Ck
A Curses Based Toolkit For Tcl

Christian Werner (Christian.Werner@t-online.de)

This manual is for Ck version 8.0

NAME

cwsh − Simple curses windowing shell

SYNOPSIS

cwsh ?*fileName arg arg ...*?

DESCRIPTION

Cwsh is a simple program consisting of the Tcl command language, the Ck toolkit, and a main program that eventually reads commands from a file. It creates a main window and then processes Tcl commands. If **cwsh** is invoked with no arguments, then it reads Tcl commands interactively from a command window. It will continue processing commands until all windows have been deleted or until the **exit** Tcl command is evaluated. If there exists a file **.cwshrc** in the home directory of the user, **cwsh** evaluates the file as a Tcl script just before presenting the command window.

If **cwsh** is invoked with an initial *fileName* argument, then *fileName* is treated as the name of a script file. **Cwsh** will evaluate the script in *fileName* (which presumably creates a user interface), then it will respond to events until all windows have been deleted. The command window will not be created. There is no automatic evaluation of **.cwshrc** in this case, but the script file can always **source** it if desired.

APPLICATION NAME AND CLASS

The name of the application, which is used for processing the option data base is taken from *fileName*, if it is specified, or from the command name by which **cwsh** was invoked. If this 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, is the same as its name except that the first letter is capitalized.

VARIABLES

Cwsh sets the following Tcl variables:

argc	Contains a count of the number of *arg* arguments (0 if none).
argv	Contains a Tcl list whose elements are the *arg* arguments that follow *fileName*, 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 **cwsh** was invoked.
tcl_interactive	Contains 1 if **cwsh** was started without *fileName* argument, 0 otherwise.

SCRIPT FILES

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

```
#!/usr/local/bin/cwsh
```

