin*'s main window. If there is an option for *tkwin* that matches the given *name* or *class*, then it is returned in the form of a Tk_Uid. If multiple options match *name* and *class*, then the highest-priority one is returned. If no option matches, then NULL is returned.

Tk_GetOption caches options related to *tkwin* so that successive calls for the same *tkwin* will execute much more quickly than successive calls for different windows.

KEYWORDS

class, name, option, retrieve

NAME

Tk_GetPixels, Tk_GetScreenMM − translate between strings and screen units

SYNOPSIS

#include <tk.h>

int
Tk_GetPixels(*interp, tkwin, string, intPtr***)**

int
Tk_GetScreenMM(*interp, tkwin, string, doublePtr***)**

ARGUMENTS

Tcl_Interp	∗*interp*	(in)	Interpreter to use for error reporting.
Tk_Window	*tkwin*	(in)	Window whose screen geometry determines the conversion between absolute units and pixels.
char	∗*string*	(in)	String that specifies a distance on the screen.
int	∗*intPtr*	(out)	Pointer to location in which to store converted distance in pixels.
double	∗*doublePtr*	(out)	Pointer to location in which to store converted distance in millimeters.

DESCRIPTION

These two procedures take as argument a specification of distance on the screen (*string*) and compute the corresponding distance either in integer pixels or floating-point millimeters. In either case, *string* specifies a screen distance as a floating-point number followed by one of the following characters that indicates units:

<none> The number specifies a distance in pixels.

c The number specifies a distance in centimeters on the screen.

i The number specifies a distance in inches on the screen.

m The number specifies a distance in millimeters on the screen.

p The number specifies a distance in printer's points (1/72 inch) on the screen.

Tk_GetPixels converts *string* to the nearest even number of pixels and stores that value at ∗*intPtr*. **Tk_GetScreenMM** converts *string* to millimeters and stores the double-precision floating-point result at ∗*doublePtr*.

Both procedures return **TCL_OK** under normal circumstances. If an error occurs (e.g. *string* contains a number followed by a character that isn't one of the ones above) then **TCL_ERROR** is returned and an error message is left in *interp->result*.

KEYWORDS

centimeters, convert, inches, millimeters, pixels, points, screen units

NAME

Tk_GetPixmap, Tk_FreePixmap − allocate and free pixmaps

SYNOPSIS

#include <tk.h>

Pixmap
Tk_GetPixmap(*display, d, width, height, depth***)**

Tk_FreePixmap(*display, pixmap***)**

ARGUMENTS

Display	*∗display*	(in)	X display for the pixmap.
Drawable	*d*	(in)	Pixmap or window where the new pixmap will be used for drawing.
int	*width*	(in)	Width of pixmap.
int	*height*	(in)	Height of pixmap.
int	*depth*	(in)	Number of bits per pixel in pixmap.
Pixmap	*pixmap*	(in)	Pixmap to destroy.

DESCRIPTION

These procedures are identical to the Xlib procedures **XCreatePixmap** and **XFreePixmap**, except that they have extra code to manage X resource identifiers so that identifiers for deleted pixmaps can be reused in the future. It is important for Tk applications to use these procedures rather than **XCreatePixmap** and **XFreePixmap**; otherwise long-running applications may run out of resource identifiers.

Tk_GetPixmap creates a pixmap suitable for drawing in *d*, with dimensions given by *width*, *height*, and *depth*, and returns its identifier. **Tk_FreePixmap** destroys the pixmap given by *pixmap* and makes its resource identifier available for reuse.

KEYWORDS

pixmap, resource identifier

NAME

Tk_GetRelief, Tk_NameOfRelief − translate between strings and relief values

SYNOPSIS

#include <tk.h>

int
Tk_GetRelief(*interp, name, reliefPtr***)**

char ∗
Tk_NameOfRelief(*relief***)**

ARGUMENTS

Tcl_Interp	∗*interp*	(in)	Interpreter to use for error reporting.
char	∗*name*	(in)	String containing relief name (one of ‘‘flat’’, ‘‘groove’’, ‘‘raised’’, ‘‘ridge’’, ‘‘solid’’, or ‘‘sunken’’).
int	∗*reliefPtr*	(out)	Pointer to location in which to store relief value corresponding to *name*.
int	*relief*	(in)	Relief value (one of TK_RELIEF_FLAT, TK_RELIEF_RAISED, TK_RELIEF_SUNKEN, TK_RELIEF_GROOVE, TK_RELIEF_SOLID, or TK_RELIEF_RIDGE).

DESCRIPTION

Tk_GetRelief places in ∗*reliefPtr* the relief value corresponding to *name*. This value will be one of TK_RELIEF_FLAT, TK_RELIEF_RAISED, TK_RELIEF_SUNKEN, TK_RELIEF_GROOVE, TK_RELIEF_SOLID, or TK_RELIEF_RIDGE. Under normal circumstances the return value is TCL_OK and *interp* is unused. If *name* doesn’t contain one of the valid relief names or an abbreviation of one of them, then an error message is stored in *interp->result*, TCL_ERROR is returned, and ∗*reliefPtr* is unmodified.

Tk_NameOfRelief is the logical inverse of **Tk_GetRelief**. Given a relief value it returns the corresponding string (‘‘flat’’, ‘‘raised’’, ‘‘sunken’’, ‘‘groove’’, ‘‘solid’’, or ‘‘ridge’’). If *relief* isn’t a legal relief value, then ‘‘unknown relief’’ is returned.

KEYWORDS

name, relief, string

NAME

Tk_GetRootCoords − Compute root-window coordinates of window

SYNOPSIS

#include <tk.h>

Tk_GetRootCoords(*tkwin, xPtr, yPtr*)

ARGUMENTS

Tk_Window	*tkwin*	(in)	Token for window.
int	∗*xPtr*	(out)	Pointer to location in which to store root-window x-coordinate corresponding to left edge of *tkwin*’s border.
int	∗*yPtr*	(out)	Pointer to location in which to store root-window y-coordinate corresponding to top edge of *tkwin*’s border.

DESCRIPTION

This procedure scans through the structural information maintained by Tk to compute the root-window coordinates corresponding to the upper-left corner of *tkwin*’s border. If *tkwin* has no border, then **Tk_GetRootCoords** returns the root-window coordinates corresponding to location (0,0) in *tkwin*. **Tk_GetRootCoords** is relatively efficient, since it doesn’t have to communicate with the X server.

KEYWORDS

coordinates, root window

NAME

Tk_GetScrollInfo − parse arguments for scrolling commands

SYNOPSIS

#include <tk.h>

int
Tk_GetScrollInfo(*interp, argc, argv, dblPtr, intPtr***)**

ARGUMENTS

Tcl_Interp	∗*interp*	(in)	Interpreter to use for error reporting.
int	*argc*	(in)	Number of strings in *argv* array.
char	∗*argv[]*	(in)	Argument strings. These represent the entire widget command, of which the first word is typically the widget name and the second word is typically **xview** or **yview**. This procedure parses arguments starting with *argv*[2].
double	∗*dblPtr*	(out)	Filled in with fraction from **moveto** option, if any.
int	∗*intPtr*	(out)	Filled in with line or page count from **scroll** option, if any. The value may be negative.

DESCRIPTION

Tk_GetScrollInfo parses the arguments expected by widget scrolling commands such as **xview** and **yview**. It receives the entire list of words that make up a widget command and parses the words starting with *argv*[2]. The words starting with *argv*[2] must have one of the following forms:

moveto *fraction*
scroll *number* **units**
scroll *number* **pages**

Any of the **moveto**, **scroll**, **units**, and **pages** keywords may be abbreviated. If *argv* has the **moveto** form, **TK_SCROLL_MOVETO** is returned as result and ∗*dblPtr* is filled in with the *fraction* argument to the command, which must be a proper real value. If *argv* has the **scroll** form, **TK_SCROLL_UNITS** or **TK_SCROLL_PAGES** is returned and ∗*intPtr* is filled in with the *number* value, which must be a proper integer. If an error occurs in parsing the arguments, **TK_SCROLL_ERROR** is returned and an error message is left in *interp->result*.

KEYWORDS

parse, scrollbar, scrolling command, xview, yview

NAME

Tk_GetSelection − retrieve the contents of a selection

SYNOPSIS

#include <tk.h>

int
Tk_GetSelection(*interp, tkwin, selection, target, proc, clientData*)

ARGUMENTS

Tcl_Interp	∗*interp*	(in)	Interpreter to use for reporting errors.
Tk_Window	*tkwin*	(in)	Window on whose behalf to retrieve the selection (determines display from which to retrieve).
Atom	*selection*	(in)	The name of the selection to be retrieved.
Atom	*target*	(in)	Form in which to retrieve selection.
Tk_GetSelProc	∗*proc*	(in)	Procedure to invoke to process pieces of the selection as they are retrieved.
ClientData	*clientData*	(in)	Arbitrary one-word value to pass to *proc*.

DESCRIPTION

Tk_GetSelection retrieves the selection specified by the atom *selection* in the format specified by *target*. The selection may actually be retrieved in several pieces; as each piece is retrieved, *proc* is called to process the piece. *Proc* should have arguments and result that match the type **Tk_GetSelProc**:

```
typedef int Tk_GetSelProc(
    ClientData clientData,
    Tcl_Interp *interp,
    char *portion);
```

The *clientData* and *interp* parameters to *proc* will be copies of the corresponding arguments to **Tk_GetSelection**. *Portion* will be a pointer to a string containing part or all of the selection. For large selections, *proc* will be called several times with successive portions of the selection. The X Inter-Client Communication Conventions Manual allows a selection to be returned in formats other than strings, e.g. as an array of atoms or integers. If this happens, Tk converts the selection back into a string before calling *proc*. If a selection is returned as an array of atoms, Tk converts it to a string containing the atom names separated by white space. For any other format besides string, Tk converts a selection to a string containing hexadecimal values separated by white space.

Tk_GetSelection returns to its caller when the selection has been completely retrieved and processed by *proc*, or when a fatal error has occurred (e.g. the selection owner didn't respond promptly). **Tk_GetSelection** normally returns TCL_OK; if an error occurs, it returns TCL_ERROR and leaves an error message in *interp->result*. *Proc* should also return either TCL_OK or TCL_ERROR. If *proc* encounters an error in dealing with the selection, it should leave an error message in *interp->result* and return TCL_ERROR; this will abort the selection retrieval.

KEYWORDS

format, get, selection retrieval

NAME

Tk_GetUid, Tk_Uid − convert from string to unique identifier

SYNOPSIS

#include <tk.h>

#typedef char *Tk_Uid

Tk_Uid
Tk_GetUid(*string*)

ARGUMENTS

char	**string*	(in)	String for which the corresponding unique identifier is desired.

DESCRIPTION

Tk_GetUid returns the unique identifier corresponding to *string*. Unique identifiers are similar to atoms in Lisp, and are used in Tk to speed up comparisons and searches. A unique identifier (type Tk_Uid) is a string pointer and may be used anywhere that a variable of type ''char *'' could be used. However, there is guaranteed to be exactly one unique identifier for any given string value. If **Tk_GetUid** is called twice, once with string *a* and once with string *b*, and if *a* and *b* have the same string value (strcmp(a, b) == 0), then **Tk_GetUid** will return exactly the same Tk_Uid value for each call (Tk_GetUid(a) == Tk_GetUid(b)). This means that variables of type Tk_Uid may be compared directly (x == y) without having to call **strcmp**. In addition, the return value from **Tk_GetUid** will have the same string value as its argument (strcmp(Tk_GetUid(a), a) == 0).

KEYWORDS

atom, unique identifier

NAME

Tk_GetVRootGeometry − Get location and size of virtual root for window

SYNOPSIS

#include <tk.h>

Tk_GetVRootGeometry(*tkwin, xPtr, yPtr, widthPtr, heightPtr***)**

ARGUMENTS

Tk_Window	*tkwin*	(in)	Token for window whose virtual root is to be queried.
int	*xPtr*	(out)	Points to word in which to store x-offset of virtual root.
int	*yPtr*	(out)	Points to word in which to store y-offset of virtual root.
int	*widthPtr*	(out)	Points to word in which to store width of virtual root.
int	*heightPtr*	(out)	Points to word in which to store height of virtual root.

DESCRIPTION

TkGetVRootGeometry returns geometry information about the virtual root window associated with *tkwin*. The ‘‘associated’’ virtual root is the one in which *tkwin*’s nearest top-level ancestor (or *tkwin* itself if it is a top-level window) has been reparented by the window manager. This window is identified by a **__SWM_ROOT** or **__WM_ROOT** property placed on the top-level window by the window manager. If *tkwin* is not associated with a virtual root (e.g. because the window manager doesn’t use virtual roots) then ∗*xPtr* and ∗*yPtr* will be set to 0 and ∗*widthPtr* and ∗*heightPtr* will be set to the dimensions of the screen containing *tkwin*.

KEYWORDS

geometry, height, location, virtual root, width, window manager

NAME

Tk_GetVisual − translate from string to visual

SYNOPSIS

#include <tk.h>

Visual ∗
Tk_GetVisual(*interp, tkwin, string, depthPtr, colormapPtr***)**

ARGUMENTS

Tcl_Interp	∗*interp*	(in)	Interpreter to use for error reporting.
Tk_Window	*tkwin*	(in)	Token for window in which the visual will be used.
char	∗*string*	(in)	String that identifies the desired visual. See below for valid formats.
int	∗*depthPtr*	(out)	Depth of returned visual gets stored here.
Colormap	∗*colormapPtr*	(out)	If non-NULL then a suitable colormap for visual is found and its identifier is stored here.

DESCRIPTION

Tk_GetVisual takes a string description of a visual and finds a suitable X Visual for use in *tkwin*, if there is one. It returns a pointer to the X Visual structure for the visual and stores the number of bits per pixel for it at ∗*depthPtr*. If *string* is unrecognizable or if no suitable visual could be found, then NULL is returned and **Tk_GetVisual** leaves an error message in *interp->result*. If *colormap* is non-NULL then **Tk_GetVisual** also locates an appropriate colormap for use with the result visual and stores its X identifier at ∗*colormapPtr*.

