GCL TK Manual

1 General

1.1 Introduction

GCL-TK is a windowing interface for **GNU Common Lisp**. It provides the functionality of the **TK** widget set, which in turn implements a widget set which has the look and feel of **Motif**.

The interface allows the user to draw graphics, get input from menus, make regions mouse sensitive, and bind lisp commands to regions. It communicates over a socket with a '`gcltksrv`' process, which speaks to the display via the **TK** library. The displaying process may run on a machine which is closer to the display, and so involves less communication. It also may remain active even though the lisp is involved in a separate user computation. The display server can, however, interrupt the lisp at will, to inquire about variables and run commands.

The user may also interface with existing `TCL/TK` programs, binding some buttons, or tracking some objects.

The size of the program is moderate. In its current form it adds only about 45K bytes to the lisp image, and the '`gcltksrv`' program uses shared libraries, and is on the order of 150Kbytes on a sparc.

This chapter describes some of the common features of the command structure of widgets, and of control functions. The actual functions for construction of windows are discussed in ⟨undefined⟩ [Widgets], page ⟨undefined⟩, and more general functions for making them appear, lowering them, querying about them in ⟨undefined⟩ [Control], page ⟨undefined⟩.

1.2 Getting Started

Once **GCL** has been properly installed you should be able to do the following simple example:

```
(in-package "TK")
(tkconnect)
(button '.hello :text "Hello World" :command '(print "hi"))
==>.HELLO
(pack '.hello)
```

We first switched to the "TK" package, so that functions like button and pack would be found. After doing the tkconnect, a window should appear on your screen, see See ⟨undefined⟩ [tkconnect], page ⟨undefined⟩. The invocation of the function `button` creates a new function called `.hello` which is a *widget function*. It is then made visible in the window by using the `pack` function.

You may now click on the little window, and you should see the command executed in your lisp. Thus "hi" should be printed in the lisp window. This will happen whether or not you have a job running in the lisp, that is lisp will be interrupted and your command will run, and then return the control to your program.

The function `button` is called a widget constructor, and the function `.hello` is called a widget. If you have managed to accomplish the above, then **GCL** is probably installed

correctly, and you can graduate to the next section! If you dont like reading but prefer to look at demos and code, then you should look in the demos directory, where you will find a number of examples. A monitor for the garbage collector (mkgcmonitor), a demonstration of canvas widgets (mkitems), a sample listbox with scrolling (mklistbox).

1.3 Common Features of Widgets

A *widget* is a lisp symbol which has a function binding. The first argument is always a keyword and is called the *option*. The argument pattern for the remaining arguments depends on the *option*. The most common *option* is `:configure` in which case the remaining arguments are alternating keyword/value pairs, with the same keywords being permitted as at the creation of the widget.

A *widget* is created by means of a *widget constructor*, of which there are currently 15, each of them appearing as the title of a section in ⟨undefined⟩ [Widgets], page ⟨undefined⟩. They live in the `"TK"` package, and for the moment we will assume we have switched to this package. Thus for example `button` is such a widget constructor function. Of course this is lisp, and you can make your own widget constructors, but when you do so it is a good idea to follow the standard argument patterns that are outlined in this section.

```
(button '.hello)
==> .HELLO
```

creates a *widget* whose name is `.hello`. There is a parent child hierarchy among widgets which is implicit in the name used for the widget. This is much like the pathname structure on a Unix or Dos file system, except that `'.'` is used as the separator rather than a `/` or `\`. For this reason the widget instances are sometimes referred to as *pathnames*. A child of the parent widget `.hello` might be called `.hello.joe`, and a child of this last might be `.hello.joe.bar`. The parent of everyone is called `.` . Multiple top level windows are created using the `toplevel` command (see ⟨undefined⟩ [toplevel], page ⟨undefined⟩).

The widget constructor functions take keyword and value pairs, which allow you to specify attributes at the time of creation:

```
(button '.hello :text "Hello World" :width 20)
==>.HELLO
```

indicating that we want the text in the button window to be `Hello World` and the width of the window to be 20 characters wide. Other types of windows allow specification in centimeters `2c`, or in inches (`2i`) or in millimeters `2m` or in pixels `2`. But text windows usually have their dimensions specified as multiples of a character width and height. This latter concept is called a grid.