then you can invoke the script file directly from your shell if you mark it as executable. This assumes that **cwsh** 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 **cwsh** 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 cwsh \
exec cwsh "$0" "$@"
```

This approach has three advantages over the approach in the previous paragraph. First, the location of the **cwsh** 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 **cwsh** is itself a shell script (this is done on some systems in order to handle multiple architectures or operating systems: the **cwsh** script selects one of several binaries to run). The three lines cause both **sh** and **cwsh** 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 **cwsh** to reprocess the entire script. When **cwsh** 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.

KEYWORDS

shell, toolkit

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 ...*?

DESCRIPTION

This command is used to delay execution of the program or to execute a command in background after a delay. 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 keypresses or any other 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 a background event handler. 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 **tkerror** 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 handler (the script runs the next time the Tk 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 **tkerror** mechanism is used to report the error.

SEE ALSO

tkerror

KEYWORDS

cancel, delay, sleep, time

NAME

bell − Ring a terminal’s bell

SYNOPSIS

bell

DESCRIPTION

This command rings the bell on the terminal if supported, otherwise the terminal’s screen is flashed. An empty string is returned as result of this command. **Bell** is carried out immediately, i.e. not deferred until the application becomes idle.

KEYWORDS

beep, bell, ring

NAME

bind − Arrange for 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 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 either of two 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:

<type-detail>

The entire event pattern is surrounded by angle brackets. Inside the angle brackets are 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.

EVENT TYPES

The *type* field may be any of the following list. Where two names appear together, they are synonyms.

BarCode	**Expose**	**Map**
ButtonPress, Button	**FocusIn**	**Unmap**
ButtonRelease	**FocusOut**	
Destroy	**KeyPress, Key, Control**	

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**, **Key** or **Control**, then *detail* may be specified in the form of a 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 some of the non-ASCII keys on the keyboard (e.g. ‘‘F1’’ is the keysym for the F1 function key, if it exists). The complete list of keysyms is not presented here; it is available by invoking the **curses haskey** Tcl command 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, **<KeyPress-comma>** is equivalent to **<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.

%b The number of the button that was pressed or released. Valid only for **ButtonPress** and **ButtonRelease** events.

%k The *keycode* field from the event. Valid only for **KeyPress** and **KeyRelease** events.

%x The *x* coordinate (window coordinate system) from **ButtonPress** and **ButtonRelease** events.

%y The *y* coordinate (window coordinate system) from **ButtonPress** and **ButtonRelease** events.

%A For **KeyPress** events, 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). For **BarCode** events, substitutes the entire barcode data packet.

%K The keysym corresponding to the event, substituted as a textual string. Valid only for **KeyPress** events.

%N The keysym corresponding to the event, substituted as a decimal number. Valid only for **KeyPress** events.

%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* coordinate (screen coordinate system) from **ButtonPress** and **ButtonRelease** events.

%Y The *y* coordinate (screen coordinate system) from **ButtonPress** and **ButtonRelease** 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 **comma**; 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 event. If the bindings are associated with different *tag*'s, then each of the bindings will be executed, in order. By default, a class binding will be executed first, followed by a binding for the widget, 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) a longer sequence (in terms of number of events matched) is more specific than a shorter sequence; (b) an event pattern that specifies a specific button or key is more specific than one that doesn't.

If an event does not match any of the existing bindings, then the event is ignored. An unbound event is not considered to be an error.

ERRORS

If an error occurs in executing the script for a binding then the **tkerror** mechanism is used to report the error. The **tkerror** command will be executed at global level (outside the context of any Tcl procedure).

SEE ALSO

tkerror

KEYWORDS

event, binding

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**.

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

button − Create and manipulate button widgets

SYNOPSIS

button *pathName* ?*options*?

STANDARD OPTIONS

activeAttributes	**attributes**	**disabledForeground**	**textVariable**
activeBackground	**background**	**foreground**	**underline**
activeForeground	**disabledAttributes**	**takeFocus**	**underlineAttributes**
anchor	**disabledBackground**	**text**	**underlineForeground**

See the ‘‘options’’ manual entry for details on the standard options.

WIDGET-SPECIFIC OPTIONS

Name: **command**
Class: **Command**
Command-Line Switch: **−command**

Specifies a Tcl command to associate with the button. This command is typically invoked when mouse button 1 is pressed over the button window.

Name: **height**
Class: **Height**
Command-Line Switch: **−height**

Specifies a desired height for the button in screen lines. If this option isn't specified, the button's desired height is 1 line.

Name: **state**
Class: **State**
Command-Line Switch: **−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 input focus is in the button. In active state the button is displayed using the **activeAttributes**, **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 **disabledAttributes**, **disabledForeground** and **disabledBackground** options determine how the button is displayed.

Name: **width**
Class: **Width**
Command-Line Switch: **−width**

Specifies a desired width for the button in screen columns. If this option isn't specified, the button's desired width is computed from the size of the 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, attributes, and text. 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. One of the characters may optionally be underlined using the **underline**, **underlineAttributes**, and **underlineForeground** options. It can display itself in either of three different ways, according to the **state** option. When a user invokes the button (e.g.

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*. 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 **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

Ck automatically creates class bindings for buttons that give them the following default behavior:

[1] A button activates whenever it gets the input focus and deactivates whenever it loses the input focus.

[2] If mouse button 1 is pressed over a button, the button is invoked.

[3] When a button has the input focus, the space or return key cause 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

checkbutton − Create and manipulate checkbutton widgets

SYNOPSIS

checkbutton *pathName* ?*options*?

STANDARD OPTIONS

activeAttributes	**attributes**	**disabledForeground**	**textVariable**
activeBackground	**background**	**foreground**	**underline**
activeForeground	**disabledAttributes**	**takeFocus**	**underlineAttributes**
anchor	**disabledBackground**	**text**	**underlineForeground**

See the ‘‘options’’ manual entry for details on the standard options.

WIDGET-SPECIFIC OPTIONS

Name: **command**
Class: **Command**
Command-Line Switch: **−command**

Specifies a Tcl command to associate with the button. This command is typically invoked when mouse button 1 is pressed on the button window. The button’s global variable (**−variable** option) will be updated before the command is invoked.

Name: **height**
Class: **Height**
Command-Line Switch: **−height**

Specifies a desired height for the button in screen lines. If this option isn’t specified, the button’s desired height is 1 line.

Name: **offValue**
Class: **Value**
Command-Line Switch: **−offvalue**

Specifies value to store in the button’s associated variable whenever this button is deselected. Defaults to ‘‘0’’.

Name: **onValue**
Class: **Value**
Command-Line Switch: **−onvalue**

Specifies value to store in the button’s associated variable whenever this button is selected. Defaults to ‘‘1’’.

Name: **selectColor**
Class: **Background**
Command-Line Switch: **−selectcolor**

Specifies a background color to use when the button is selected. If **indicatorOn** is true then the color applicies to the indicator.

Name: **state**
Class: **State**
Command-Line Switch: **−state**

Specifies one of three states for the checkbutton: **normal**, **active**, or **disabled**. In normal state the checkbutton is displayed using the **attributes**, **foreground** and **background** options. The active state is used when the input focus is in the checkbutton. In active state the checkbutton is displayed using the **activeAttributes**, **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 **disabledAttributes**, **disabledForeground**, and **disabledBackground** options determine how the checkbutton is displayed.

Name: **variable**
Class: **Variable**
Command-Line Switch: **−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).

Name: **width**
Class: **Width**
Command-Line Switch: **−width**

Specifies a desired width for the button in screen columns. If this option isn't specified, the button's desired width is computed from the size of the 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 and a square called an *indicator*. One of the characters of the string may optionally be underlined using the **underline**, **underlineAttributes**, and **underlineForeground** options. 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, 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 drawn with a special color, and a Tcl variable associated with the checkbutton is set to a particular value (normally 1). If the checkbutton is not selected, then the indicator is drawn with no special color, and the associated variable is set to a different value (typically 0). 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. 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*. 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 **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