The *string* argument specifies the desired visual in one of the following ways:

class depth — The string consists of a class name followed by an integer depth, with any amount of white space (including none) in between. *class* selects what sort of visual is desired and must be one of **directcolor**, **grayscale**, **greyscale**, **pseudocolor**, **staticcolor**, **staticgray**, **staticgrey**, or **truecolor**, or a unique abbreviation. *depth* specifies how many bits per pixel are needed for the visual. If possible, **Tk_GetVisual** will return a visual with this depth; if there is no visual of the desired depth then **Tk_GetVisual** looks first for a visual with greater depth, then one with less depth.

default — Use the default visual for *tkwin*'s screen.

pathName — Use the visual for the window given by *pathName*. *pathName* must be the name of a window on the same screen as *tkwin*.

number — Use the visual whose X identifier is *number*.

best ?*depth*? — Choose the ''best possible'' visual, using the following rules, in decreasing order of priority: (a) a visual that has exactly the desired depth is best, followed by a visual with greater depth than requested (but as little extra as possible), followed by a visual with less depth than requested (but as great a depth as possible); (b) if no *depth* is specified, then the deepest available visual is chosen; (c) **pseudocolor** is better than **truecolor** or **directcolor**, which are better than **staticcolor**, which is better than **staticgray** or **grayscale**; (d) the default visual for the screen is better than any other visual.

CREDITS

The idea for **Tk_GetVisual**, and the first implementation, came from Paul Mackerras.

KEYWORDS

colormap, screen, visual

NAME

Tk_HandleEvent − invoke event handlers for window system events

SYNOPSIS

#include <tk.h>

Tk_HandleEvent(*eventPtr*)

ARGUMENTS

XEvent	∗*eventPtr*	(in)	Pointer to X event to dispatch to relevant handler(s).

DESCRIPTION

Tk_HandleEvent is a lower-level procedure that deals with window events. It is called by **Tk_ServiceEvent** (and indirectly by **Tk_DoOneEvent**), and in a few other cases within Tk. It makes callbacks to any window event handlers (created by calls to **Tk_CreateEventHandler**) that match *eventPtr* and then returns. In some cases it may be useful for an application to bypass the Tk event queue and call **Tk_HandleEvent** directly instead of calling **Tk_QueueEvent** followed by **Tk_ServiceEvent**.

This procedure may be invoked recursively. For example, it is possible to invoke **Tk_HandleEvent** recursively from a handler called by **Tk_HandleEvent**. This sort of operation is useful in some modal situations, such as when a notifier has been popped up and an application wishes to wait for the user to click a button in the notifier before doing anything else.

KEYWORDS

callback, event, handler, window

NAME

Tk_IdToWindow − Find Tk's window information for an X window

SYNOPSIS

#include <tk.h>

Tk_Window
Tk_IdToWindow(*display, window*)

ARGUMENTS

Display	**display*	(in)	X display containing the window.
Window	*window*	(in)	X id for window.

DESCRIPTION

Given an X window identifier and the X display it corresponds to, this procedure returns the corresponding Tk_Window handle. If there is no Tk_Window corresponding to *window* then NULL is returned.

KEYWORDS

X window id

NAME

Tk_ImageChanged − notify widgets that image needs to be redrawn

SYNOPSIS

#include <tk.h>

Tk_ImageChanged(*imageMaster, x, y, width, height, imageWidth, imageHeight*)

ARGUMENTS

Tk_ImageMaster	*imageMaster*	(in)	Token for image, which was passed to image's *createProc* when the image was created.
int	*x*	(in)	X-coordinate of upper-left corner of region that needs redisplay (measured from upper-left corner of image).
int	*y*	(in)	Y-coordinate of upper-left corner of region that needs redisplay (measured from upper-left corner of image).
int	*width*	(in)	Width of region that needs to be redrawn, in pixels.
int	*height*	(in)	Height of region that needs to be redrawn, in pixels.
int	*imageWidth*	(in)	Current width of image, in pixels.
int	*imageHeight*	(in)	Current height of image, in pixels.

DESCRIPTION

An image manager calls **Tk_ImageChanged** for an image whenever anything happens that requires the image to be redrawn. As a result of calling **Tk_ImageChanged**, any widgets using the image are notified so that they can redisplay themselves appropriately. The *imageMaster* argument identifies the image, and *x*, *y*, *width*, and *height* specify a rectangular region within the image that needs to be redrawn. *imageWidth* and *imageHeight* specify the image's (new) size.

An image manager should call **Tk_ImageChanged** during its *createProc* to specify the image's initial size and to force redisplay if there are existing instances for the image. If any of the pixel values in the image should change later on, **Tk_ImageChanged** should be called again with *x*, *y*, *width*, and *height* values that cover all the pixels that changed. If the size of the image should change, then **Tk_ImageChanged** must be called to indicate the new size, even if no pixels need to be redisplayed.

SEE ALSO

Tk_CreateImageType

KEYWORDS

images, redisplay, image size changes

NAME

Tk_InternAtom, Tk_GetAtomName − manage cache of X atoms

SYNOPSIS

#include <tk.h>

Atom
Tk_InternAtom(*tkwin, name*)

char ∗
Tk_GetAtomName(*tkwin, atom*)

ARGUMENTS

Tk_Window	*tkwin*	(in)	Token for window. Used to map atom or name relative to a particular display.
char	∗*name*	(in)	String name for which atom is desired.
Atom	*atom*	(in)	Atom for which corresponding string name is desired.

DESCRIPTION

These procedures are similar to the Xlib procedures **XInternAtom** and **XGetAtomName**. **Tk_InternAtom** returns the atom identifier associated with string given by *name*; the atom identifier is only valid for the display associated with *tkwin*. **Tk_GetAtomName** returns the string associated with *atom* on *tkwin*'s display. The string returned by **Tk_GetAtomName** is in Tk's storage: the caller need not free this space when finished with the string, and the caller should not modify the contents of the returned string. If there is no atom *atom* on *tkwin*'s display, then **Tk_GetAtomName** returns the string ''?bad atom?''.

Tk caches the information returned by **Tk_InternAtom** and **Tk_GetAtomName** so that future calls for the same information can be serviced from the cache without contacting the server. Thus **Tk_InternAtom** and **Tk_GetAtomName** are generally much faster than their Xlib counterparts, and they should be used in place of the Xlib procedures.

KEYWORDS

atom, cache, display

NAME

Tk_MainLoop − loop for events until all windows are deleted

SYNOPSIS

#include <tk.h>

Tk_MainLoop()

DESCRIPTION

Tk_MainLoop is a procedure that loops repeatedly calling **Tcl_DoOneEvent**. It returns only when there are no applications left in this process (i.e. no main windows exist anymore). Most windowing applications will call **Tk_MainLoop** after initialization; the main execution of the application will consist entirely of callbacks invoked via **Tcl_DoOneEvent**.

KEYWORDS

application, event, main loop

NAME

Tk_MainWindow − find the main window for an application

SYNOPSIS

#include <tk.h>

Tk_Window
Tk_MainWindow(*interp*)

ARGUMENTS

Tcl_Interp	∗*interp*	(in/out)	Interpreter associated with the application.

DESCRIPTION

If *interp* is associated with a Tk application then **Tk_MainWindow** returns the application's main window. If there is no Tk application associated with *interp* then **Tk_MainWindow** returns NULL and leaves an error message in *interp->result*.

KEYWORDS

application, main window

NAME

Tk_MaintainGeometry, Tk_UnmaintainGeometry − maintain geometry of one window relative to another

SYNOPSIS

#include <tk.h>

Tk_MaintainGeometry(*slave, master, x, y, width, height*)

Tk_UnmaintainGeometry(*slave, master*)

ARGUMENTS

Tk_Window	*slave*	(in)	Window whose geometry is to be controlled.
Tk_Window	*master*	(in)	Window relative to which *slave*'s geometry will be controlled.
int	*x*	(in)	Desired x-coordinate of *slave* in *master*, measured in pixels from the inside of *master*'s left border to the outside of *slave*'s left border.
int	*y*	(in)	Desired y-coordinate of *slave* in *master*, measured in pixels from the inside of *master*'s top border to the outside of *slave*'s top border.
int	*width*	(in)	Desired width for *slave*, in pixels.
int	*height*	(in)	Desired height for *slave*, in pixels.

DESCRIPTION

Tk_MaintainGeometry and **Tk_UnmaintainGeometry** make it easier for geometry managers to deal with slaves whose masters are not their parents. Three problems arise if the master for a slave is not its parent:

[1] The x- and y-position of the slave must be translated from the coordinate system of the master to that of the parent before positioning the slave.

[2] If the master window, or any of its ancestors up to the slave's parent, is moved, then the slave must be repositioned within its parent in order to maintain the correct position relative to the master.

[3] If the master or one of its ancestors is mapped or unmapped, then the slave must be mapped or unmapped to correspond.

None of these problems is an issue if the parent and master are the same. For example, if the master or one of its ancestors is unmapped, the slave is automatically removed by the screen by X.

Tk_MaintainGeometry deals with these problems for slaves whose masters aren't their parents. **Tk_MaintainGeometry** is typically called by a window manager once it has decided where a slave should be positioned relative to its master. **Tk_MaintainGeometry** translates the coordinates to the coordinate system of *slave*'s parent and then moves and resizes the slave appropriately. Furthermore, it remembers the desired position and creates event handlers to monitor the master and all of its ancestors up to (but not including) the slave's parent. If any of these windows is moved, mapped, or unmapped, the slave will be adjusted so that it is mapped only when the master is mapped and its geometry relative to the master remains as specified by *x*, *y*, *width*, and *height*.

When a window manager relinquishes control over a window, or if it decides that it does not want the window to appear on the screen under any conditions, it calls **Tk_UnmaintainGeometry**. **Tk_UnmaintainGeometry** unmaps the window and cancels any previous calls to **Tk_MaintainGeometry** for the *master−slave* pair, so that the slave's geometry and mapped state are no longer maintained automatically. **Tk_UnmaintainGeometry** need not be called by a geometry manager if the slave, the master, or any of the master's ancestors is destroyed: Tk will call it automatically.

If **Tk_MaintainGeometry** is called repeatedly for the same *master−slave* pair, the information from the most recent call supersedes any older information. If **Tk_UnmaintainGeometry** is called for a *master−slave* pair that is isn't currently managed, the call has no effect.

KEYWORDS

geometry manager, map, master, parent, position, slave, unmap

NAME

Tk_ManageGeometry − arrange to handle geometry requests for a window

SYNOPSIS

#include <tk.h>

Tk_ManageGeometry(*tkwin, mgrPtr, clientData*)

ARGUMENTS

Tk_Window	*tkwin*	(in)	Token for window to be managed.
Tk_GeomMgr	∗*mgrPtr*	(in)	Pointer to data structure containing information about the geometry manager, or NULL to indicate that *tkwin*'s geometry shouldn't be managed anymore. The data structure pointed to by *mgrPtr* must be static: Tk keeps a reference to it as long as the window is managed.
ClientData	*clientData*	(in)	Arbitrary one-word value to pass to geometry manager callbacks.

DESCRIPTION

Tk_ManageGeometry arranges for a particular geometry manager, described by the *mgrPtr* argument, to control the geometry of a particular slave window, given by *tkwin*. If *tkwin* was previously managed by some other geometry manager, the previous manager loses control in favor of the new one. If *mgrPtr* is NULL, geometry management is cancelled for *tkwin*.

The structure pointed to by *mgrPtr* contains information about the geometry manager:

```
typedef struct {
    char ∗name;
    Tk_GeomRequestProc ∗requestProc;
    Tk_GeomLostSlaveProc ∗lostSlaveProc;
} Tk_GeomMgr;
```

The *name* field is the textual name for the geometry manager, such as **pack** or **place**; this value will be returned by the command **winfo manager**.

requestProc is a procedure in the geometry manager that will be invoked whenever **Tk_GeometryRequest** is called by the slave to change its desired geometry. *requestProc* should have arguments and results that match the type **Tk_GeomRequestProc**:

```
typedef void Tk_GeomRequestProc(
    ClientData clientData,
    Tk_Window tkwin);
```

The parameters to *requestProc* will be identical to the corresponding parameters passed to **Tk_ManageGeometry**. *clientData* usually points to a data structure containing application-specific information about how to manage *tkwin*'s geometry.

The *lostSlaveProc* field of *mgrPtr* points to another procedure in the geometry manager. Tk will invoke *lostSlaveProc* if some other manager calls **Tk_ManageGeometry** to claim *tkwin* away from the current geometry manager. *lostSlaveProc* is not invoked if **Tk_ManageGeometry** is called with a NULL value for *mgrPtr* (presumably the current geometry manager has made this call, so it already knows that the window is no longer managed), nor is it called if *mgrPtr* is the same as the window's current geometry manager. *lostSlaveProc* should have arguments and results that match the following prototype:

```
typedef void Tk_GeomLostSlaveProc(
    ClientData clientData,
    Tk_Window tkwin);
```

The parameters to *lostSlaveProc* will be identical to the corresponding parameters passed to **Tk_ManageGeometry**.

KEYWORDS

callback, geometry, managed, request, unmanaged

NAME

Tk_MapWindow, Tk_UnmapWindow − map or unmap a window

SYNOPSIS

#include <tk.h>

Tk_Window
Tk_MapWindow(*tkwin*)

Tk_UnmapWindow(*tkwin*)

ARGUMENTS

Tk_Window	*tkwin*	(in)	Token for window.

DESCRIPTION

These procedures may be used to map and unmap windows managed by Tk. **Tk_MapWindow** maps the window given by *tkwin*, and also creates an X window corresponding to *tkwin* if it doesn't already exist. See the **Tk_CreateWindow** manual entry for information on deferred window creation. **Tk_UnmapWindow** unmaps *tkwin*'s window from the screen.

If *tkwin* is a child window (i.e. **Tk_CreateChildWindow** was used to create it), then event handlers interested in map and unmap events are invoked immediately. If *tkwin* isn't an internal window, then the event handlers will be invoked later, after X has seen the request and returned an event for it.

These procedures should be used in place of the X procedures **XMapWindow** and **XUnmapWindow**, since they update Tk's local data structure for *tkwin*. Applications using Tk should not invoke **XMapWindow** and **XUnmapWindow** directly.

KEYWORDS

map, unmap, window

NAME

Tk_MeasureChars, Tk_TextWidth, Tk_DrawChars, Tk_UnderlineChars − routines to measure and display simple single-line strings.

SYNOPSIS

#include <tk.h>

int
Tk_MeasureChars(*tkfont, string, maxChars, maxPixels, flags, lengthPtr***)**

int
Tk_TextWidth(*tkfont, string, numChars***)**

void
Tk_DrawChars(*display, drawable, gc, tkfont, string, numChars, x, y***)**

void
Tk_UnderlineChars(*display, drawable, gc, tkfont, string, x, y, firstChar, lastChar***)**

ARGUMENTS

Tk_Font	*tkfont*	(in)	Token for font in which text is to be drawn or measured. Must have been returned by a previous call to **Tk_GetFont**.
const char	∗*string*	(in)	Text to be measured or displayed. Need not be null terminated. Any non-printing meta-characters in the string (such as tabs, newlines, and other control characters) will be measured or displayed in a platform-dependent manner.
int	*maxChars*	(in)	The maximum number of characters to consider when measuring *string*. Must be greater than or equal to 0.
int	*maxPixels*	(in)	If *maxPixels* is greater than 0, it specifies the longest permissible line length in pixels. Characters from *string* are processed only until this many pixels have been covered. If *maxPixels* is <= 0, then the line length is unbounded and the *flags* argument is ignored.
int	*flags*	(in)	Various flag bits OR-ed together: TK_PARTIAL_OK means include a character as long as any part of it fits in the length given by *maxPixels*; otherwise, a character must fit completely to be considered. TK_WHOLE_WORDS means stop on a word boundary, if possible. If TK_AT_LEAST_ONE is set, it means return at least one character even if no characters could fit in the length given by *maxPixels*. If TK_AT_LEAST_ONE is set and TK_WHOLE_WORDS is also set, it means that if not even one word fits on the line, return the first few letters of the word that did fit; if not even one letter of the word fit, then the first letter will still be returned.
int	∗*lengthPtr*	(out)	Filled with the number of pixels occupied by the number of characters returned as the result of **Tk_MeasureChars**.
int	*numChars*	(in)	The total number of characters to measure or draw from *string*. Must be greater than or equal to 0.
Display	∗*display*	(in)	Display on which to draw.
Drawable	*drawable*	(in)	Window or pixmap in which to draw.

GC	*gc*	(in)	Graphics context for drawing characters. The font selected into this GC must be the same as the *tkfont*.
int	*x, y*	(in)	Coordinates at which to place the left edge of the baseline when displaying *string*.
int	*firstChar*	(in)	The index of the first character to underline in the *string*. Underlining begins at the left edge of this character.
int	*lastChar*	(in)	The index of the last character up to which the underline will be drawn. The character specified by *lastChar* will not itself be underlined.

DESCRIPTION

These routines are for measuring and displaying simple single-font, single-line, strings. To measure and display single-font, multi-line, justified text, refer to the documentation for **Tk_ComputeTextLayout**. There is no programming interface in the core of Tk that supports multi-font, multi-line text; support for that behavior must be built on top of simpler layers.

A glyph is the displayable picture of a letter, number, or some other symbol. Not all character codes in a given font have a glyph. Characters such as tabs, newlines/returns, and control characters that have no glyph are measured and displayed by these procedures in a platform-dependent manner; under X, they are replaced with backslashed escape sequences, while under Windows and Macintosh hollow or solid boxes may be substituted. Refer to the documentation for **Tk_ComputeTextLayout** for a programming interface that supports the platform-independent expansion of tab characters into columns and newlines/returns into multi-line text.

Tk_MeasureChars is used both to compute the length of a given string and to compute how many characters from a string fit in a given amount of space. The return value is the number of characters from *string* that fit in the space specified by *maxPixels* subject to the conditions described by *flags*. If all characters fit, the return value will be *maxChars*. ∗*lengthPtr* is filled with the computed width, in pixels, of the portion of the string that was measured. For example, if the return value is 5, then ∗*lengthPtr* is filled with the distance between the left edge of *string*[0] and the right edge of *string*[4].

Tk_TextWidth is a wrapper function that provides a simpler interface to the **Tk_MeasureChars** function. The return value is how much space in pixels the given *string* needs.

Tk_DrawChars draws the *string* at the given location in the given *drawable*.

Tk_UnderlineChars underlines the given range of characters in the given *string*. It doesn't draw the characters (which are assumed to have been displayed previously by **Tk_DrawChars**); it just draws the underline. This procedure is used to underline a few characters without having to construct an underlined font. To produce natively underlined text, the appropriate underlined font should be constructed and used.

KEYWORDS

font

NAME

Tk_MoveToplevelWindow − Adjust the position of a top-level window

SYNOPSIS

#include <tk.h>

Tk_MoveToplevelWindow(*tkwin, x, y***)**

ARGUMENTS

Tk_Window	*tkwin*	(in)	Token for top-level window to move.
int	*x*	(in)	New x-coordinate for the top-left pixel of *tkwin*'s border, or the top-left pixel of the decorative border supplied for *tkwin* by the window manager, if there is one.
int	*y*	(in)	New y-coordinate for the top-left pixel of *tkwin*'s border, or the top-left pixel of the decorative border supplied for *tkwin* by the window manager, if there is one.

DESCRIPTION

In general, a window should never set its own position; this should be done only by the geometry manger that is responsible for the window. For top-level windows the window manager is effectively the geometry manager; Tk provides interface code between the application and the window manager to convey the application's desires to the geometry manager. The desired size for a top-level window is conveyed using the usual **Tk_GeometryRequest** mechanism. The procedure **Tk_MoveToplevelWindow** may be used by an application to request a particular position for a top-level window; this procedure is similar in function to the **wm geometry** Tcl command except that negative offsets cannot be specified. It is invoked by widgets such as menus that want to appear at a particular place on the screen.

When **Tk_MoveToplevelWindow** is called it doesn't immediately pass on the new desired location to the window manager; it defers this action until all other outstanding work has been completed, using the **Tk_DoWhenIdle** mechanism.

KEYWORDS

position, top-level window, window manager

NAME

Tk_Name, Tk_PathName, Tk_NameToWindow − convert between names and window tokens

SYNOPSIS

```
#include <tk.h>

Tk_Uid
Tk_Name(tkwin)

char *
Tk_PathName(tkwin)

Tk_Window
Tk_NameToWindow(interp, pathName, tkwin)
```

ARGUMENTS

Tk_Window	*tkwin*	(in)	Token for window.
Tcl_Interp	**interp*	(out)	Interpreter to use for error reporting.
char	**pathName*	(in)	Character string containing path name of window.

DESCRIPTION

Each window managed by Tk has two names, a short name that identifies a window among children of the same parent, and a path name that identifies the window uniquely among all the windows belonging to the same main window. The path name is used more often in Tk than the short name; many commands, like **bind**, expect path names as arguments.

The **Tk_Name** macro returns a window's short name, which is the same as the *name* argument passed to **Tk_CreateWindow** when the window was created. The value is returned as a Tk_Uid, which may be used just like a string pointer but also has the properties of a unique identifier (see the manual entry for **Tk_GetUid** for details).

The **Tk_PathName** macro returns a hierarchical name for *tkwin*. Path names have a structure similar to file names in Unix but with dots between elements instead of slashes: the main window for an application has the path name ''.''; its children have names like ''.a'' and ''.b''; their children have names like ''.a.aa'' and ''.b.bb''; and so on. A window is considered to be be a child of another window for naming purposes if the second window was named as the first window's *parent* when the first window was created. This is not always the same as the X window hierarchy. For example, a pop-up is created as a child of the root window, but its logical parent will usually be a window within the application.

The procedure **Tk_NameToWindow** returns the token for a window given its path name (the *pathName* argument) and another window belonging to the same main window (*tkwin*). It normally returns a token for the named window, but if no such window exists **Tk_NameToWindow** leaves an error message in *interp->result* and returns NULL. The *tkwin* argument to **Tk_NameToWindow** is needed because path names are only unique within a single application hierarchy. If, for example, a single process has opened two main windows, each will have a separate naming hierarchy and the same path name might appear in each of the hierarchies. Normally *tkwin* is the main window of the desired hierarchy, but this need not be the case: any window in the desired hierarchy may be used.

KEYWORDS

name, path name, token, window

NAME

Tk_NameOfImage − Return name of image.

SYNOPSIS

#include <tk.h>

char *
Tk_NameOfImage(*typePtr*)

ARGUMENTS

Tk_ImageMaster	**masterPtr*	(in)	Token for image, which was passed to image manager's *createProc* when the image was created.

DESCRIPTION

This procedure is invoked by image managers to find out the name of an image. Given the token for the image, it returns the string name for the image.

KEYWORDS

image manager, image name

NAME

Tk_OwnSelection − make a window the owner of the primary selection

SYNOPSIS

#include <tk.h>

Tk_OwnSelection(*tkwin, selection, proc, clientData*)

ARGUMENTS

Tk_Window	*tkwin*	(in)	Window that is to become new selection owner.
Atom	*selection*	(in)	The name of the selection to be owned, such as XA_PRIMARY.
Tk_LostSelProc	∗*proc*	(in)	Procedure to invoke when *tkwin* loses selection ownership later.
ClientData	*clientData*	(in)	Arbitrary one-word value to pass to *proc*.

DESCRIPTION

Tk_OwnSelection arranges for *tkwin* to become the new owner of the selection specified by the atom *selection*. After this call completes, future requests for the selection will be directed to handlers created for *tkwin* using **Tk_CreateSelHandler**. When *tkwin* eventually loses the selection ownership, *proc* will be invoked so that the window can clean itself up (e.g. by unhighlighting the selection). *Proc* should have arguments and result that match the type **Tk_LostSelProc**:

typedef void Tk_LostSelProc(ClientData *clientData*);

The *clientData* parameter to *proc* is a copy of the *clientData* argument given to **Tk_OwnSelection**, and is usually a pointer to a data structure containing application-specific information about *tkwin*.

KEYWORDS

own, selection owner

NAME

Tk_ParseArgv − process command-line options

SYNOPSIS

#include <tk.h>

int
Tk_ParseArgv(*interp, tkwin, argcPtr, argv, argTable, flags*)

ARGUMENTS

Tcl_Interp	∗*interp*	(in)	Interpreter to use for returning error messages.
Tk_Window	*tkwin*	(in)	Window to use when arguments specify Tk options. If NULL, then no Tk options will be processed.
int	*argcPtr*	(in/out)	Pointer to number of arguments in argv; gets modified to hold number of unprocessed arguments that remain after the call.
char	∗∗*argv*	(in/out)	Command line arguments passed to main program. Modified to hold unprocessed arguments that remain after the call.
Tk_ArgvInfo	∗*argTable*	(in)	Array of argument descriptors, terminated by element with type TK_ARGV_END.
int	*flags*	(in)	If non-zero, then it specifies one or more flags that control the parsing of arguments. Different flags may be OR'ed together. The flags currently defined are TK_ARGV_DONT_SKIP_FIRST_ARG, TK_ARGV_NO_ABBREV, TK_ARGV_NO_LEFTOVERS, and TK_ARGV_NO_DEFAULTS.

DESCRIPTION

Tk_ParseArgv processes an array of command-line arguments according to a table describing the kinds of arguments that are expected. Each of the arguments in *argv* is processed in turn: if it matches one of the entries in *argTable*, the argument is processed according to that entry and discarded. The arguments that do not match anything in *argTable* are copied down to the beginning of *argv* (retaining their original order) and returned to the caller. At the end of the call **Tk_ParseArgv** sets ∗*argcPtr* to hold the number of arguments that are left in *argv*, and *argv[*∗*argcPtr]* will hold the value NULL. Normally, **Tk_ParseArgv** assumes that *argv[0]* is a command name, so it is treated like an argument that doesn't match *argTable* and returned to the caller; however, if the TK_ARGV_DONT_SKIP_FIRST_ARG bit is set in *flags* then *argv[0]* will be processed just like the other elements of *argv*.

Tk_ParseArgv normally returns the value TCL_OK. If an error occurs while parsing the arguments, then TCL_ERROR is returned and **Tk_ParseArgv** will leave an error message in *interp->result* in the standard Tcl fashion. In the event of an error return, ∗*argvPtr* will not have been modified, but *argv* could have been partially modified. The possible causes of errors are explained below.

The *argTable* array specifies the kinds of arguments that are expected; each of its entries has the following structure:

```
typedef struct {
    char *key;
    int type;
    char *src;
    char *dst;
    char *help;
} Tk_ArgvInfo;
```

The *key* field is a string such as ''−display'' or ''−bg'' that is compared with the values in *argv*. *Type*

indicates how to process an argument that matches *key* (more on this below). *Src* and *dst* are additional values used in processing the argument. Their exact usage depends on *type*, but typically *src* indicates a value and *dst* indicates where to store the value. The **char** * declarations for *src* and *dst* are placeholders: the actual types may be different. Lastly, *help* is a string giving a brief description of this option; this string is printed when users ask for help about command-line options.

When processing an argument in *argv*, **Tk_ParseArgv** compares the argument to each of the *key*'s in *argTable*. **Tk_ParseArgv** selects the first specifier whose *key* matches the argument exactly, if such a specifier exists. Otherwise **Tk_ParseArgv** selects a specifier for which the argument is a unique abbreviation. If the argument is a unique abbreviation for more than one specifier, then an error is returned. If there is no matching entry in *argTable*, then the argument is skipped and returned to the caller.

Once a matching argument specifier is found, **Tk_ParseArgv** processes the argument according to the *type* field of the specifier. The argument that matched *key* is called ''the matching argument'' in the descriptions below. As part of the processing, **Tk_ParseArgv** may also use the next argument in *argv* after the matching argument, which is called ''the following argument''. The legal values for *type*, and the processing that they cause, are as follows:

TK_ARGV_END
Marks the end of the table. The last entry in *argTable* must have this type; all of its other fields are ignored and it will never match any arguments.

TK_ARGV_CONSTANT
Src is treated as an integer and *dst* is treated as a pointer to an integer. *Src* is stored at **dst*. The matching argument is discarded.

TK_ARGV_INT
The following argument must contain an integer string in the format accepted by **strtol** (e.g. ''0'' and ''0x'' prefixes may be used to specify octal or hexadecimal numbers, respectively). *Dst* is treated as a pointer to an integer; the following argument is converted to an integer value and stored at **dst*. *Src* is ignored. The matching and following arguments are discarded from *argv*.

TK_ARGV_FLOAT
The following argument must contain a floating-point number in the format accepted by **strtol**. *Dst* is treated as the address of an double-precision floating point value; the following argument is converted to a double-precision value and stored at **dst*. The matching and following arguments are discarded from *argv*.

TK_ARGV_STRING
In this form, *dst* is treated as a pointer to a (char *); **Tk_ParseArgv** stores at **dst* a pointer to the following argument, and discards the matching and following arguments from *argv*. *Src* is ignored.

TK_ARGV_UID
This form is similar to TK_ARGV_STRING, except that the argument is turned into a Tk_Uid by calling **Tk_GetUid**. *Dst* is treated as a pointer to a Tk_Uid; **Tk_ParseArgv** stores at **dst* the Tk_Uid corresponding to the following argument, and discards the matching and following arguments from *argv*. *Src* is ignored.