Ck automatically creates class bindings for checkbuttons that give them the following default behavior:

[1] A checkbutton activates whenever it gets the input focus and deactivates whenever it loses the input focus.

[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 or return keys cause the checkbutton to be invoked.

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

ck_dialog − Create dialog and wait for response

SYNOPSIS

ck_dialog *window title text string string ...*

DESCRIPTION

This procedure is part of the Ck 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's top line as title for the dialog.

text
: Message to appear in the top portion of the dialog box.

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, **ck_dialog** waits for the user to select one of the buttons either by clicking on the button with the mouse or by typing return or space to invoke the focus 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.

KEYWORDS

bitmap, dialog

NAME

ck_focusNext, ck_focusPrev − Utility procedures for managing the input focus.

SYNOPSIS

ck_focusNext *window*
ck_focusPrev *window*

DESCRIPTION

ck_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 **ck_focusNext** never returns a window in a different top-level from *window*.

After computing the next window, **ck_focusNext** examines the window’s **−takefocus** option to see whether it should be skipped. If so, **ck_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*.

ck_focusPrev is similar to **ck_focusNext** except that it returns the window just before *window* in the focus order.

KEYWORDS

focus, keyboard traversal, toplevel

NAME

curses − Retrieve/modify curses based information

SYNOPSIS

curses *option* ?*arg arg ...*?

DESCRIPTION

The **curses** command is used to retrieve or modify information which is related to the **curses(3)** library providing the input/output mechanisms used by Ck. It can take any of a number of different forms, depending on the *option* argument. The legal forms are:

curses barcode *startChar endChar ?timeout?*

Enables or modifies barcode reader support with delivery of **BarCode** events. *StartChar* and *endChar* are the start and end characters which delimit the barcode data packet without being delivered to the application. They must be specified as decimal numbers. The optional *timeout* argument is the maximum time between reception of start and end characters in millisecond for receiving the data packet; the default value is 1000.

curses barcode *?off?*

If *off* is present, barcode reader support is disabled. Otherwise, the current start/end characters and the timeout are returned as a list of three decimal numbers.

curses baudrate

Returns the baud rate of the terminal as decimal string.

curses encoding *?ISO8859|IBM437?*

Sets or returns the character encoding being or to be used for displaying text. This affects for example the output of the text widget for the character values 0x80..0x9f.

curses gchar *?charName? ?value?*

Sets or returns the mappings of ‘‘Alternate Character Set’’ characters used to display the arrows of scrollbars, the indicators for checkbuttons and radiobuttons etc. *CharName* must be a valid name of an ACS character (see list below), and *value* must be an integer, i.e. the value of the **curses(3)** character which shall be output for the ACS character. By default the **terminfo(5)** entry for the terminal provides these mappings and there’s rarely a need to modify them.

Ck name	**description**
ulcorner	upper left corner
urcorner	upper right corner
llcorner	lower left corner
lrcorner	lower right corner
rtee	tee pointing right
ltee	tee pointing left
btee	tee pointing up
ttee	tee pointing down
hline	horizontal line
vline	vertical line
plus	large plus or crossover
s1	scan line #1
s9	scan line #9
diamond	diamond
ckboard	checker board (stipple)
degree	degree symbol
plminus	plus/minus
bullet	bullet

larrow	arrow pointing left
rarrow	arrow pointing right
uarrow	arrow pointing up
darrow	arrow pointing down
board	board of squares
lantern	lantern symbol
block	solid square block

curses haskey *?keyName?*

If *keyName* is omitted this command returns a list of all valid symbolic names of keyboard keys. If *keyName* is given, a boolean is returned indicating if the terminal can generate that key.

curses purgeinput

Removes all characters typed so far from the keyboard input queue. This command should be used with great caution, since **xterm(1)** mouse events and barcode events are reported through the keyboard input queue as a character stream which can be interrupted by this command.

curses refreshdelay *?milliseconds?*

Sets or returns a time value which is used to limit the number of **curses(3)** screen updates. By default the delay is zero, which does not impose any limits. Setting the refresh delay to a positive number can be useful in environments where the terminal is connected via terminal servers or **rlogin(1)** sessions.

curses reversekludge *?boolean?*

Queries or modifies special code for treatment of the reverse video attribute in conjunction with colors. On some terminals (e.g. the infamous AT386 Interactive console), the reverse attribute overrides the colors in effect. If the special code is enabled, the reverse attribute is emulated by swapping the foreground and background colors.

curses screendump *fileName*

Dumps the current screen contents to the file *fileName* if the curses library supports the **scr_dmp(3)** function. Otherwise an error is reported. The screen dump file is per se not useful, since it contains some binary representation internal to curses. However, there may exist an external utility program which transforms the screen dump file to ASCII in order to print it on paper.

curses suspend

Takes appropriate actions for job control, such as saving **curses(3)** terminal state, sending the stop signal to the process and restoring the terminal state when the process is continued.

SEE ALSO

curses(3)

KEYWORDS

screen, terminal, curses

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 and the actions of the **exit** command are taken. The *window*s are destroyed in order, and if an error occurs in destroying a window the command aborts without destroying the remaining windows.

KEYWORDS

application, destroy, window

NAME

entry − Create and manipulate entry widgets

SYNOPSIS

entry *pathName* ?*options*?

STANDARD OPTIONS

attributes	**justify**	**selectForeground**	**xScrollCommand**
background	**selectAttributes**	**takeFocus**	
foreground	**selectBackground**	**textVariable**	

See the ‘‘options’’ manual entry for details on the standard options.

WIDGET-SPECIFIC OPTIONS

Name: **show**
Class: **Show**
Command-Line Switch: **−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.

Name: **state**
Class: **State**
Command-Line Switch: **−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.

Name: **width**
Class: **Width**
Command-Line Switch: **−width**

Specifies an integer value indicating the desired width of the entry window, in screen columns. If the value is less than or equal to zero, the widget picks a size just large enough to hold its current text. The default width is 16.

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 and attributes. 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. Entries also observe the standard Ck 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).

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 **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*. 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 **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

Ck automatically creates class bindings for entries that give them the following default behavior.

[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.

[2] If any normal printing characters are typed in an entry, they are inserted at the point of the insertion cursor.

[3] 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. Control-b and Control-f behave the same as Left and Right, respectively.

[4] The Home key, or Control-a, will move the insertion cursor to the beginning of the entry and clear any selection in the entry.

[5] The End key, or Control-e, will move the insertion cursor to the end of the entry and clear any selection in the entry.

[6] The Select key sets the selection anchor to the position of the insertion cursor. It doesn't affect the current selection.

[7] 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.

[8] 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.

[9] Control-d deletes the character to the right of the insertion cursor.

[10] Control-k deletes all the characters to the right of the insertion cursor.

[11] 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

entryx − Create extended entry widgets

SYNOPSIS

entryx *pathName ?options?*

DESCRIPTION

This procedure is part of the Ck script library. It is a slightly extended version of the Tcl **entry** command which provides the following additional command line options:

−default *value*
: Default value for **integer**, **unsigned**, and **float** modes, which is stored in entry widget on FocusOut, if the value in the widget is not a legal number. Defaults to an empty string.

−fieldwidth *number*
: Limits the string in the entry widget to at most *number* characters. Defaults to some ten thousand characters.

−initial *value*
: Initial value for the entry's string. If this option is omitted, no initial value is set.

−mode *modeName*
: Determines the additional bindings for input checking which will be bound to the entry widget. *ModeName* must be one of **integer** (for integer numbers including optional sign), **unsigned** (for integer numbers without sign), **float** (for floating point numbers including optional sign and fractional part, but without exponent part), **normal** (for entries without input checking at all), **regexp** (for checking the entry's string against a regular expression), and **boolean** (for boolean values, e.g. 0 or 1, Y or N and so on). If the **−mode** option is omitted, **normal** is chosen as default.

−offvalue *char*
: For mode **boolean** entry widgets only: *char* is used for the ''false'' state of the entry. *Char* defaults to ''0''. It is always converted to upper case.

−onvalue *char*
: For mode **boolean** entry widgets only: *char* is used for the ''true'' state of the entry. *Char* defaults to ''1''. It is always converted to upper case.

−regexp *regExp*
: For all modes this is the regular expression which provides input filtering. This option is ignored for **boolean** mode.

−touchvariable *varName*
: The global variable *varName* is set to 1 whenever the user changes the entry's string. The user may reset this variable to 0 at any time.