TK_ARGV_CONST_OPTION
This form causes a Tk option to be set (as if the **option** command had been invoked). The *src* field is treated as a pointer to a string giving the value of an option, and *dst* is treated as a pointer to the name of the option. The matching argument is discarded. If *tkwin* is NULL, then argument specifiers of this type are ignored (as if they did not exist).

TK_ARGV_OPTION_VALUE
This form is similar to TK_ARGV_CONST_OPTION, except that the value of the option is taken from the following argument instead of from *src*. *Dst* is used as the name of the option. *Src* is

ignored. The matching and following arguments are discarded. If *tkwin* is NULL, then argument specifiers of this type are ignored (as if they did not exist).

TK_ARGV_OPTION_NAME_VALUE
In this case the following argument is taken as the name of a Tk option and the argument after that is taken as the value for that option. Both *src* and *dst* are ignored. All three arguments are discarded from *argv*. If *tkwin* is NULL, then argument specifiers of this type are ignored (as if they did not exist).

TK_ARGV_HELP
When this kind of option is encountered, **Tk_ParseArgv** uses the *help* fields of *argTable* to format a message describing all the valid arguments. The message is placed in *interp->result* and **Tk_ParseArgv** returns TCL_ERROR. When this happens, the caller normally prints the help message and aborts. If the *key* field of a TK_ARGV_HELP specifier is NULL, then the specifier will never match any arguments; in this case the specifier simply provides extra documentation, which will be included when some other TK_ARGV_HELP entry causes help information to be returned.

TK_ARGV_REST
This option is used by programs or commands that allow the last several of their options to be the name and/or options for some other program. If a **TK_ARGV_REST** argument is found, then **Tk_ParseArgv** doesn't process any of the remaining arguments; it returns them all at the beginning of *argv* (along with any other unprocessed arguments). In addition, **Tk_ParseArgv** treats *dst* as the address of an integer value, and stores at **dst* the index of the first of the **TK_ARGV_REST** options in the returned *argv*. This allows the program to distinguish the **TK_ARGV_REST** options from other unprocessed options that preceded the **TK_ARGV_REST**.

TK_ARGV_FUNC
For this kind of argument, *src* is treated as the address of a procedure, which is invoked to process the following argument. The procedure should have the following structure:

```
int
func(dst, key, nextArg)
    char *dst;
    char *key;
    char *nextArg;
{
}
```

The *dst* and *key* parameters will contain the corresponding fields from the *argTable* entry, and *nextArg* will point to the following argument from *argv* (or NULL if there aren't any more arguments left in *argv*). If *func* uses *nextArg* (so that **Tk_ParseArgv** should discard it), then it should return 1. Otherwise it should return 0 and **TkParseArgv** will process the following argument in the normal fashion. In either event the matching argument is discarded.

TK_ARGV_GENFUNC
This form provides a more general procedural escape. It treats *src* as the address of a procedure, and passes that procedure all of the remaining arguments. The procedure should have the following form:

```
int
genfunc(dst, interp, key, argc, argv)
    char *dst;
    Tcl_Interp *interp;
    char *key;
    int argc;
    char **argv;
```

```
{
}
```

The *dst* and *key* parameters will contain the corresponding fields from the *argTable* entry. *Interp* will be the same as the *interp* argument to **Tcl_ParseArgv**. *Argc* and *argv* refer to all of the options after the matching one. *Genfunc* should behave in a fashion similar to **Tk_ParseArgv**: parse as many of the remaining arguments as it can, then return any that are left by compacting them to the beginning of *argv* (starting at *argv*[0]). *Genfunc* should return a count of how many arguments are left in *argv*; **Tk_ParseArgv** will process them. If *genfunc* encounters an error then it should leave an error message in *interp->result*, in the usual Tcl fashion, and return -1; when this happens **Tk_ParseArgv** will abort its processing and return TCL_ERROR.

FLAGS

TK_ARGV_DONT_SKIP_FIRST_ARG

Tk_ParseArgv normally treats *argv[0]* as a program or command name, and returns it to the caller just as if it hadn't matched *argTable*. If this flag is given, then *argv[0]* is not given special treatment.

TK_ARGV_NO_ABBREV

Normally, **Tk_ParseArgv** accepts unique abbreviations for *key* values in *argTable*. If this flag is given then only exact matches will be acceptable.

TK_ARGV_NO_LEFTOVERS

Normally, **Tk_ParseArgv** returns unrecognized arguments to the caller. If this bit is set in *flags* then **Tk_ParseArgv** will return an error if it encounters any argument that doesn't match *argTable*. The only exception to this rule is *argv[0]*, which will be returned to the caller with no errors as long as TK_ARGV_DONT_SKIP_FIRST_ARG isn't specified.

TK_ARGV_NO_DEFAULTS

Normally, **Tk_ParseArgv** searches an internal table of standard argument specifiers in addition to *argTable*. If this bit is set in *flags*, then **Tk_ParseArgv** will use only *argTable* and not its default table.

EXAMPLE

Here is an example definition of an *argTable* and some sample command lines that use the options. Note the effect on *argc* and *argv*; arguments processed by **Tk_ParseArgv** are eliminated from *argv*, and *argc* is updated to reflect reduced number of arguments.

```
/*
 * Define and set default values for globals.
 */
int debugFlag = 0;
int numReps = 100;
char defaultFileName[] = "out";
char *fileName = defaultFileName;
Boolean exec = FALSE;

/*
 * Define option descriptions.
 */
Tk_ArgvInfo argTable[] = {
    {"-X", TK_ARGV_CONSTANT, (char *) 1, (char *) &debugFlag,
        "Turn on debugging printfs"},
    {"-N", TK_ARGV_INT, (char *) NULL, (char *) &numReps,
        "Number of repetitions"},
```

```
        {"-of", TK_ARGV_STRING, (char *) NULL, (char *) &fileName,
            "Name of file for output"},
        {"x", TK_ARGV_REST, (char *) NULL, (char *) &exec,
            "File to exec, followed by any arguments (must be last argument)."},
        {(char *) NULL, TK_ARGV_END, (char *) NULL, (char *) NULL,
           (char *) NULL}
};

main(argc, argv)
    int argc;
    char *argv[];
{
    ...

    if (Tk_ParseArgv(interp, tkwin, &argc, argv, argTable, 0) != TCL_OK) {
        fprintf(stderr, "%s\n", interp->result);
        exit(1);
    }

    /*
     * Remainder of the program.
     */
}
```

Note that default values can be assigned to variables named in *argTable*: the variables will only be overwritten if the particular arguments are present in *argv*. Here are some example command lines and their effects.

```
prog -N 200 infile# just sets the numReps variable to 200
prog -of out200 infile # sets fileName to reference "out200"
prog -XN 10 infile# sets the debug flag, also sets numReps
```

In all of the above examples, *argc* will be set by **Tk_ParseArgv** to 2, *argv*[0] will be ‘‘prog’’, *argv*[1] will be ‘‘infile’’, and *argv*[2] will be NULL.

KEYWORDS

arguments, command line, options

NAME

Tk_QueueWindowEvent − Add a window event to the Tcl event queue

SYNOPSIS

#include <tk.h>

Tk_QueueWindowEvent(*eventPtr, position*)

ARGUMENTS

XEvent	∗*eventPtr*	(in)	An event to add to the event queue.
Tcl_QueuePosition	*position*	(in)	Where to add the new event in the queue: **TCL_QUEUE_TAIL**, **TCL_QUEUE_HEAD**, or **TCL_QUEUE_MARK**.

DESCRIPTION

This procedure places a window event on Tcl's internal event queue for eventual servicing. It creates a Tcl_Event structure, copies the event into that structure, and calls **Tcl_QueueEvent** to add the event to the queue. When the event is eventually removed from the queue it is processed just like all window events.

The *position* argument to **Tk_QueueWindowEvent** has the same significance as for **Tcl_QueueEvent**; see the documentation for **Tcl_QueueEvent** for details.

KEYWORDS

callback, clock, handler, modal timeout

NAME

Tk_RestackWindow − Change a window's position in the stacking order

SYNOPSIS

#include <tk.h>

int
Tk_RestackWindow(*tkwin, aboveBelow, other*)

ARGUMENTS

Tk_Window	*tkwin*	(in)	Token for window to restack.
int	*aboveBelow*	(in)	Indicates new position of *tkwin* relative to *other*; must be **Above** or **Below**.
Tk_Window	*other*	(in)	*Tkwin* will be repositioned just above or below this window. Must be a sibling of *tkwin* or a descendant of a sibling. If NULL then *tkwin* is restacked above or below all siblings.

DESCRIPTION

Tk_RestackWindow changes the stacking order of *window* relative to its siblings. If *other* is specified as NULL then *window* is repositioned at the top or bottom of its stacking order, depending on whether *aboveBelow* is **Above** or **Below**. If *other* has a non-NULL value then *window* is repositioned just above or below *other*.

The *aboveBelow* argument must have one of the symbolic values **Above** or **Below**. Both of these values are defined by the include file <X11/Xlib.h>.

KEYWORDS

above, below, obscure, stacking order

NAME

Tk_RestrictEvents − filter and selectively delay X events

SYNOPSIS

#include <tk.h>

Tk_RestrictProc ∗
Tk_RestrictEvents(*proc*, *clientData*, *prevClientDataPtr*)

ARGUMENTS

Tk_RestrictProc	∗*proc*	(in)	Predicate procedure to call to filter incoming X events. NULL means do not restrict events at all.
ClientData	*clientData*	(in)	Arbitrary argument to pass to *proc*.
ClientData	∗*prevClientDataPtr*	(out)	Pointer to place to save argument to previous restrict procedure.

DESCRIPTION

This procedure is useful in certain situations where applications are only prepared to receive certain X events. After **Tk_RestrictEvents** is called, **Tk_DoOneEvent** (and hence **Tk_MainLoop**) will filter X input events through *proc*. *Proc* indicates whether a given event is to be processed immediately, deferred until some later time (e.g. when the event restriction is lifted), or discarded. *Proc* is a procedure with arguments and result that match the type **Tk_RestrictProc**:

```
typedef Tk_RestrictAction Tk_RestrictProc(
    ClientData clientData,
    XEvent ∗eventPtr);
```

The *clientData* argument is a copy of the *clientData* passed to **Tk_RestrictEvents**; it may be used to provide *proc* with information it needs to filter events. The *eventPtr* points to an event under consideration. *Proc* returns a restrict action (enumerated type **Tk_RestrictAction**) that indicates what **Tk_DoOneEvent** should do with the event. If the return value is **TK_PROCESS_EVENT**, then the event will be handled immediately. If the return value is **TK_DEFER_EVENT**, then the event will be left on the event queue for later processing. If the return value is **TK_DISCARD_EVENT**, then the event will be removed from the event queue and discarded without being processed.

Tk_RestrictEvents uses its return value and *prevClientDataPtr* to return information about the current event restriction procedure (a NULL return value means there are currently no restrictions). These values may be used to restore the previous restriction state when there is no longer any need for the current restriction.

There are very few places where **Tk_RestrictEvents** is needed. In most cases, the best way to restrict events is by changing the bindings with the **bind** Tcl command or by calling **Tk_CreateEventHandler** and **Tk_DeleteEventHandler** from C. The main place where **Tk_RestrictEvents** must be used is when performing synchronous actions (for example, if you need to wait for a particular event to occur on a particular window but you don't want to invoke any handlers for any other events). The ''obvious'' solution in these situations is to call **XNextEvent** or **XWindowEvent**, but these procedures cannot be used because Tk keeps its own event queue that is separate from the X event queue. Instead, call **Tk_RestrictEvents** to set up a filter, then call **Tk_DoOneEvent** to retrieve the desired event(s).

KEYWORDS

delay, event, filter, restriction

NAME

Tk_SetAppName − Set the name of an application for ‘‘send’’ commands

SYNOPSIS

#include <tk.h>

char ∗
Tk_SetAppName(*tkwin, name*)

ARGUMENTS

Tk_Window	*tkwin*	(in)	Token for window in application. Used only to select a particular application.
char	∗*name*	(in)	Name under which to register the application.

DESCRIPTION

Tk_SetAppName associates a name with a given application and records that association on the display containing with the application’s main window. After this procedure has been invoked, other applications on the display will be able to use the **send** command to invoke operations in the application. If *name* is already in use by some other application on the display, then a new name will be generated by appending ‘‘ **#2**’’ to *name*; if this name is also in use, the number will be incremented until an unused name is found. The return value from the procedure is a pointer to the name actually used.

If the application already has a name when **Tk_SetAppName** is called, then the new name replaces the old name.

Tk_SetAppName also adds a **send** command to the application’s interpreter, which can be used to send commands from this application to others on any of the displays where the application has windows.

The application’s name registration persists until the interpreter is deleted or the **send** command is deleted from *interp*, at which point the name is automatically unregistered and the application becomes inaccessible via **send**. The application can be made accessible again by calling **Tk_SetAppName**.

Tk_SetAppName is called automatically by **Tk_Init**, so applications don’t normally need to call it explicitly.

The command **tk appname** provides Tcl-level access to the functionality of **Tk_SetAppName**.

KEYWORDS

application, name, register, send command

NAME

Tk_SetClass, Tk_Class − set or retrieve a window's class

SYNOPSIS

#include <tk.h>

Tk_SetClass(*tkwin, class*)

Tk_Uid
Tk_Class(*tkwin*)

ARGUMENTS

Tk_Window	*tkwin*	(in)	Token for window.
char	∗*class*	(in)	New class name for window.

DESCRIPTION

Tk_SetClass is called to associate a class with a particular window. The *class* string identifies the type of the window; all windows with the same general class of behavior (button, menu, etc.) should have the same class. By convention all class names start with a capital letter, and there exists a Tcl command with the same name as each class (except all in lower-case) which can be used to create and manipulate windows of that class. A window's class string is initialized to NULL when the window is created.

For main windows, Tk automatically propagates the name and class to the WM_CLASS property used by window managers. This happens either when a main window is actually created (e.g. in **Tk_MakeWindowExist**), or when **Tk_SetClass** is called, whichever occurs later. If a main window has not been assigned a class then Tk will not set the WM_CLASS property for the window.

Tk_Class is a macro that returns the current value of *tkwin*'s class. The value is returned as a Tk_Uid, which may be used just like a string pointer but also has the properties of a unique identifier (see the manual entry for **Tk_GetUid** for details). If *tkwin* has not yet been given a class, then **Tk_Class** will return NULL.

KEYWORDS

class, unique identifier, window, window manager

NAME

Tk_SetGrid, Tk_UnsetGrid − control the grid for interactive resizing

SYNOPSIS

#include <tk.h>

Tk_SetGrid(*tkwin, reqWidth, reqHeight, widthInc, heightInc*)

Tk_UnsetGrid(*tkwin*)

ARGUMENTS

Tk_Window	*tkwin*	(in)	Token for window.
int	*reqWidth*	(in)	Width in grid units that corresponds to the pixel dimension *tkwin* has requested via **Tk_GeometryRequest**.
int	*reqHeight*	(in)	Height in grid units that corresponds to the pixel dimension *tkwin* has requested via **Tk_GeometryRequest**.
int	*widthInc*	(in)	Width of one grid unit, in pixels.
int	*heightInc*	(in)	Height of one grid unit, in pixels.

DESCRIPTION

Tk_SetGrid turns on gridded geometry management for *tkwin*'s toplevel window and specifies the geometry of the grid. **Tk_SetGrid** is typically invoked by a widget when its **setGrid** option is true. It restricts interactive resizing of *tkwin*'s toplevel window so that the space allocated to the toplevel is equal to its requested size plus or minus even multiples of *widthInc* and *heightInc*. Furthermore, the *reqWidth* and *reqHeight* values are passed to the window manager so that it can report the window's size in grid units during interactive resizes. If *tkwin*'s configuration changes (e.g., the size of a grid unit changes) then the widget should invoke **Tk_SetGrid** again with the new information.

Tk_UnsetGrid cancels gridded geometry management for *tkwin*'s toplevel window.

For each toplevel window there can be at most one internal window with gridding enabled. If **Tk_SetGrid** or **Tk_UnsetGrid** is invoked when some other window is already controlling gridding for *tkwin*'s toplevel, the calls for the new window have no effect.

See the **wm** manual entry for additional information on gridded geometry management.

KEYWORDS

grid, window, window manager

NAME

Tk_SetWindowVisual − change visual characteristics of window

SYNOPSIS

#include <tk.h>

int
Tk_SetWindowVisual(*tkwin, visual, depth, colormap*)

ARGUMENTS

Tk_Window	*tkwin*	(in)	Token for window.
Visual	**visual*	(in)	New visual type to use for *tkwin*.
int	*depth*	(in)	Number of bits per pixel desired for *tkwin*.
Colormap	*colormap*	(in)	New colormap for *tkwin*, which must be compatible with *visual* and *depth*.

DESCRIPTION

When Tk creates a new window it assigns it the default visual characteristics (visual, depth, and colormap) for its screen. **Tk_SetWindowVisual** may be called to change them. **Tk_SetWindowVisual** must be called before the window has actually been created in X (e.g. before **Tk_MapWindow** or **Tk_MakeWindowExist** has been invoked for the window). The safest thing is to call **Tk_SetWindowVisual** immediately after calling **Tk_CreateWindow**. If *tkwin* has already been created before **Tk_SetWindowVisual** is called then it returns 0 and doesn't make any changes; otherwise it returns 1 to signify that the operation completed successfully.

Note: **Tk_SetWindowVisual** should not be called if you just want to change a window's colormap without changing its visual or depth; call **Tk_SetWindowColormap** instead.

KEYWORDS

colormap, depth, visual

NAME

Tk_StrictMotif − Return value of tk_strictMotif variable

SYNOPSIS

#include <tk.h>

int
Tk_StrictMotif(*tkwin*)

ARGUMENTS

Tk_Window	*tkwin*	(in)	Token for window.

DESCRIPTION

This procedure returns the current value of the **tk_strictMotif** variable in the interpreter associated with *tkwin*'s application. The value is returned as an integer that is either 0 or 1. 1 means that strict Motif compliance has been requested, so anything that is not part of the Motif specification should be avoided. 0 means that ''Motif-like'' is good enough, and extra features are welcome.

This procedure uses a link to the Tcl variable to provide much faster access to the variable's value than could be had by calling **Tcl_GetVar**.

KEYWORDS

Motif compliance, tk_strictMotif variable

NAME

Tk_ComputeTextLayout, Tk_FreeTextLayout, Tk_DrawTextLayout, Tk_UnderlineTextLayout, Tk_PointToChar, Tk_CharBbox, Tk_DistanceToTextLayout, Tk_IntersectTextLayout, Tk_TextLayoutToPostscript − routines to measure and display single-font, multi-line, justified text.

SYNOPSIS

#include <tk.h>

Tk_TextLayout
Tk_ComputeTextLayout(*tkfont, string, numChars, wrapLength, justify, flags, widthPtr, heightPtr***)**

void
Tk_FreeTextLayout(*layout***)**

void
Tk_DrawTextLayout(*display, drawable, gc, layout, x, y, firstChar, lastChar***)**

void
Tk_UnderlineTextLayout(*display, drawable, gc, layout, x, y, underline***)**

int
Tk_PointToChar(*layout, x, y***)**

int
Tk_CharBbox(*layout, index, xPtr, yPtr, widthPtr, heightPtr***)**

int
Tk_DistanceToTextLayout(*layout, x, y***)**

int
Tk_IntersectTextLayout(*layout, x, y, width, height***)**

void
Tk_TextLayoutToPostscript(*interp, layout***)**

ARGUMENTS

Tk_Font	*tkfont*	(in)	Font to use when constructing and displaying a text layout. The *tkfont* must remain valid for the lifetime of the text layout. Must have been returned by a previous call to **Tk_GetFont**.
const char	∗*string*	(in)	Potentially multi-line string whose dimensions are to be computed and stored in the text layout. The *string* must remain valid for the lifetime of the text layout.
int	*numChars*	(in)	The number of characters to consider from *string*. If *numChars* is less than 0, then assumes *string* is null terminated and uses **strlen(***string***)**.
int	*wrapLength*	(in)	Longest permissible line length, in pixels. Lines in *string* will automatically be broken at word boundaries and wrapped when they reach this length. If *wrapLength* is too small for even a single character to fit on a line, it will be expanded to allow one character to fit on each line. If *wrapLength* is <= 0, there is no automatic wrapping; lines will get as long as they

			need to be and only wrap if a newline/return character is encountered.
Tk_Justify	*justify*	(in)	How to justify the lines in a multi-line text layout. Possible values are TK_JUSTIFY_LEFT, TK_JUSTIFY_CENTER, or TK_JUSTIFY_RIGHT. If the text layout only occupies a single line, then *justify* is irrelevant.
int	*flags*	(in)	Various flag bits OR-ed together. TK_IGNORE_TABS means that tab characters should not be expanded to the next tab stop. TK_IGNORE_NEWLINES means that newline/return characters should not cause a line break. If either tabs or newlines/returns are ignored, then they will be treated as regular characters, being measured and displayed in a platform-dependent manner as described in **Tk_MeasureChars**, and will not have any special behaviors.
int	**widthPtr*	(out)	If non-NULL, filled with either the width, in pixels, of the widest line in the text layout, or the width, in pixels, of the bounding box for the character specified by *index*.
int	**heightPtr*	(out)	If non-NULL, filled with either the total height, in pixels, of all the lines in the text layout, or the height, in pixels, of the bounding box for the character specified by *index*.
Tk_TextLayout	*layout*	(in)	A token that represents the cached layout information about the single-font, multi-line, justified piece of text. This token is returned by **Tk_ComputeTextLayout**.
Display	**display*	(in)	Display on which to draw.
Drawable	*drawable*	(in)	Window or pixmap in which to draw.
GC	*gc*	(in)	Graphics context to use for drawing text layout. The font selected in this GC must correspond to the *tkfont* used when constructing the text layout.
int	*x, y*	(in)	Point, in pixels, at which to place the upper-left hand corner of the text layout when it is being drawn, or the coordinates of a point (with respect to the upper-left hand corner of the text layout) to check against the text layout.
int	*firstChar*	(in)	The index of the first character to draw from the given text layout. The number 0 means to draw from the beginning.
int	*lastChar*	(in)	The index of the last character up to which to draw. The character specified by *lastChar* itself will not be drawn. A number less than 0 means to draw all characters in the text layout.
int	*underline*	(in)	Index of the single character to underline in the text layout, or a number less than 0 for no underline.
int	*index*	(in)	The index of the character whose bounding box is desired. The bounding box is computed with respect to the upper-left hand corner of the text layout.
int	**xPtr, *yPtr*	(out)	Filled with the upper-left hand corner, in pixels, of the bounding box for the character specified by *index*. Either or both *xPtr* and *yPtr* may be NULL, in which case the corresponding value is not calculated.

int	*width, height*	(in)	Specifies the width and height, in pixels, of the rectangular area to compare for intersection against the text layout.
Tcl_Interp	∗*interp*	(out)	Postscript code that will print the text layout is appended to *interp->result*.

DESCRIPTION

These routines are for measuring and displaying single-font, multi-line, justified text. To measure and display simple single-font, single-line strings, refer to the documentation for **Tk_MeasureChars**. There is no programming interface in the core of Tk that supports multi-font, multi-line text; support for that behavior must be built on top of simpler layers.

The routines described here are built on top of the programming interface described in the **Tk_MeasureChars** documentation. Tab characters and newline/return characters may be treated specially by these procedures, but all other characters are passed through to the lower level.

Tk_ComputeTextLayout computes the layout information needed to display a single-font, multi-line, justified *string* of text and returns a Tk_TextLayout token that holds this information. This token is used in subsequent calls to procedures such as **Tk_DrawTextLayout**, **Tk_DistanceToTextLayout**, and **Tk_FreeTextLayout**. The *string* and *tkfont* used when computing the layout must remain valid for the lifetime of this token.

Tk_FreeTextLayout is called to release the storage associated with *layout* when it is no longer needed. A *layout* should not be used in any other text layout procedures once it has been released.

Tk_DrawTextLayout uses the information in *layout* to display a single-font, multi-line, justified string of text at the specified location.

Tk_UnderlineTextLayout uses the information in *layout* to display an underline below an individual character. This procedure does not draw the text, just the underline. To produce natively underlined text, an underlined font should be constructed and used. All characters, including tabs, newline/return characters, and spaces at the ends of lines, can be underlined using this method. However, the underline will never be drawn outside of the computed width of *layout*; the underline will stop at the edge for any character that would extend partially outside of *layout*, and the underline will not be visible at all for any character that would be located completely outside of the layout.

Tk_PointToChar uses the information in *layout* to determine the character closest to the given point. The point is specified with respect to the upper-left hand corner of the *layout*, which is considered to be located at (0, 0). Any point whose *y*-value is less that 0 will be considered closest to the first character in the text layout; any point whose *y*-value is greater than the height of the text layout will be considered closest to the last character in the text layout. Any point whose *x*-value is less than 0 will be considered closest to the first character on that line; any point whose *x*-value is greater than the width of the text layout will be considered closest to the last character on that line. The return value is the index of the character that was closest to the point. Given a *layout* with no characters, the value 0 will always be returned, referring to a hypothetical zero-width placeholder character.

Tk_CharBBox uses the information in *layout* to return the bounding box for the character specified by *index*. The width of the bounding box is the advance width of the character, and does not include any left or right bearing. Any character that extends partially outside of *layout* is considered to be truncated at the edge. Any character that would be located completely outside of *layout* is considered to be zero-width and pegged against the edge. The height of the bounding box is the line height for this font, extending from the top of the ascent to the bottom of the descent; information about the actual height of individual letters is not available. For measurement purposes, a *layout* that contains no characters is considered to contain a single zero-width placeholder character at index 0. If *index* was not a valid character index, the return value is 0 and ∗*xPtr*, ∗*yPtr*, ∗*widthPtr*, and ∗*heightPtr* are unmodified. Otherwise, if *index* did specify a valid, the

return value is non-zero, and **xPtr*, **yPtr*, **widthPtr*, and **heightPtr* are filled with the bounding box information for the character. If any of *xPtr*, *yPtr*, *widthPtr*, or *heightPtr* are NULL, the corresponding value is not calculated or stored.

Tk_DistanceToTextLayout computes the shortest distance in pixels from the given point (*x*, *y*) to the characters in *layout*. Newline/return characters and non-displaying space characters that occur at the end of individual lines in the text layout are ignored for hit detection purposes, but tab characters are not. The return value is 0 if the point actually hits the *layout*. If the point didn't hit the *layout* then the return value is the distance in pixels from the point to the *layout*.

Tk_IntersectTextLayout determines whether a *layout* lies entirely inside, entirely outside, or overlaps a given rectangle. Newline/return characters and non-displaying space characters that occur at the end of individual lines in the *layout* are ignored for intersection calculations. The return value is −1 if the *layout* is entirely outside of the rectangle, 0 if it overlaps, and 1 if it is entirely inside of the rectangle.

Tk_TextLayoutToPostscript outputs code consisting of a Postscript array of strings that represent the individual lines in *layout*. It is the responsibility of the caller to take the Postscript array of strings and add some Postscript function operate on the array to render each of the lines. The code that represents the Postscript array of strings is appended to *interp->result*.

DISPLAY MODEL

When measuring a text layout, space characters that occur at the end of a line are ignored. The space characters still exist and the insertion point can be positioned amongst them, but their additional width is ignored when justifying lines or returning the total width of a text layout. All end-of-line space characters are considered to be attached to the right edge of the line; this behavior is logical for left-justified text and reasonable for center-justified text, but not very useful when editing right-justified text. Spaces are considered variable width characters; the first space that extends past the edge of the text layout is clipped to the edge, and any subsequent spaces on the line are considered zero width and pegged against the edge. Space characters that occur in the middle of a line of text are not suppressed and occupy their normal space width.

Tab characters are not ignored for measurement calculations. If wrapping is turned on and there are enough tabs on a line, the next tab will wrap to the beginning of the next line. There are some possible strange interactions between tabs and justification; tab positions are calculated and the line length computed in a left-justified world, and then the whole resulting line is shifted so it is centered or right-justified, causing the tab columns not to align any more.

When wrapping is turned on, lines may wrap at word breaks (space or tab characters) or newline/returns. A dash or hyphen character in the middle of a word is not considered a word break. **Tk_ComputeTextLayout** always attempts to place at least one word on each line. If it cannot because the *wrapLength* is too small, the word will be broken and as much as fits placed on the line and the rest on subsequent line(s). If *wrapLength* is so small that not even one character can fit on a given line, the *wrapLength* is ignored for that line and one character will be placed on the line anyhow. When wrapping is turned off, only newline/return characters may cause a line break.

When a text layout has been created using an underlined *tkfont*, then any space characters that occur at the end of individual lines, newlines/returns, and tabs will not be displayed underlined when **Tk_DrawTextLayout** is called, because those characters are never actually drawn − they are merely placeholders maintained in the *layout*.

KEYWORDS

font

NAME

Tk_Init − add Tk to an interpreter and make a new Tk application.