These options must be given at creation time of the entry. They cannot be modified later using the **configure** widget command.

After creating the entry widget, **entryx** binds procedures to do input checking using the **bindtags** mechanism to the entry widget. These procedures provide for overtype rather than insert mode and give the following behaviour:

[1] If mouse button 1 is pressed on the entry and the entry accepts the input focus, the input focus is set on the entry and the entry's insertion cursor is placed on the very first character.

[2] The Left and Right keys move the insertion cursor one character to the left or right. In **boolean** mode these keys are used for keyboard traversal, i.e. the Left key moves the focus to the previous widget in focus order, the Right key to the next widget.

[3] The return key moves the input focus to the next widget in focus order.

[4] The Home key moves the insertion cursor to the beginning of the entry. In **boolean** mode this key is ignored.

[5] The End key moves the insertion cursor to the end of the entry. In **boolean** mode this key is ignored.

[6] The Delete key deletes the character to the right of the insertion cursor. In **boolean** mode this key is ignored.

[7] The BackSpace key and Control-h delete the character to the left of the insertion cursor. In **boolean** mode this key is ignored.

[8] The space key deletes from the insertion cursor until the end of the entry, if the mode is **integer**, **unsigned**, **float** or **regexp**. For **regexp** mode, the space character must not be part of the regular expression to achieve this behaviour. Otherwise it is treated as all other printable keys. In **boolean** mode this key toggles the entry's value.

[9] All other printable keys are checked according to the entry's mode. If allowed they overtype the character under the insertion cursor, otherwise they are ignored and the terminal's bell is rung. Lower case characters are automatically converted to upper case, if the regular expression filters denies lower case characters but allows upper case characters.

[10] FocusIn is bound to display the entry with the *reverse* attribute for monochrome screens or with swapped foreground and background colors on color screens; additionally, the insertion cursor is placed on the very first character in the entry.

[11] FocusOut is bound to restore the visual effects of FocusIn, i.e. on mononochrome screens, the *reverse* attribute is removed, on color screens, the foreground and background colors are restored to their original values. For **integer**, **unsigned**, and *float* modes, the entry's value is finally checked using the **scan** Tcl command. If the value is legal it is restored into the entry as the return from the **scan**, thus giving the Tcl canonical form for the value, i.e. no leading zeros for integral values (which otherwise could be interpreted as octal numbers) and a decimal point with at least one fractional digit for floating point values (which otherwise could be interpreted as integral numbers). If the **scan** conversion fails, the value specified in the **−default** option is stored into the entry.

KEYWORDS

entry, input

NAME

exit − Exit the process

SYNOPSIS

exit ?−*noclear*? ?*returnCode*?

DESCRIPTION

Terminate the process, returning *returnCode* (an integer) to the system as the exit status. If *returnCode* isn't specified then it defaults to 0. This command replaces the Tcl command by the same name. It is identical to Tcl's **exit** command except that before exiting it destroys all the windows managed by the process. This allows various cleanup operations to be performed, such as restoring the terminal's state and clearing the terminal's screen. If the −*noclear* switch is given, no screen clear takes place.

KEYWORDS

exit, process

NAME

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

SYNOPSIS

fileevent *fileId* **readable** ?*script*?
fileevent *fileId* **writable** ?*script*?

DESCRIPTION

This command is used to create *file event handlers*. A file event handler is a binding between a file and a script, such that the script is evaluated whenever the file becomes readable or writable. File event handlers are most commonly used to allow data to be received from a child 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** 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 *fileId* argument to **fileevent** refers to an open file; it must be **stdin**, **stdout**, **stderr**, or the return value from some previous **open** command. If the *script* argument is specified, then **fileevent** creates a new event handler: *script* will be evaluated whenever the file 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. If **fileevent** is called when the specified handler already exists, the new script replaces the old one.

If the *script* argument is not specified, **fileevent** returns the current script for *fileId*, 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 file is closed or its interpreter is deleted.

A file is considered to be readable whenever the **gets** and **read** commands can return without blocking. A file is also considered to be readable if an end-of-file or error condition is present. 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.

When using **fileevent** for event-driven I/O, it's important to read the file in the same units that are written from the other end. For example, suppose that you are using **fileevent** to read data generated by a child process. If the child process is writing whole lines, then you should use **gets** to read those lines. If the child generates one line at a time then you shouldn't make more than a single call to **gets** in *script*: the first call will consume all the available data, so the second call may block. You can also use **read** to read the child's data, but only if you know how many bytes the child is writing at a time: if you try to read more bytes than the child has written, the **read** call will block.

A file is considered to be writable if at least one byte of data can be written to the file without blocking, or if an error condition is present. Write handlers are probably not very useful without additional command support. The **puts** command is dangerous since it write more than one byte at a time and may thus block. What is really needed is a new non-blocking form of write that saves any data that couldn't be written to the file.

The script for a file event is executed at global level (outside the context of any Tcl procedure). If an error occurs while executing the script then the **tkerror** 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

tkerror

KEYWORDS

asynchronous I/O, event handler, file, readable, script, writable

NAME

focus − Manage the input focus

SYNOPSIS

focus
focus *window*

DESCRIPTION

The **focus** command is used to manage the Ck input focus. At any given time, one window on the terminal's screen is designated as the *focus window*; any key press events are sent to that window. The Tcl procedures **ck_focusNext** and **ck_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. Switching the focus among different top-levels is up to the user.

The **focus** command can take any of the following forms:

focus Returns the path name of the focus window or an empty string if no window in the application has the focus.

focus *window*

This command sets the input focus to *window* and returns an empty string. If *window* is in a different top-level than the current input focus window, then *window's* top-level is automatically raised just as if the **raise** Tcl command had been invoked. If *window* is an empty string then the command does nothing.

KEYWORDS

events, focus, keyboard, top-level

NAME

frame − Create and manipulate frame widgets

SYNOPSIS

frame *pathName ?options*?

STANDARD OPTIONS

attributes **border** **foreground** **takefocus**
background

See the ‘‘options’’ manual entry for details on the standard options.

WIDGET-SPECIFIC OPTIONS

Name: **class**
Class: **Class**
Command-Line Switch: **−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.

Name: **height**
Class: **Height**
Command-Line Switch: **−height**

Specifies the desired height for the window in screen lines. If this option is equal to zero then the window will not request any size at all.

Name: **width**
Class: **Width**
Command-Line Switch: **−width**

Specifies the desired width for the window in screen columns. If this option is 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 attributes. 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, attributes and border.

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*. 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

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*

The bounding box (in rows or columns) is returned for the space occupied by the grid position indicated by *column* and *row*. The return value consists of 4 integers. The first two are the column/row offset from the master window (x then y) of the top-left corner of the grid cell, and the second two are the width and height of the cell.

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** and **−weight**. The **−minsize** option sets the minimum column size, and the **−weight** option (a floating point value) sets the relative weight for apportioning any extra spaces among columns. If no value is specified, the current value is returned.

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 **−**.

−ipadx *amount*

The *amount* specifies how much horizontal internal padding to leave on each side of the slave(s). *Amount* is specified in terminal columns. 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). *Amount* is specified in terminal rows. It defaults to 0.

−padx *amount*

The *amount* specifies how much horizontal external padding to leave on each side of the slave(s). The *amount* defaults to 0.

−pady *amount*
The *amount* specifies how much vertical external padding to leave on the top and bottom of the slave(s). The *amount* defaults to 0.

−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 parcel is larger than its requested dimensions, this option may be used to position (or stretch) the slave within its cavity. *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.

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**.

grid location *master x y*
Given *x* and *y* values in terminal columns/rows 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** and **−weight**. **Minsize** sets the minimum row size, in screen units, and **weight** sets the relative weight for apportioning any extra spaces among rows. If no value is specified, the current value is returned.

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** or **weight**.

grid slaves *master* ?−*option value*?