SYNOPSIS

#include <tk.h>

int
Tk_Init(*interp*)

ARGUMENTS

Tcl_Interp	∗*interp*	(in)	Interpreter in which to load Tk. Tk should not already be loaded in this interpreter.

DESCRIPTION

Tk_Init is the package initialization procedure for Tk. It is normally invoked by the **Tcl_AppInit** procedure for an application or by the **load** command. **Tk_Init** adds all of Tk's commands to *interp* and creates a new Tk application, including its main window. If the initialization is successful **Tk_Init** returns **TCL_OK**; if there is an error it returns **TCL_ERROR**. **Tk_Init** also leaves a result or error message in *interp->result*.

If there is a variable **argv** in *interp*, **Tk_Init** treats the contents of this variable as a list of options for the new Tk application. The options may have any of the forms documented for the **wish** application (in fact, **wish** uses Tk_Init to process its command-line arguments).

KEYWORDS

application, initialization, load, main window

NAME

Tk_Main − main program for Tk-based applications

SYNOPSIS

#include <tk.h>

Tk_Main(*argc, argv, appInitProc*)

ARGUMENTS

int	*argc*	(in)	Number of elements in *argv*.
char	∗*argv[]*	(in)	Array of strings containing command-line arguments.
Tcl_AppInitProc	∗*appInitProc*	(in)	Address of an application-specific initialization procedure. The value for this argument is usually **Tcl_AppInit**.

DESCRIPTION

Tk_Main acts as the main program for most Tk-based applications. Starting with Tk 4.0 it is not called **main** anymore because it is part of the Tk library and having a function **main** in a library (particularly a shared library) causes problems on many systems. Having **main** in the Tk library would also make it hard to use Tk in C++ programs, since C++ programs must have special C++ **main** functions.

Normally each application contains a small **main** function that does nothing but invoke **Tk_Main**. **Tk_Main** then does all the work of creating and running a **wish**-like application.

When it is has finished its own initialization, but before it processes commands, **Tk_Main** calls the procedure given by the *appInitProc* argument. This procedure provides a ''hook'' for the application to perform its own initialization, such as defining application-specific commands. The procedure must have an interface that matches the type **Tcl_AppInitProc**:

typedef int Tcl_AppInitProc(Tcl_Interp ∗*interp*);

AppInitProc is almost always a pointer to **Tcl_AppInit**; for more details on this procedure, see the documentation for **Tcl_AppInit**.

KEYWORDS

application-specific initialization, command-line arguments, main program

NAME

Tk_WindowId, Tk_Parent, Tk_Display, Tk_DisplayName, Tk_ScreenNumber, Tk_Screen, Tk_X, Tk_Y, Tk_Width, Tk_Height, Tk_Changes, Tk_Attributes, Tk_IsMapped, Tk_IsTopLevel, Tk_ReqWidth, Tk_ReqHeight, Tk_InternalBorderWidth, Tk_Visual, Tk_Depth, Tk_Colormap − retrieve information from Tk's local data structure

SYNOPSIS

#include <tk.h>

Window
Tk_WindowId(*tkwin*)

Tk_Window
Tk_Parent(*tkwin*)

Display ∗
Tk_Display(*tkwin*)

char ∗
Tk_DisplayName(*tkwin*)

int
Tk_ScreenNumber(*tkwin*)

Screen ∗
Tk_Screen(*tkwin*)

int
Tk_X(*tkwin*)

int
Tk_Y(*tkwin*)

int
Tk_Width(*tkwin*)

int
Tk_Height(*tkwin*)

XWindowChanges ∗
Tk_Changes(*tkwin*)

XSetWindowAttributes ∗
Tk_Attributes(*tkwin*)

int
Tk_IsMapped(*tkwin*)

int
Tk_IsTopLevel(*tkwin*)

int

Tk_ReqWidth(*tkwin*)

int
Tk_ReqHeight(*tkwin*)

int
Tk_InternalBorderWidth(*tkwin*)

Visual *
Tk_Visual(*tkwin*)

int
Tk_Depth(*tkwin*)

Colormap
Tk_Colormap(*tkwin*)

ARGUMENTS

Tk_Window	*tkwin*	(in)	Token for window.

DESCRIPTION

Tk_WindowID and the other names listed above are all macros that return fields from Tk's local data structure for *tkwin*. None of these macros requires any interaction with the server; it is safe to assume that all are fast.

Tk_WindowId returns the X identifier for *tkwin*, or **NULL** if no X window has been created for *tkwin* yet.

Tk_Parent returns Tk's token for the logical parent of *tkwin*. The parent is the token that was specified when *tkwin* was created, or NULL for main windows.

Tk_Display returns a pointer to the Xlib display structure corresponding to *tkwin*. **Tk_DisplayName** returns an ASCII string identifying *tkwin*'s display. **Tk_ScreenNumber** returns the index of *tkwin*'s screen among all the screens of *tkwin*'s display. **Tk_Screen** returns a pointer to the Xlib structure corresponding to *tkwin*'s screen.

Tk_X, **Tk_Y**, **Tk_Width**, and **Tk_Height** return information about *tkwin's* location within its parent and its size. The location information refers to the upper-left pixel in the window, or its border if there is one. The width and height information refers to the interior size of the window, not including any border. **Tk_Changes** returns a pointer to a structure containing all of the above information plus a few other fields. **Tk_Attributes** returns a pointer to an XSetWindowAttributes structure describing all of the attributes of the *tkwin*'s window, such as background pixmap, event mask, and so on (Tk keeps track of all this information as it is changed by the application). Note: it is essential that applications use Tk procedures like **Tk_ResizeWindow** instead of X procedures like **XResizeWindow**, so that Tk can keep its data structures up-to-date.

Tk_IsMapped returns a non-zero value if *tkwin* is mapped and zero if *tkwin* isn't mapped.

Tk_IsTopLevel returns a non-zero value if *tkwin* is a top-level window (its X parent is the root window of the screen) and zero if *tkwin* isn't a top-level window.

Tk_ReqWidth and **Tk_ReqHeight** return information about the window's requested size. These values correspond to the last call to **Tk_GeometryRequest** for *tkwin*.

Tk_InternalBorderWidth returns the width of internal border that has been requested for *tkwin*, or 0 if no internal border was requested. The return value is simply the last value passed to **Tk_SetInternalBorder** for *tkwin*.

Tk_Visual, **Tk_Depth**, and **Tk_Colormap** return information about the visual characteristics of a window. **Tk_Visual** returns the visual type for the window, **Tk_Depth** returns the number of bits per pixel, and **Tk_Colormap** returns the current colormap for the window. The visual characteristics are normally set from the defaults for the window's screen, but they may be overridden by calling **Tk_SetWindowVisual**.

KEYWORDS

attributes, colormap, depth, display, height, geometry manager, identifier, mapped, requested size, screen, top-level, visual, width, window, x, y

`tclsh`	Simple shell containing Tcl interpreter

Http	Client-side implementation of the HTTP/1.0 protocol.
Safe Base	A mechanism for creating and manipulating safe interpreters.
Tcl	Summary of Tcl language syntax.
after	Execute a command after a time delay
append	Append to variable
array	Manipulate array variables
bgerror	Command invoked to process background errors
binary	Insert and extract fields from binary strings
break	Abort looping command
case	Evaluate one of several scripts, depending on a given value
catch	Evaluate script and trap exceptional returns
cd	Change working directory
clock	Obtain and manipulate time
close	Close an open channel.
concat	Join lists together
continue	Skip to the next iteration of a loop
eof	Check for end of file condition on channel
error	Generate an error
eval	Evaluate a Tcl script
exec	Invoke subprocess(es)
exit	End the application
expr	Evaluate an expression
fblocked	Test whether the last input operation exhausted all available input
fconfigure	Set and get options on a channel
fcopy	Copy data from one channel to another.
file	Manipulate file names and attributes
fileevent	Execute a script when a channel becomes readable or writable
filename	File name conventions supported by Tcl commands
flush	Flush buffered output for a channel
for	‘‘For’’ loop
foreach	Iterate over all elements in one or more lists
format	Format a string in the style of sprintf
gets	Read a line from a channel
glob	Return names of files that match patterns
global	Access global variables
history	Manipulate the history list
if	Execute scripts conditionally
incr	Increment the value of a variable

info	Return information about the state of the Tcl interpreter
interp	Create and manipulate Tcl interpreters
join	Create a string by joining together list elements
lappend	Append list elements onto a variable
library	standard library of Tcl procedures
lindex	Retrieve an element from a list
linsert	Insert elements into a list
list	Create a list
llength	Count the number of elements in a list
load	Load machine code and initialize new commands.
lrange	Return one or more adjacent elements from a list
lreplace	Replace elements in a list with new elements
lsearch	See if a list contains a particular element
lsort	Sort the elements of a list
namespace	create and manipulate contexts for commands and variables
open	Open a file-based or command pipeline channel
package	Facilities for package loading and version control
pid	Retrieve process id(s)
pkg_mkIndex	Build an index for automatic loading of packages
proc	Create a Tcl procedure
puts	Write to a channel
pwd	Return the current working directory
read	Read from a channel
regexp	Match a regular expression against a string
registry	Manipulate the Windows registry
regsub	Perform substitutions based on regular expression pattern matching
rename	Rename or delete a command
resource	Manipulate Macintosh resources
return	Return from a procedure
scan	Parse string using conversion specifiers in the style of sscanf
seek	Change the access position for an open channel
set	Read and write variables
socket	Open a TCP network connection
source	Evaluate a file or resource as a Tcl script
split	Split a string into a proper Tcl list
string	Manipulate strings
subst	Perform backslash, command, and variable substitutions
switch	Evaluate one of several scripts, depending on a given value

`tclvars`	Variables used by Tcl
`tell`	Return current access position for an open channel
`time`	Time the execution of a script
`trace`	Monitor variable accesses
`unknown`	Handle attempts to use non-existent commands
`unset`	Delete variables
`update`	Process pending events and idle callbacks
`uplevel`	Execute a script in a different stack frame
`upvar`	Create link to variable in a different stack frame
`variable`	create and initialize a namespace variable
`vwait`	Process events until a variable is written
`while`	Execute script repeatedly as long as a condition is met

TclConcatObj	manipulate Tcl objects as strings
Tcl_AddErrorInfo	record information about errors
Tcl_AddObjErrorInfo	record information about errors
Tcl_Alloc	allocate or free heap memory
Tcl_AllowExceptions	allow all exceptions in next script evaluation
Tcl_AppInit	perform application-specific initialization
Tcl_AppendAllObjType	manipulate Tcl object types
Tcl_AppendElement	manipulate Tcl result
Tcl_AppendResult	manipulate Tcl result
Tcl_AppendStringsToO	manipulate Tcl objects as strings
Tcl_AppendToObj	manipulate Tcl objects as strings
Tcl_AsyncCreate	handle asynchronous events
Tcl_AsyncDelete	handle asynchronous events
Tcl_AsyncInvoke	handle asynchronous events
Tcl_AsyncMark	handle asynchronous events
Tcl_BackgroundError	report Tcl error that occurred in background processing
Tcl_Backslash	parse a backslash sequence
Tcl_BadChannelOption	procedures for creating and manipulating channels
Tcl_CallWhenDeleted	Arrange for callback when interpreter is deleted
Tcl_CancelIdleCall	invoke a procedure when there are no pending events
Tcl_Close	buffered I/O facilities using channels
Tcl_CommandComplete	Check for unmatched braces in a Tcl command
Tcl_Concat	concatenate a collection of strings
Tcl_ConvertElement	manipulate Tcl lists
Tcl_ConvertToType	manipulate Tcl object types
Tcl_CreateAlias	manage multiple Tcl interpreters, aliases and hidden commands.
Tcl_CreateAliasObj	manage multiple Tcl interpreters, aliases and hidden commands.
Tcl_CreateChannel	procedures for creating and manipulating channels
Tcl_CreateChannelHan	call a procedure when a channel becomes readable or writable
Tcl_CreateCloseHandl	arrange for callbacks when channels are closed
Tcl_CreateCommand	implement new commands in C
Tcl_CreateEventSourc	the event queue and notifier interfaces
Tcl_CreateExitHandle	end the application (and invoke exit handlers)
Tcl_CreateFileHandle	associate procedure callbacks with files or devices (Unix only)
Tcl_CreateHashEntry	procedures to manage hash tables
Tcl_CreateInterp	create and delete Tcl command interpreters
Tcl_CreateMathFunc	Define a new math function for expressions
Tcl_CreateObjCommand	implement new commands in C

Tcl_CreateSlave	manage multiple Tcl interpreters, aliases and hidden commands.
Tcl_CreateTimerHandl	call a procedure at a given time
Tcl_CreateTrace	arrange for command execution to be traced
Tcl_DStringAppend	manipulate dynamic strings
Tcl_DStringAppendEle	manipulate dynamic strings
Tcl_DStringEndSublis	manipulate dynamic strings
Tcl_DStringFree	manipulate dynamic strings
Tcl_DStringGetResult	manipulate dynamic strings
Tcl_DStringInit	manipulate dynamic strings
Tcl_DStringLength	manipulate dynamic strings
Tcl_DStringResult	manipulate dynamic strings
Tcl_DStringSetLength	manipulate dynamic strings
Tcl_DStringStartSubl	manipulate dynamic strings
Tcl_DStringValue	manipulate dynamic strings
Tcl_DecrRefCount	manipulate Tcl objects
Tcl_DeleteAssocData	manage associations of string keys and user specified data with Tcl interp
Tcl_DeleteChannelHan	call a procedure when a channel becomes readable or writable
Tcl_DeleteCloseHandl	arrange for callbacks when channels are closed
Tcl_DeleteCommand	implement new commands in C
Tcl_DeleteCommandFro	implement new commands in C
Tcl_DeleteEventSourc	the event queue and notifier interfaces
Tcl_DeleteEvents	the event queue and notifier interfaces
Tcl_DeleteExitHandle	end the application (and invoke exit handlers)
Tcl_DeleteFileHandle	associate procedure callbacks with files or devices (Unix only)
Tcl_DeleteHashEntry	procedures to manage hash tables
Tcl_DeleteHashTable	procedures to manage hash tables
Tcl_DeleteInterp	create and delete Tcl command interpreters
Tcl_DeleteTimerHandl	call a procedure at a given time
Tcl_DeleteTrace	arrange for command execution to be traced
Tcl_DetachPids	manage child processes in background
Tcl_DoOneEvent	wait for events and invoke event handlers
Tcl_DoWhenIdle	invoke a procedure when there are no pending events
Tcl_DontCallWhenDele	Arrange for callback when interpreter is deleted
Tcl_DuplicateObj	manipulate Tcl objects
Tcl_Eof	buffered I/O facilities using channels
Tcl_Eval	execute Tcl commands
Tcl_EvalFile	execute Tcl commands
Tcl_EvalObj	execute Tcl commands