If no options are supplied, a list of all of the slaves in *master* are returned. *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. S − 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.

GEOMETRY PROPAGATION

Grid 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 be 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.

CREDITS

The **grid** command is based on the *GridBag* geometry manager written by D. Stein.

KEYWORDS

geometry manager, location, grid, parcel, propagation, size, pack

NAME

label − Create and manipulate label widgets

SYNOPSIS

label *pathName* ?*options*?

STANDARD OPTIONS

anchor	**foreground**	**textVariable**	**underlineForeground**
attributes	**takeFocus**	**underline**	
background	**text**	**underlineAttributes**	

See the ''options'' manual entry for details on the standard options.

WIDGET-SPECIFIC OPTIONS

Name: **height**
Class: **Height**
Command-Line Switch: **−height**

Specifies a desired height for the label in screen lines. If this option isn't specified, the label's desired height is 1 line.

Name: **width**
Class: **Width**
Command-Line Switch: **−width**

Specifies a desired width for the label in screen columns. If this option isn't specified, the label's desired width is computed from the size of the 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. The label can be manipulated in a few simple ways, such as changing its attributes 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*. 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

activeAttributes	**attributes**	**selectAttributes**	**takeFocus**
activeBackground	**background**	**selectBackground**	**xScrollCommand**
activeForeground	**foreground**	**selectForeground**	**yScrollCommand**

See the ‘‘options’’ manual entry for details on the standard options.

WIDGET-SPECIFIC OPTIONS

Name: **height**
Class: **Height**
Command-Line Switch: **−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.

Name: **selectMode**
Class: **SelectMode**
Command-Line Switch: **−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** or **multiple**; the default value is **browse**.

Name: **width**
Class: **Width**
Command-Line Switch: **−width**

Specifies the desired width for the window in characters. 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, attributes and text. 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.

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.

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 the **activeAttributes**, **activeBackground**, and **activeForeground** options if the keyboard focus is in the listbox. The element 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 screen 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*. The active element is drawn with the **activeAttributes**, **activeBackground**, and **activeForeground** options when the widget has the input focus, and its index may be retrieved with the index **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 **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*. 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*. 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 a decimal string giving the integer index value that corresponds to *index*.

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 **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*. 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

Ck automatically creates class bindings for listboxes. Much of the behavior of a listbox is determined by its **selectMode** option, which selects one of three 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.

If the selection mode is **multiple**, any number of elements may be selected at once, including discontiguous ranges. Clicking button 1 on an element toggles its selection state without affecting any other elements.

Most people will probably want to use **browse** mode for single selections and **multiple** mode for multiple selections.

In addition to the above behavior, the following additional behavior is defined by the default bindings:

[1] 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** then the new active element is also selected and all other elements are deselected.

[2] The Left and Right keys scroll the listbox view left and right by the one column.

[3] The Prior and Next keys scroll the listbox view up and down by one page (the height of the window).

[4] The Home and End keys scroll the listbox horizontally to the left and right edges, respectively.

[5] 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.

The behavior of listboxes can be changed by defining new bindings for individual widgets or by redefining the class bindings.

KEYWORDS

listbox, widget

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*.

KEYWORDS

lower, obscure, stacking order

NAME

menu − Create and manipulate menu widgets

SYNOPSIS

menu *pathName* ?*options*?

STANDARD OPTIONS

activeAttributes	**background**	**disabledForeground**	**underlineForeground**
activeBackground	**border**	**foreground**	
activeForeground	**disabledAttributes**	**takeFocus**	
attributes	**disabledBackground**	**underlineAttributes**	

See the ''options'' manual entry for details on the standard options.

WIDGET-SPECIFIC OPTIONS

Name: **postCommand**
Class: **Command**
Command-Line Switch: **−postcommand**

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.

Name: **selectColor**
Class: **Background**
Command-Line Switch: **−selectcolor**

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. On color terminals this defaults to red, on monochrome terminals to white.

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 a column. 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. 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 input focus is over the entry. If a mouse button is pressed 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 other 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.

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 or menu entry. 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 or menu entry. 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 to the right 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. 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.

If a **−command** option is specified for a cascade entry then it is evaluated as a Tcl command whenever the entry is invoked.

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:

−activeattributes *value*

Specifies video attributes to use for displaying this entry when it is active. If this option is specified as an empty string (the default), then the **activeAttributes** option for the overall menu is used. This option is not available for separator entries.

−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. This option is not available for separator 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 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 entries.

−attributes *value*
Specifies video attributes 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 **attributes** option for the overall menu is used. This option is not available for separator 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 entries.

−command *value*
For command, checkbutton, and radiobutton entries, specifies a Tcl command to execute when the menu entry is invoked. For cascade entries, specifies a Tcl command to execute when the entry is activated (i.e. just before its submenu is posted). Not available for separator 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 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 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.

−state *value*
Specifies one of three states for the entry: **normal**, **active**, or **disabled**. In normal state the entry is displayed using the **attributes**, **foreground**, and **background** options for the entry or for the menu. The active state is typically used when the input focus is in the entry. In active state the entry is displayed using the **activeAttributes**,

activeForeground, and **activeBackground** options for the entry or for the menu. 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 **disabledAttributes**, **disabledForeground**, and **disabledBackground** options for the menu. 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. This option is not available for separator entries.

−**underlineAttributes** *value*
Specifies video attributes to use for displaying the underlined character in 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 **underlineAttributes** option for the overall menu is used. This option is not available for separator entries.

−**underlineForeground** *value*
Specifies a foreground color to use for displaying the underlined character in 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 **underlineForeground** option for the overall menu is used. This option is not available for separator entries.

−**value** *value*
Available only for radiobutton entries. Specifies the value to store in the entry's associated variable when the entry is selected.

−**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 **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*. 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*.

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*.

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**.

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.

pathName **yposition** *index*
Returns a decimal string giving the y-coordinate within the menu window of the line in the entry specified by *index*.

MENU CONFIGURATIONS

The default bindings support two different ways of using menus:

Pulldown Menus

This is the most common case. 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.

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 **ck_optionMenu** procedure to create option menubuttons and their menus.

DEFAULT BINDINGS

Ck automatically creates class bindings for menus that give them the following default behavior:

[1] When button 1 is pressed on a menu, the active entry (if any) is invoked. The menu also unposts.

[2] The Space and Return keys invoke the active entry and unpost the menu.

[3] 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.

[4] The Escape key aborts a menu selection in progress without invoking any entry. It also unposts the menu.

[5] 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.

[6] 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: Ck assumes that the lowest menubutton (which by default is the first one created) is on the left.

[7] 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

menubutton − Create and manipulate menubutton widgets

SYNOPSIS

menubutton *pathName* ?*options*?

STANDARD OPTIONS

activeAttributes	**attributes**	**disabledForeground**	**textVariable**
activeBackground	**background**	**foreground**	**underline**
activeForeground	**disabledAttributes**	**takeFocus**	**underlineAttributes**
anchor	**disabledBackground**	**text**	**underlineForeground**

See the ‘‘options’’ manual entry for details on the standard options.

WIDGET-SPECIFIC OPTIONS

Name: **height**
Class: **Height**
Command-Line Switch: **−height**

Specifies a desired height for the menubutton in screen lines. If this option isn’t specified, the menubutton’s desired height is 1 line.

Name: **indicatorForeground**
Class: **IndicatorForeground**
Command-Line Switch: **−indicatorforeground**

Color in which the indicator rectangle, if any, is drawn. On color terminals this defaults to red, on monochrome terminals to white.

Name: **indicatorOn**
Class: **IndicatorOn**
Command-Line Switch: **−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.

Name: **menu**
Class: **MenuName**
Command-Line Switch: **−menu**

Specifies the path name of the menu associated with this menubutton. The menu must be a child of the menubutton.

Name: **state**
Class: **State**
Command-Line Switch: **−state**

Specifies one of three states for the menubutton: **normal**, **active**, or **disabled**. In normal state the menubutton is displayed using the **attributes**, **foreground**, and **background** options. The active state is typically used when the input focus is in the menubutton. In active state the menubutton is displayed using the **activeAttributes**, **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 **disabledAttributes**, **disabledForeground**, and **disabledBackground** options determine how the button is displayed.

Name: **width**
Class: **Width**
Command-Line Switch: **−width**

Specifies a desired width for the menubutton in screen columns. If this option isn’t specified, the menubutton’s desired width is computed from the size of the 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, attributes, and text. 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 and is associated with a menu widget. One of the characters may optionally be underlined using the **underline**, **underlineAttributes**, and **underlineForeground** options. In normal usage, pressing mouse button 1 over the menubutton causes the associated menu to be posted just underneath the menubutton.

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*. 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

Ck automatically creates class bindings for menubuttons that give them the following default behavior:

[1] A menubutton activates whenever it gets the input focus and deactivates whenever it loses the input focus.

[2] Pressing mouse button 1 over a menubutton posts the menubutton: its associated menu is posted under the menubutton. Once a menu entry has been invoked, the menubutton unposts itself.

[3] 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.

[4] The F10 key may be typed in any window to post the first menubutton under its toplevel window that isn't disabled.

[5] 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	**background**	**takeFocus**	**textVariable**
attributes	**foreground**	**text**	

See the ''options'' manual entry for details on the standard options.

WIDGET-SPECIFIC OPTIONS

Name: **aspect**
Class: **Aspect**
Command-Line Switch: **−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 320.

Name: **justify**
Class: **Justify**
Command-Line Switch: **−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**, 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** option determines 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; the entire text block will be centered in the vertical span of the window.

Name: **width**
Class: **Width**
Command-Line Switch: **−width**

Specifies the length of lines in the window in screen columns. 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 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, attributes, and text. 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.

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*. 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

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 Ck 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 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. 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

options − Standard options supported by widgets

DESCRIPTION

This manual entry describes the common configuration options supported by widgets in the Ck 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, ''Name'' refers to the option's name in the option database ''Class'' refers to the option's class value in the option database. ''Command-Line Switch'' refers to the switch used in widget-creation 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.

Name: **activeAttributes**
Class: **Attributes**
Command-Line Switch: **−activeattributes**

Specifies video attributes to use when drawing active elements of widgets. This option must be a proper Tcl list which may contain the elements:

blink	**reverse**
bold	**standout**
dim	**underline**
normal	

If the list is empty, the **normal** attribute is automatically present.

Name: **activeBackground**
Class: **Foreground**
Command-Line Switch: **−activebackground**

Specifies background color to use when drawing active elements of widgets. Color specifications are always symbolic; valid color names are:

black	**magenta**
blue	**red**
cyan	**yellow**
green	**white**

Name: **activeForeground**
Class: **Background**
Command-Line Switch: **−activeforeground**

Specifies foreground color to use when drawing active elements. See above for possible colors.

Name: **anchor**
Class: **Anchor**
Command-Line Switch: **−anchor**

Specifies how the text in a widget 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 text such that its top-left corner is at the top-left corner of the widget.

Name: **attributes**
Class: **Attributes**
Command-Line Switch: **−attributes**

Specifies video attributes to use when displaying the widget. See **activeAttributes** for possible values.

Name: **background**
Class: **Background**
Command-Line Switch: **−background or −bg**

Specifies the normal background color to use when displaying the widget. See **activeBackground** for possible colors.

Name: **border**
Class: **Border**
Command-Line Switch: **−border**

Specifies the characters used for drawing a border around a widget. This options must be a proper Tcl list with exactly zero, one, three, six, or eight elements:

0 elements — No extra space for the border is allocated by the widget.

1 element — All four sides of the border's rectangle plus the corners are made from the sole element.

3 elements — The first element is used for the rectangle's corners, the second for the horizontal sides, and the third for the vertical sides.

6 elements — The order of elements in the rectangle is: upper left corner, horizontal side, upper right corner, vertical side, lower right corner, lower left corner.

8 elements — Each element gives corner and side, alternating, starting at the upper left corner of the square, clockwise.

The list elements must be either a single character or a symbolic name of a graphical character. For valid names of graphical characters refer to the **curses gchar** command.

Name: **disabledAttributes**
Class: **DisabledAttributes**
Command-Line Switch: **−disabledattributes**

Specifies video attributes to use when drawing a disabled element. See **activeAttributes** for possible values.

Name: **disabledBackground**
Class: **DisabledBackground**
Command-Line Switch: **−disabledbackground**

Specifies background color to use when drawing a disabled element. See **activeBackground** for possible colors.

Name: **disabledForeground**
Class: **DisabledForeground**
Command-Line Switch: **−disabledforeground**

Specifies foreground color to use when drawing a disabled element. See **activeBackground** for possible colors.

Name: **foreground**
Class: **Foreground**
Command-Line Switch: **−foreground or −fg**

Specifies the normal foreground color to use when displaying the widget. See **activeBackground** for possible colors.

Name: **justify**
Class: **Justify**
Command-Line Switch: **−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.

Name: **orient**
Class: **Orient**
Command-Line Switch: **−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.

Name: **selectAttributes**
Class: **SelectAttributes**
Command-Line Switch: **−selectattributes**

Specifies video attributes to use when displaying selected items. See **activeAttributes** for possible values.

Name: **selectBackground**
Class: **Foreground**
Command-Line Switch: **−selectbackground**

Specifies the background color to use when displaying selected items. See **activeBackground** for possible colors.

Name: **selectForeground**
Class: **Background**
Command-Line Switch: **−selectforeground**

Specifies the foreground color to use when displaying selected items. See **activeBackground** for possible colors.

Name: **takeFocus**
Class: **TakeFocus**
Command-Line Switch: **−takefocus**

Provides information used when moving the focus from window to window via keyboard traversal (e.g., Tab and BackTab). Before setting the focus to a window, the traversal scripts first check whether the window is viewable (it and all its ancestors are mapped); if not, the window is skipped. Next, the scripts consult the value of the **takeFocus** option. A value of **0** means that this window should be skipped entirely during keyboard traversal. **1** means that the this window should always receive the input focus. An empty value 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 or if it has no key bindings. 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; this value is used just as if the option had that value in the first place. 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.

Name: **text**

Class: **Text**
Command-Line Switch: **−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**.

Name: **textVariable**
Class: **Variable**
Command-Line Switch: **−textvariable**

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**.

Name: **underline**
Class: **Underline**
Command-Line Switch: **−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.

Name: **underlineAttributes**
Class: **UnderlineAttributes**
Command-Line Switch: **−underlineattributes**

Name: **underlineForeground**
Class: **UnderlineForeground**
Command-Line Switch: **−underlineforeground**

Specifies the foreground color to use when displaying an underlined character. See **activeBackground** for possible colors.

Name: **xScrollCommand**
Class: **ScrollCommand**
Command-Line Switch: **−xscrollcommand**

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.

Name: **yScrollCommand**
Class: **ScrollCommand**
Command-Line Switch: **−yscrollcommand**

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 − 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.