Tcl_EventuallyFree	avoid freeing storage while it’s being used
Tcl_Exit	end the application (and invoke exit handlers)
Tcl_ExposeCommand	manage multiple Tcl interpreters, aliases and hidden commands.
Tcl_ExprBoolean	evaluate an expression
Tcl_ExprBooleanObj	evaluate an expression
Tcl_ExprDouble	evaluate an expression
Tcl_ExprDoubleObj	evaluate an expression
Tcl_ExprLong	evaluate an expression
Tcl_ExprLongObj	evaluate an expression
Tcl_ExprObj	evaluate an expression
Tcl_ExprString	evaluate an expression
Tcl_Finalize	end the application (and invoke exit handlers)
Tcl_FindExecutable	identify or return the name of the binary file containing the application
Tcl_FindHashEntry	procedures to manage hash tables
Tcl_FirstHashEntry	procedures to manage hash tables
Tcl_Flush	buffered I/O facilities using channels
Tcl_Free	allocate or free heap memory
Tcl_GetAlias	manage multiple Tcl interpreters, aliases and hidden commands.
Tcl_GetAliasObj	manage multiple Tcl interpreters, aliases and hidden commands.
Tcl_GetAssocData	manage associations of string keys and user specified data with Tcl interp
Tcl_GetBoolean	convert from string to integer, double, or boolean
Tcl_GetBooleanFromOb	manipulate Tcl objects as boolean values
Tcl_GetChannel	buffered I/O facilities using channels
Tcl_GetChannelBuffer	procedures for creating and manipulating channels
Tcl_GetChannelHandle	procedures for creating and manipulating channels
Tcl_GetChannelInstan	procedures for creating and manipulating channels
Tcl_GetChannelMode	procedures for creating and manipulating channels
Tcl_GetChannelName	procedures for creating and manipulating channels
Tcl_GetChannelOption	buffered I/O facilities using channels
Tcl_GetChannelType	procedures for creating and manipulating channels
Tcl_GetCommandInfo	implement new commands in C
Tcl_GetCommandName	implement new commands in C
Tcl_GetDouble	convert from string to integer, double, or boolean
Tcl_GetDoubleFromObj	manipulate Tcl objects as floating-point values
Tcl_GetErrno	manipulate errno to store and retrieve error codes
Tcl_GetHashKey	procedures to manage hash tables
Tcl_GetHashValue	procedures to manage hash tables
Tcl_GetIndexFromObj	lookup string in table of keywords

Tcl_GetInt	convert from string to integer, double, or boolean
Tcl_GetIntFromObj	manipulate Tcl objects as integers
Tcl_GetInterpPath	manage multiple Tcl interpreters, aliases and hidden commands.
Tcl_GetLongFromObj	manipulate Tcl objects as integers
Tcl_GetMaster	manage multiple Tcl interpreters, aliases and hidden commands.
Tcl_GetNameOfExecuta	identify or return the name of the binary file containing the application
Tcl_GetObjResult	manipulate Tcl result
Tcl_GetObjType	manipulate Tcl object types
Tcl_GetOpenFile	Get a standard IO File * handle from a channel. (Unix only)
Tcl_GetPathType	manipulate platform-dependent file paths
Tcl_GetServiceMode	the event queue and notifier interfaces
Tcl_GetSlave	manage multiple Tcl interpreters, aliases and hidden commands.
Tcl_GetStdChannel	procedures for retrieving and replacing the standard channels
Tcl_GetStringFromObj	manipulate Tcl objects as strings
Tcl_GetStringResult	manipulate Tcl result
Tcl_GetVar	manipulate Tcl variables
Tcl_GetVar2	manipulate Tcl variables
Tcl_Gets	buffered I/O facilities using channels
Tcl_GlobalEval	execute Tcl commands
Tcl_GlobalEvalObj	execute Tcl commands
Tcl_HashStats	procedures to manage hash tables
Tcl_HideCommand	manage multiple Tcl interpreters, aliases and hidden commands.
Tcl_IncrRefCount	manipulate Tcl objects
Tcl_InitHashTable	procedures to manage hash tables
Tcl_InputBlocked	buffered I/O facilities using channels
Tcl_InputBuffered	buffered I/O facilities using channels
Tcl_Interp	client-visible fields of interpreter structures
Tcl_InterpDeleted	create and delete Tcl command interpreters
Tcl_IsSafe	manage multiple Tcl interpreters, aliases and hidden commands.
Tcl_IsShared	manipulate Tcl objects
Tcl_JoinPath	manipulate platform-dependent file paths
Tcl_LinkVar	link Tcl variable to C variable
Tcl_ListObjAppendEle	manipulate Tcl objects as lists
Tcl_ListObjAppendLis	manipulate Tcl objects as lists
Tcl_ListObjGetElemen	manipulate Tcl objects as lists
Tcl_ListObjIndex	manipulate Tcl objects as lists
Tcl_ListObjLength	manipulate Tcl objects as lists
Tcl_ListObjReplace	manipulate Tcl objects as lists

Tcl_Main	main program for Tcl-based applications
Tcl_MakeFileChannel	buffered I/O facilities using channels
Tcl_MakeSafe	manage multiple Tcl interpreters, aliases and hidden commands.
Tcl_MakeTcpClientCha	procedures to open channels using TCP sockets
Tcl_Merge	manipulate Tcl lists
Tcl_NewBooleanObj	manipulate Tcl objects as boolean values
Tcl_NewDoubleObj	manipulate Tcl objects as floating-point values
Tcl_NewIntObj	manipulate Tcl objects as integers
Tcl_NewListObj	manipulate Tcl objects as lists
Tcl_NewLongObj	manipulate Tcl objects as integers
Tcl_NewObj	manipulate Tcl objects
Tcl_NewStringObj	manipulate Tcl objects as strings
Tcl_NextHashEntry	procedures to manage hash tables
Tcl_NotifyChannel	procedures for creating and manipulating channels
Tcl_ObjGetVar2	manipulate Tcl variables
Tcl_ObjSetVar2	manipulate Tcl variables
Tcl_OpenCommandChann	buffered I/O facilities using channels
Tcl_OpenFileChannel	buffered I/O facilities using channels
Tcl_OpenTcpClient	procedures to open channels using TCP sockets
Tcl_OpenTcpServer	procedures to open channels using TCP sockets
Tcl_PkgProvide	package version control
Tcl_PkgRequire	package version control
Tcl_PosixError	record information about errors
Tcl_Preserve	avoid freeing storage while it’s being used
Tcl_PrintDouble	Convert floating value to string
Tcl_QueueEvent	the event queue and notifier interfaces
Tcl_Read	buffered I/O facilities using channels
Tcl_Realloc	allocate or free heap memory
Tcl_ReapDetachedProc	manage child processes in background
Tcl_RecordAndEval	save command on history list before evaluating
Tcl_RecordAndEvalObj	save command on history list before evaluating
Tcl_RegExpCompile	Pattern matching with regular expressions
Tcl_RegExpExec	Pattern matching with regular expressions
Tcl_RegExpMatch	Pattern matching with regular expressions
Tcl_RegExpRange	Pattern matching with regular expressions
Tcl_RegisterChannel	buffered I/O facilities using channels
Tcl_RegisterObjType	manipulate Tcl object types
Tcl_Release	avoid freeing storage while it’s being used

Tcl_ResetResult	manipulate Tcl result
Tcl_ScanElement	manipulate Tcl lists
Tcl_Seek	buffered I/O facilities using channels
Tcl_ServiceAll	the event queue and notifier interfaces
Tcl_ServiceEvent	the event queue and notifier interfaces
Tcl_SetAssocData	manage associations of string keys and user specified data with Tcl interp
Tcl_SetBooleanObj	manipulate Tcl objects as boolean values
Tcl_SetChannelBuffer	procedures for creating and manipulating channels
Tcl_SetChannelOption	buffered I/O facilities using channels
Tcl_SetCommandInfo	implement new commands in C
Tcl_SetDefaultTransl	procedures for creating and manipulating channels
Tcl_SetDoubleObj	manipulate Tcl objects as floating-point values
Tcl_SetErrno	manipulate errno to store and retrieve error codes
Tcl_SetErrorCode	record information about errors
Tcl_SetHashValue	procedures to manage hash tables
Tcl_SetIntObj	manipulate Tcl objects as integers
Tcl_SetListObj	manipulate Tcl objects as lists
Tcl_SetLongObj	manipulate Tcl objects as integers
Tcl_SetMaxBlockTime	the event queue and notifier interfaces
Tcl_SetObjLength	manipulate Tcl objects as strings
Tcl_SetObjResult	manipulate Tcl result
Tcl_SetRecursionLimi	set maximum allowable nesting depth in interpreter
Tcl_SetResult	manipulate Tcl result
Tcl_SetServiceMode	the event queue and notifier interfaces
Tcl_SetStdChannel	procedures for retrieving and replacing the standard channels
Tcl_SetStringObj	manipulate Tcl objects as strings
Tcl_SetTimer	the event queue and notifier interfaces
Tcl_SetVar	manipulate Tcl variables
Tcl_SetVar2	manipulate Tcl variables
Tcl_Sleep	delay execution for a given number of milliseconds
Tcl_SplitList	manipulate Tcl lists
Tcl_SplitPath	manipulate platform-dependent file paths
Tcl_StaticPackage	make a statically linked package available via the Bload R command
Tcl_StringMatch	test whether a string matches a pattern
Tcl_Tell	buffered I/O facilities using channels
Tcl_TraceVar	monitor accesses to a variable
Tcl_TraceVar2	monitor accesses to a variable
Tcl_TranslateFileNam	convert file name to native form and replace tilde with home directory

Tcl_UnlinkVar	link Tcl variable to C variable
Tcl_UnregisterChanne	buffered I/O facilities using channels
Tcl_UnsetVar	manipulate Tcl variables
Tcl_UnsetVar2	manipulate Tcl variables
Tcl_UntraceVar	monitor accesses to a variable
Tcl_UntraceVar2	monitor accesses to a variable
Tcl_UpVar	link one variable to another
Tcl_UpVar2	link one variable to another
Tcl_UpdateLinkedVar	link Tcl variable to C variable
Tcl_VarEval	execute Tcl commands
Tcl_VarTraceInfo	monitor accesses to a variable
Tcl_VarTraceInfo2	monitor accesses to a variable
Tcl_WaitForEvent	the event queue and notifier interfaces
Tcl_Write	buffered I/O facilities using channels
Tcl_WrongNumArgs	generate standard error message for wrong number of arguments

`wish`	Simple windowing shell

`bell`	Ring a display's bell
`bind`	Arrange for X events to invoke Tcl scripts
`bindtags`	Determine which bindings apply to a window, and order of evaluation
`bitmap`	Images that display two colors
`button`	Create and manipulate button widgets
`canvas`	Create and manipulate canvas widgets
`checkbutton`	Create and manipulate checkbutton widgets
`clipboard`	Manipulate Tk clipboard
`destroy`	Destroy one or more windows
`entry`	Create and manipulate entry widgets
`event`	Miscellaneous event facilities
`focus`	Manage the input focus
`font`	Create and inspect fonts.
`frame`	Create and manipulate frame widgets
`grab`	Confine pointer and keyboard events to a window sub-tree
`grid`	Geometry manager that arranges widgets in a grid
`image`	Create and manipulate images
`label`	Create and manipulate label widgets
`listbox`	Create and manipulate listbox widgets
`loadTk`	Load Tk into a safe interpreter.
`lower`	Change a window's position in the stacking order
`menu`	Create and manipulate menu widgets
`menubutton`	Create and manipulate menubutton widgets
`message`	Create and manipulate message widgets
`option`	Add/retrieve window options to/from the option database
`options`	Standard options supported by widgets
`pack`	Obsolete syntax for packer geometry manager
`pack`	Geometry manager that packs around edges of cavity
`photo`	Full-color images
`place`	Geometry manager for fixed or rubber-sheet placement
`radiobutton`	Create and manipulate radiobutton widgets
`raise`	Change a window's position in the stacking order
`scale`	Create and manipulate scale widgets
`scrollbar`	Create and manipulate scrollbar widgets
`selection`	Manipulate the X selection
`send`	Execute a command in a different application
`text`	Create and manipulate text widgets
`tk`	Manipulate Tk internal state

tk_bindForTraversal	Obsolete support for menu bars
tk_bisque	Modify the Tk color palette
tk_chooseColor	pops up a dialog box for the user to select a color.
tk_dialog	Create modal dialog and wait for response
tk_focusFollowsMouse	Utility procedures for managing the input focus.
tk_focusNext	Utility procedures for managing the input focus.
tk_focusPrev	Utility procedures for managing the input focus.
tk_getOpenFile	pop up a dialog box for the user to select a file to open or save.
tk_getSaveFile	pop up a dialog box for the user to select a file to open or save.
tk_menuBar	Obsolete support for menu bars
tk_messageBox	pops up a message window and waits for user response.
tk_optionMenu	Create an option menubutton and its menu
tk_popup	Post a popup menu
tk_setPalette	Modify the Tk color palette
tkerror	Command invoked to process background errors
tkvars	Variables used or set by Tk
tkwait	Wait for variable to change or window to be destroyed
toplevel	Create and manipulate toplevel widgets
winfo	Return window-related information
wm	Communicate with window manager