−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 **−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 be 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.

KEYWORDS

geometry manager, location, packer, parcel, propagation, size

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 Ck. 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:

−x *location*

Location specifies the x-coordinate within the master window of the anchor point for *window*. The location is specified in screen columns 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 columns 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 lines 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 lines 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 columns. 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 columns wider than the master.

−height *size*

Size specifies the height for *window* in screen lines. 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 lines 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 **ignore**, borders are ignored: the area of the master is considered to be its official area, which includes any internal border.

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

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.

The **place** command is the only way to position toplevel windows on the screen. In this special case, the master of a toplevel window is assumed to be the entire screen area and the toplevel's location and area is computed based on the screen's area.

KEYWORDS

geometry manager, height, location, master, place, rubber sheet, slave, width, toplevel

NAME

radiobutton − Create and manipulate radiobutton widgets

SYNOPSIS

radiobutton *pathName* ?*options*?

STANDARD OPTIONS

activeAttributes	**attributes**	**disabledForeground**	**textVariable**
activeBackground	**background**	**foreground**	**underline**
activeForeground	**disabledAttributes**	**takeFocus**	**underlineAttributes**
anchor	**disabledBackground**	**text**	**underlineForeground**

See the ‘‘options’’ manual entry for details on the standard options.

WIDGET-SPECIFIC OPTIONS

Name: **command**
Class: **Command**
Command-Line Switch: **−command**

Specifies a Tcl command to associate with the button. This command is typically invoked when mouse button 1 is pressed in the button window. The button’s global variable (**−variable** option) will be updated before the command is invoked.

Name: **height**
Class: **Height**
Command-Line Switch: **−height**

Specifies a desired height for the button in screen lines. If this option isn’t specified, the button’s desired height is 1 line.

Name: **selectColor**
Class: **Background**
Command-Line Switch: **−selectcolor**

Specifies a background color to use when the button is selected. If **indicatorOn** is true, the color applicies to the indicator.

Name: **state**
Class: **State**
Command-Line Switch: **−state**

Specifies one of three states for the radiobutton: **normal**, **active**, or **disabled**. In normal state the radiobutton is displayed using the **attributes**, **foreground**, and **background** options. The active state is used when the input focus is in the radiobutton. In active state the radiobutton is displayed using the **activeAttributes**, **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 **disabledAttributes**, **disabledForeground** and **disabledBackground** options determine how the radiobutton is displayed.

Name: **value**
Class: **Value**
Command-Line Switch: **−value**

Specifies value to store in the button’s associated variable whenever this button is selected.

Name: **variable**
Class: **Variable**
Command-Line Switch: **−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**.

Name: **width**
Class: **Width**
Command-Line Switch: **−width**

Specifies a desired width for the button in screen columns. If this option isn't specified, the button's desired width is computed from the size of the 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, attributes, and text. 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 and a circle called an *indicator*. One of the characters of the string may optionally be underlined using the **underline**, **underlineAttributes**, and **underlineForeground** options. 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, 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 special color, and a Tcl variable associated with the radiobutton is set to a particular value. If the radiobutton is not selected, the indicator is drawn with no special color. 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. By default a radio button 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*. 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 **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

Ck automatically creates class bindings for radiobuttons that give them the following default behavior:

[1] The radiobutton activates whenever it gets the input focus and deactivates whenever it loses the input focus.

[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 or return keys cause 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*.

KEYWORDS

obscure, raise, stacking order

NAME

recorder − Simple event recorder/player

SYNOPSIS

recorder replay *fileName*
recorder start ?−*withdelay*? *fileName*
recorder stop

DESCRIPTION

This command provides a simple recorder/player for certain kinds of events. The **recorder start** form arranges for recording events to the event log file *fileName*. If the −*withdelay* switch is specified, the delays between events are also recorded. The event log file may be replayed using the **recorder replay** command form. With **recorder stop** all recording/playing activity is stopped and all event log files are closed.

Each event takes up one line in an event log file. Event types are the first word in angle brackets in the line. They are followed by parameters for the event:

<ButtonPress> *window button x y rootX rooty*
Mouse button **button** (1, 2, or 3) pressed in window *window* at window coordinate *x, y*. Root coordinates are in *rootX*, *rootY*.

<ButtonRelease> *window button x y rootX rooty*
Mouse button released, analogous to **<ButtonPress>**.

<Delay> *milliseconds*
Delay replay for *milliseconds*.

<Key> *window keysym*
Key pressed in *window*. *Keysym* is the symbolic name of the key, e.g. ‘‘Linefeed’’, ‘‘Return’’, ‘‘Control-A’’, or a hexadecimal key code like 0xc3. Note that hexadecimal key codes greater than 0x7f are not portable accross different systems.

Lines starting with a hash are treated as comments. All other lines whose first word does not start with an open angle bracket are evaluated as normal Tcl commands. As in Tcl source files, newline-backslash sequences are treated as continuation lines.

Errors occuring during replay are reported using the background error mechanism. Upon error, the replay event log file is closed.

KEYWORDS

event, recorder

NAME

scrollbar − Create and manipulate scrollbar widgets

SYNOPSIS

scrollbar *pathName* ?*options*?

STANDARD OPTIONS

activeAttributes	**activeForeground**	**background**	**orient**
activeBackground	**attributes**	**foreground**	**takeFocus**

See the ‘‘options’’ manual entry for details on the standard options.

WIDGET-SPECIFIC OPTIONS

Name: **command**
Class: **Command**
Command-Line Switch: **−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.

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). 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**
Marks the scrollbar as active, which causes it to be displayed as specified by the **activeAttributes**, **activeBackground** and **activeForeground** options.

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*. 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 **deactivate**
Marks the scrollbar as normal, which causes it to be displayed as specified by the **attributes**, **background** and **foreground** options.

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 screen 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 screen 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* **unit**
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* **page**
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.

BINDINGS

Ck 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.

[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.

[3] 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.

[4] 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.

[5] 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.

[6] In horizontal scrollbars the Left and Right keys have the same behavior as mouse clicks over **arrow1** and **arrow2**, respectively. In vertical scrollbars these keys have no effect.

[7] The Prior and Next keys have the same behavior as mouse clicks over **trough1** and **trough2**, respectively.

[8] The Home key adjusts the view to the top (left edge) of the document.

[9] The End key adjusts the view to the bottom (right edge) of the document.

[10] FocusIn and FocusOut events activate and deactive the scrollbars, respectively.

KEYWORDS

scrollbar, widget

NAME

text − Create and manipulate text widgets

SYNOPSIS

text *pathName* ?*options*?

STANDARD OPTIONS

attributes	**selectAttributes**	**selectForeground**	**xScrollCommand**
background	**selectBackground**	**takeFocus**	**yScrollCommand**
foreground			

See the ‘‘options’’ manual entry for details on the standard options.

WIDGET-SPECIFIC OPTIONS

Name: **height**
Class: **Height**
Command-Line Switch: **−height**

Specifies the desired height for the window, in screen lines. Must be at least one.

Name: **state**
Class: **State**
Command-Line Switch: **−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.

Name: **tabs**
Class: **Tabs**
Command-Line Switch: **−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 {2 left 4 6 center}** creates three tab stops at two-column 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 Ck 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 Ck uses default tabs spaced every eight columns.

Name: **width**
Class: **Width**
Command-Line Switch: **−width**

Specifies the desired width for the window in screen columns.

Name: **wrap**
Class: **Wrap**
Command-Line Switch: **−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 colors and attributes. 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 two different kinds of annotations on the text, called tags and marks. Tags allow different portions of the text to be displayed with different attributes and colors. 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.

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.

@*x*,*y* Indicates the character that covers the place 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*.

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 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 two 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**, **attributes**, 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:

−attributes *attrList*
AttrList specifies the attributes to use for characters associated with the tag.

−background *color*
Color specifies the background color to use for characters associated with the tag.

−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 *columns*
If the first character of a text line has a tag for which this option has been specified, then *columns* specifies how much the line should be indented from the left edge of the window. 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 *columns*
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 *columns* specifies how much the line should be indented from the left edge of the window. This option is only used when wrapping is enabled, and it only applies to the second and later display lines for a text line.

−rmargin *columns*
If the first character of a display line has a tag for which this option has been specified, then *columns* specifies how wide a margin to leave between the end of the line and the right edge of the window. 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.

−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.

−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 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.

THE SELECTION

Selection support is implemented via tags. 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**, **selectAttributes**, and **selectForeground** options for the text widget are tied to the **−background**, **−attributes**, 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 moved to 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 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*. 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 (always zero). All of this information is measured in screen coordinates. 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. If the display line containing *index* is not visible on the screen then the return value is an empty list.

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.

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 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 **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 a position later than *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 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 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*. 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 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 **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

Ck 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.

[2] If any normal printing characters are typed, they are inserted at the point of the insertion cursor.

[3] The Left and Right keys move the insertion cursor one character to the left or right; they also clear any selection in the text. Control-b and Control-f behave the same as Left and Right, respectively.

[4] The Up and Down keys move the insertion cursor one line up or down and clear any selection in the text. Control-p and Control-n behave the same as Up and Down, respectively.

[5] The Next and Prior keys move the insertion cursor forward or backwards by one screenful and clear any selection in the text. Control-v moves the view down one screenful without moving the insertion cursor or adjusting the selection.

[6] Home and Control-a move the insertion cursor to the beginning of its line and clear any selection in the widget.

[7] End and Control-e move the insertion cursor to the end of the line and clear any selection in the widget.

[8] 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.

[9] 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.

[10] Control-d deletes the character to the right of the insertion cursor.

[11] 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.

[12] Control-o opens a new line by inserting a newline character in front of the insertion cursor without moving the insertion cursor.

[13] Control-x moves the input focus to the next widget in focus order.

[14] 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. The only known mode of operation where a text widget may not run efficiently is if it has a very large number of different tags. Hundreds of tags should be fine, or even a thousand, but tens of thousands of tags will make texts consume a lot of memory and run slowly.

KEYWORDS

text, widget

NAME

tkerror − Command invoked to process background errors

SYNOPSIS

tkerror *message*

DESCRIPTION

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

A background error is one that occurs in a command that didn't originate with the application. For example, if an error occurs while executing a command specified with a **bind** or **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 Ck library and there is no obvious way for Ck to report the error.

When Ck detects a background error, it saves information about the error and invokes the **tkerror** command later when Ck is idle. Before invoking **tkerror**, Ck restores the **errorInfo** and **errorCode** variables to their values at the time the error occurred, then it invokes **tkerror** with the error message as its only argument. Ck assumes that the application has implemented the **tkerror** command, and that the command will report the error in a way that makes sense for the application. Ck will ignore any result returned by the **tkerror** command.

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

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

The Ck script library includes a default **tkerror** procedure that 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

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. This form is typically used to wait for a newly-created window to appear on the screen before taking some action. At the time of this writing, visibility state changes are unreliable. Thus this form of the **tkwait** command is strongly discouraged. 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.

KEYWORDS

variable, visibility, wait, window

NAME

toplevel − Create and manipulate toplevel widgets

SYNOPSIS

toplevel *pathName* ?*options*?

STANDARD OPTIONS

attributes **border** **foreground** **takefocus**
background

See the ‘‘options’’ manual entry for details on the standard options.

WIDGET-SPECIFIC OPTIONS

Name: **class**
Class: **Class**
Command-Line Switch: **−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.

Name: **height**
Class: **Height**
Command-Line Switch: **−height**

Specifies the desired height for the window in screen lines. If this option is equal to zero then the window will not request any size at all.

Name: **width**
Class: **Width**
Command-Line Switch: **−width**

Specifies the desired width for the window in screen columns. If this option is 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 parent with respect to screen real estate is the terminal’s 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, attributes and border.

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*. 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.

PLACEMENT

The only means to place a toplevel widget on the screen is the **place** geometry manager.

BINDINGS

When a new toplevel is created, it has no default event bindings: toplevels are not intended to be interactive.

KEYWORDS

toplevel, widget, place

NAME

update − Process pending events and/or when-idle handlers

SYNOPSIS

update ?**idletasks**|**screen**?

DESCRIPTION

This command is used to bring the entire application world ‘‘up to date.’’ It flushes all pending output to the display, waits for the server to process that output and return errors or events, handles all pending events of any sort (including when-idle handlers), and repeats this set of operations until there are no pending events, no pending when-idle handlers, no pending output to the server, and no operations still outstanding at the server.

If the **idletasks** keyword is specified as an argument to the command, then no new events or errors are processed; only when-idle idlers 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 handlers, 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**.

If the **screen** keyword is specified as an argument to the command, then the entire screen is repainted from scratch without handling any other events. This is useful if the terminal’s screen has been garbled by another process.

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 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

winfo − Return window-related information

SYNOPSIS

winfo *option* ?*arg arg ...*?

DESCRIPTION

The **winfo** command is used to retrieve information about windows managed by Ck. It can take any of a number of different forms, depending on the *option* argument. The legal forms are:

winfo children *window*

Returns a list containing the path names of all the children of *window*. Top-level windows are returned as children of their logical parents.

winfo class *window*

Returns the class name for *window*.

winfo containing *rootX rootY*

Returns the path name for the window containing the point given by *rootX* and *rootY*. *RootX* and *rootY* are specified as cursor position in the coordinate system of the terminal. 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*. 1 means the terminal's screen is monochrome. Any number higher than 1 means that the terminal supports colors.

winfo exists *window*

Returns 1 if there exists a window named *window*, 0 if no such window exists.

winfo geometry *window*

Returns the geometry for *window*, in the form *width***x***height*+*x*+*y*. All dimensions are in terminal coordinates.

winfo height *window*

Returns a decimal string giving *window*'s height in terminal lines. When a window is first created its height will be 1; 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 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**.

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 reqheight *window*
Returns a decimal string giving *window*'s requested height, in lines. 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 columns. This is the value used by *window*'s geometry manager to compute its geometry.

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 screenheight *window*
Returns a decimal string giving the height of *window*'s terminal screen, in lines.

winfo screenwidth *window*
Returns a decimal string giving the width of *window*'s terminal screen, in columns.

winfo toplevel *window*
Returns the path name of the top-level window containing *window*.

winfo width *window*
Returns a decimal string giving *window*'s width in columns. When a window is first created its width will be 1; 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
children, class, geometry, height, identifier, information, mapped, parent, path name, screen, terminal, width, window