Tk_3DBorderColor	draw borders with three-dimensional appearance
Tk_3DBorderGC	draw borders with three-dimensional appearance
Tk_3DHorizontalBevel	draw borders with three-dimensional appearance
Tk_3DVerticalBevel	draw borders with three-dimensional appearance
Tk_Attributes	retrieve information from Tk's local data structure
Tk_BindEvent	invoke scripts in response to X events
Tk_CanvasDrawableCoo	utility procedures for canvas type managers
Tk_CanvasEventuallyR	utility procedures for canvas type managers
Tk_CanvasGetCoord	utility procedures for canvas type managers
Tk_CanvasPsBitmap	utility procedures for generating Postscript for canvases
Tk_CanvasPsColor	utility procedures for generating Postscript for canvases
Tk_CanvasPsFont	utility procedures for generating Postscript for canvases
Tk_CanvasPsPath	utility procedures for generating Postscript for canvases
Tk_CanvasPsStipple	utility procedures for generating Postscript for canvases
Tk_CanvasPsY	utility procedures for generating Postscript for canvases
Tk_CanvasSetStippleO	utility procedures for canvas type managers
Tk_CanvasTagsOption	utility procedures for canvas type managers
Tk_CanvasTextInfo	additional information for managing text items in canvases
Tk_CanvasTkwin	utility procedures for canvas type managers
Tk_CanvasWindowCoord	utility procedures for canvas type managers
Tk_ChangeWindowAttri	change window configuration or attributes
Tk_Changes	retrieve information from Tk's local data structure
Tk_CharBbox	routines to measure and display single-font, multi-line, justified text.
Tk_Class	set or retrieve a window's class
Tk_ClearSelection	Deselect a selection
Tk_ClipboardAppend	Manage the clipboard
Tk_ClipboardClear	Manage the clipboard
Tk_Colormap	retrieve information from Tk's local data structure
Tk_ComputeTextLayout	routines to measure and display single-font, multi-line, justified text.
Tk_ConfigureInfo	process configuration options for widgets
Tk_ConfigureValue	process configuration options for widgets
Tk_ConfigureWidget	process configuration options for widgets
Tk_ConfigureWindow	change window configuration or attributes
Tk_CoordsToWindow	Find window containing a point
Tk_CreateBinding	invoke scripts in response to X events
Tk_CreateBindingTabl	invoke scripts in response to X events
Tk_CreateErrorHandle	handle X protocol errors
Tk_CreateEventHandle	associate procedure callback with an X event

Tk_CreateGenericHand	associate procedure callback with all X events
Tk_CreateImageType	define new kind of image
Tk_CreateItemType	define new kind of canvas item
Tk_CreatePhotoImageF	define new file format for photo images
Tk_CreateSelHandler	arrange to handle requests for a selection
Tk_CreateWindow	create or delete window
Tk_CreateWindowFromP	create or delete window
Tk_DefineBitmap	maintain database of single-plane pixmaps
Tk_DefineCursor	change window configuration or attributes
Tk_DeleteAllBindings	invoke scripts in response to X events
Tk_DeleteBinding	invoke scripts in response to X events
Tk_DeleteBindingTabl	invoke scripts in response to X events
Tk_DeleteErrorHandle	handle X protocol errors
Tk_DeleteEventHandle	associate procedure callback with an X event
Tk_DeleteGenericHand	associate procedure callback with all X events
Tk_DeleteImage	Destroy an image.
Tk_DeleteSelHandler	arrange to handle requests for a selection
Tk_Depth	retrieve information from Tk’s local data structure
Tk_DestroyWindow	create or delete window
Tk_Display	retrieve information from Tk’s local data structure
Tk_DisplayName	retrieve information from Tk’s local data structure
Tk_DistanceToTextLay	routines to measure and display single-font, multi-line, justified text.
Tk_Draw3DPolygon	draw borders with three-dimensional appearance
Tk_Draw3DRectangle	draw borders with three-dimensional appearance
Tk_DrawChars	routines to measure and display simple single-line strings.
Tk_DrawFocusHighligh	draw the traversal highlight ring for a widget
Tk_DrawTextLayout	routines to measure and display single-font, multi-line, justified text.
Tk_Fill3DPolygon	draw borders with three-dimensional appearance
Tk_Fill3DRectangle	draw borders with three-dimensional appearance
Tk_FindPhoto	manipulate the image data stored in a photo image.
Tk_FontId	accessor functions for fonts
Tk_FontMetrics	accessor functions for fonts
Tk_Free3DBorder	draw borders with three-dimensional appearance
Tk_FreeBitmap	maintain database of single-plane pixmaps
Tk_FreeColor	maintain database of colors
Tk_FreeColormap	allocate and free colormaps
Tk_FreeCursor	maintain database of cursors
Tk_FreeFont	maintain database of fonts

Tk_FreeGC	maintain database of read-only graphics contexts
Tk_FreeImage	use an image in a widget
Tk_FreeOptions	process configuration options for widgets
Tk_FreePixmap	allocate and free pixmaps
Tk_FreeTextLayout	routines to measure and display single-font, multi-line, justified text.
Tk_FreeXId	make X resource identifier available for reuse
Tk_GeometryRequest	specify desired geometry or internal border for a window
Tk_Get3DBorder	draw borders with three-dimensional appearance
Tk_GetAllBindings	invoke scripts in response to X events
Tk_GetAnchor	translate between strings and anchor positions
Tk_GetAtomName	manage cache of X atoms
Tk_GetBinding	invoke scripts in response to X events
Tk_GetBitmap	maintain database of single-plane pixmaps
Tk_GetBitmapFromData	maintain database of single-plane pixmaps
Tk_GetCapStyle	translate between strings and cap styles
Tk_GetColor	maintain database of colors
Tk_GetColorByValue	maintain database of colors
Tk_GetColormap	allocate and free colormaps
Tk_GetCursor	maintain database of cursors
Tk_GetCursorFromData	maintain database of cursors
Tk_GetFont	maintain database of fonts
Tk_GetGC	maintain database of read-only graphics contexts
Tk_GetImage	use an image in a widget
Tk_GetImageMasterDat	define new kind of image
Tk_GetItemTypes	define new kind of canvas item
Tk_GetJoinStyle	translate between strings and join styles
Tk_GetJustify	translate between strings and justification styles
Tk_GetOption	retrieve an option from the option database
Tk_GetPixels	translate between strings and screen units
Tk_GetPixmap	allocate and free pixmaps
Tk_GetRelief	translate between strings and relief values
Tk_GetRootCoords	Compute root-window coordinates of window
Tk_GetScreenMM	translate between strings and screen units
Tk_GetScrollInfo	parse arguments for scrolling commands
Tk_GetSelection	retrieve the contents of a selection
Tk_GetUid	convert from string to unique identifier
Tk_GetVRootGeometry	Get location and size of virtual root for window
Tk_GetVisual	translate from string to visual

Tk_HandleEvent	invoke event handlers for window system events
Tk_Height	retrieve information from Tk's local data structure
Tk_IdToWindow	Find Tk's window information for an X window
Tk_ImageChanged	notify widgets that image needs to be redrawn
Tk_Init	add Tk to an interpreter and make a new Tk application.
Tk_InternAtom	manage cache of X atoms
Tk_InternalBorderWid	retrieve information from Tk's local data structure
Tk_IntersectTextLayo	routines to measure and display single-font, multi-line, justified text.
Tk_IsMapped	retrieve information from Tk's local data structure
Tk_IsTopLevel	retrieve information from Tk's local data structure
Tk_Main	main program for Tk-based applications
Tk_MainLoop	loop for events until all windows are deleted
Tk_MainWindow	find the main window for an application
Tk_MaintainGeometry	maintain geometry of one window relative to another
Tk_MakeWindowExist	create or delete window
Tk_ManageGeometry	arrange to handle geometry requests for a window
Tk_MapWindow	map or unmap a window
Tk_MeasureChars	routines to measure and display simple single-line strings.
Tk_MoveResizeWindow	change window configuration or attributes
Tk_MoveToplevelWindo	Adjust the position of a top-level window
Tk_MoveWindow	change window configuration or attributes
Tk_Name	convert between names and window tokens
Tk_NameOf3DBorder	draw borders with three-dimensional appearance
Tk_NameOfAnchor	translate between strings and anchor positions
Tk_NameOfBitmap	maintain database of single-plane pixmaps
Tk_NameOfCapStyle	translate between strings and cap styles
Tk_NameOfColor	maintain database of colors
Tk_NameOfCursor	maintain database of cursors
Tk_NameOfFont	maintain database of fonts
Tk_NameOfImage	Return name of image.
Tk_NameOfJoinStyle	translate between strings and join styles
Tk_NameOfJustify	translate between strings and justification styles
Tk_NameOfRelief	translate between strings and relief values
Tk_NameToWindow	convert between names and window tokens
Tk_Offset	process configuration options for widgets
Tk_OwnSelection	make a window the owner of the primary selection
Tk_Parent	retrieve information from Tk's local data structure
Tk_ParseArgv	process command-line options

Tk_PathName	convert between names and window tokens
Tk_PhotoBlank	manipulate the image data stored in a photo image.
Tk_PhotoExpand	manipulate the image data stored in a photo image.
Tk_PhotoGetImage	manipulate the image data stored in a photo image.
Tk_PhotoGetSize	manipulate the image data stored in a photo image.
Tk_PhotoPutBlock	manipulate the image data stored in a photo image.
Tk_PhotoPutZoomedBlo	manipulate the image data stored in a photo image.
Tk_PhotoSetSize	manipulate the image data stored in a photo image.
Tk_PointToChar	routines to measure and display single-font, multi-line, justified text.
Tk_PostscriptFontNam	accessor functions for fonts
Tk_QueueWindowEvent	Add a window event to the Tcl event queue
Tk_RedrawImage	use an image in a widget
Tk_ReqHeight	retrieve information from Tk’s local data structure
Tk_ReqWidth	retrieve information from Tk’s local data structure
Tk_ResizeWindow	change window configuration or attributes
Tk_RestackWindow	Change a window’s position in the stacking order
Tk_RestrictEvents	filter and selectively delay X events
Tk_Screen	retrieve information from Tk’s local data structure
Tk_ScreenNumber	retrieve information from Tk’s local data structure
Tk_SetAppName	Set the name of an application for ‘‘send’’ commands
Tk_SetBackgroundFrom	draw borders with three-dimensional appearance
Tk_SetClass	set or retrieve a window’s class
Tk_SetGrid	control the grid for interactive resizing
Tk_SetInternalBorder	specify desired geometry or internal border for a window
Tk_SetWindowBackgrou	change window configuration or attributes
Tk_SetWindowBackgrou	change window configuration or attributes
Tk_SetWindowBorder	change window configuration or attributes
Tk_SetWindowBorderPi	change window configuration or attributes
Tk_SetWindowBorderWi	change window configuration or attributes
Tk_SetWindowColormap	change window configuration or attributes
Tk_SetWindowVisual	change visual characteristics of window
Tk_SizeOfBitmap	maintain database of single-plane pixmaps
Tk_SizeOfImage	use an image in a widget
Tk_StrictMotif	Return value of tk_strictMotif variable
Tk_TextLayoutToPosts	routines to measure and display single-font, multi-line, justified text.
Tk_TextWidth	routines to measure and display simple single-line strings.
Tk_Uid	convert from string to unique identifier
Tk_UndefineCursor	change window configuration or attributes

Tk_UnderlineChars	routines to measure and display simple single-line strings.
Tk_UnderlineTextLayo	routines to measure and display single-font, multi-line, justified text.
Tk_UnmaintainGeometr	maintain geometry of one window relative to another
Tk_UnmapWindow	map or unmap a window
Tk_UnsetGrid	control the grid for interactive resizing
Tk_Visual	retrieve information from Tk's local data structure
Tk_Width	retrieve information from Tk's local data structure
Tk_WindowId	retrieve information from Tk's local data structure
Tk_X	retrieve information from Tk's local data structure
Tk_Y	retrieve information from Tk's local data structure

buttonbox	Create and manipulate a manager widget for buttons
calendar	Create and manipulate a monthly calendar
canvasprintbox	Create and manipulate a canvas print box widget
canvasprintdialog	Create and manipulate a canvas print dialog widget
checkbox	Create and manipulate a checkbox widget
combobox	Create and manipulate combination box widgets
dateentry	Create and manipulate a dateentry widget
datefield	Create and manipulate a date field widget
dialog	Create and manipulate a dialog widget
dialogshell	Create and manipulate a dialog shell widget
disjointlistbox	Create and manipulate a disjointlistbox widget
entryfield	Create and manipulate a entry field widget
extfileselectionbox	Create and manipulate a file selection box widget
extfileselectiondial	Create and manipulate a file selection dialog widget
feedback	Create and manipulate a feedback widget to display feedback on the curr
fileselectionbox	Create and manipulate a file selection box widget
fileselectiondialog	Create and manipulate a file selection dialog widget
finddialog	Create and manipulate a find dialog widget
hierarchy	Create and manipulate a hierarchy widget
hyperhelp	Create and manipulate a hyperhelp widget
labeledframe	Create and manipulate a labeled frame widget
labeledwidget	Create and manipulate a labeled widget
mainwindow	Create and manipulate a mainwindow widget
menubar	Create and manipulate menubar menu widgets
messagebox	Create and manipulate a messagebox text widget
messagedialog	Create and manipulate a message dialog widget
notebook	create and manipulate notebook widgets
optionmenu	Create and manipulate a option menu widget
panedwindow	Create and manipulate a paned window widget
promptdialog	Create and manipulate a prompt dialog widget
pushbutton	Create and manipulate a push button widget
radiobox	Create and manipulate a radiobox widget
scopedobject	Create and manipulate a scoped [incr Tcl] class object.
scrolledcanvas	Create and manipulate scrolled canvas widgets
scrolledframe	Create and manipulate scrolled frame widgets
scrolledhtml	Create and manipulate a scrolled text widget with the capability of displa
scrolledlistbox	Create and manipulate scrolled listbox widgets
scrolledtext	Create and manipulate a scrolled text widget

`selectionbox`	Create and manipulate a selection box widget
`selectiondialog`	Create and manipulate a selection dialog widget
`shell`	Create and manipulate a shell widget
`spindate`	Create and manipulate time spinner widgets
`spinint`	Create and manipulate a integer spinner widget
`spinner`	Create and manipulate a spinner widget
`spintime`	Create and manipulate time spinner widgets
`tabnotebook`	create and manipulate tabnotebook widgets
`tabset`	create and manipulate tabs as as set
`timeentry`	Create and manipulate a timeentry widget
`timefield`	Create and manipulate a time field widget
`toolbar`	Create and manipulate a tool bar
`watch`	Create and manipulate time with a watch widgets

`body`	change the body for a class method/proc
`class`	create a class of objects
`code`	capture the namespace context for a code fragment
`configbody`	change the "config" code for a public variable
`delete`	delete things in the interpreter
`ensemble`	create or modify a composite command
`find`	search for classes and objects
`itcl`	object-oriented extensions to Tcl
`itcl_class`	create a class of objects (obsolete)
`itcl_info`	query info regarding classes and objects (obsolete)
`itclsh`	Simple shell for [incr Tcl]
`itclvars`	variables used by [incr Tcl]
`local`	create an object local to a procedure
`scope`	capture the namespace context for a variable

`Archetype`	base class for all [incr Tk] mega-widgets
`Toplevel`	base class for mega-widgets in a top-level window
`Widget`	base class for mega-widgets within a frame
`itk`	framework for building mega-widgets in Tcl/Tk
`itkvars`	variables used by [incr Tk]
`itkwish`	Simple windowing shell for [incr Tcl] / [incr Tk]
`usual`	access default option-handling commands .br for a mega-widget compon