Once the window has been created, if you want to change the text you do NOT do:

```
(button '.hello :text "Bye World" :width 20)
```

This would be in error, because the window .hello already exists. You would either have to first call

```
(destroy '.hello)
```

But usually you just want to change an attribute. `.hello` is actually a function, as we mentioned earlier, and it is this function that you use:

```
(.hello :configure :text "Bye World")
```

This would simply change the text, and not change where the window had been placed on the screen (if it had), or how it had been packed into the window hierarchy. Here the argument `:configure` is called an *option*, and it specifies which types of keywords can follow it. For example

```
(.hello :flash)
```

is also valid, but in this case the `:text` keyword is not permitted after flash. If it were, then it would mean something else besides what it means in the above. For example one might have defined

```
(.hello :flash :text "PUSH ME")
```

so here the same keyword `:text` would mean something else, eg to flash a subliminal message on the screen.

We often refer to calls to the widget functions as messages. One reason for this is that they actually turn into messages to the graphics process '`gcltksrv`'. To actually see these messages you can do

```
(debugging t).
```

1.4 Return Values

1.4.1 Widget Constructor Return Values

On successful completion, the widget constructor functions return the symbol passed in as the first argument. It will now have a functional binding. It is an error to pass in a symbol which already corresponds to a widget, without first calling the `destroy` command. On failure, an error is signalled.

1.4.2 Widget Return Values

The *widget* functions themselves, do not normally return any value. Indeed the lisp process does not wait for them to return, but merely dispatches the commands, such as to change the text in themselves. Sometimes however you either wish to wait, in order to synchronize, or you wish to see if your command fails or succeeds. You request values by passing the keyword :return and a value indicating the type.

```
(.hello :configure :text "Bye World" :return 'string)
==> ""
==> T
```

the empty string is returned as first value, and the second value `T` indicates that the new text value was successfully set. LISP will not continue until the tkclsrv process indicates back that the function call has succeeded. While waiting of course LISP will continue to process other graphics events which arrive, since otherwise a deadlock would arise: the user for instance might click on a mouse, just after we had decided to wait for a return value from the `.hello` function. More generally a user program may be running in **GCL** and be interrupted to receive and act on communications from the '`gcltksrv`' process. If an error occurred then the second return value of the lisp function will be NIL. In this case the first value, the string is usually an informative message about the type of error.

A special variable `tk::*break-on-errors*` which if not `nil`, requests that that **LISP** signal an error when a message is received indicating a function failed. Whenever a command fails, whether a return value was requested or not, '`gcltksrv`' returns a message indicating failure. The default is to not go into the debugger. When debugging your windows it may be convenient however to set this variable to `T` to track down incorrect messages.

The '`gcltksrv`' process always returns strings as values. If `:return` *type* is specified, then conversion to *type* is accomplished by calling

`(coerce-result` *return-string type*`)`

Here *type* must be a symbol with a `coercion-functions` property. The builtin return types which may be requested are:

`T` in which case the string passed back from the '`gcltksrv`' process, will be read by the lisp reader.

`number` the string is converted to a number using the current *read-base*

`list-strings`

```
(coerce-result "a b {c d} e" 'list-strings)
==> ("a" "b" "c d" "e")
```

`boolean` (coerce-result `"1"` 'boolean) ==> T (coerce-result `"0"` 'boolean) ==> NIL

The above symbols are in the `TK` or `LISP` package. It would be possible to add new types just as the `:return t` is done:

```
(setf (get 't 'coercion-functions)
      (cons #'(lambda (x) (our-read-from-string x 0))
    #'(lambda (x) (format nil "~s" x))))
```

The `coercion-functions` property of a symbol, is a cons whose `car` is the coercion form from a string to some possibly different lisp object, and whose `cdr` is a function which builds a string to send to the graphics server. Often the two functions are inverse functions one of the other up to equal.

1.4.3 Control Function Return Values

The *control* funcions (see ⟨undefined⟩ [Control], page ⟨undefined⟩) do not return a value or wait unless requested to do so, using the `:return` keyword. The types and method of specification are the same as for the Widget Functions in the previous section.

```
(winfo :width '.hello :return 'number)
==> 120
```

indicates that the `.hello` button is actually 120 pixels wide.

1.5 Argument Lists

1.5.1 Widget Functions

The rule is that the first argument for a widget function is a keyword, called the *option*. The pattern of the remaining arguments depends completely on the *option* argument. Thus

```
(.hello option ?arg1? ?arg2? ...)
```

One *option* which is permitted for every widget function is `:configure`. The argument pattern following it is the same keyword/value pair list which is used in widget creation. For a `button` widget, the other valid options are `:deactivate`, `:flash`, and `:invoke`. To find these, since `.hello` was constructed with the `button` constructor, you should see See ⟨undefined⟩ [button], page ⟨undefined⟩. The argument pattern for other options depends completely on the option and the widget function. For example if `.scrollbar` is a scroll bar window, then the option `:set` must be followed by 4 numeric arguments, which indicate how the scrollbar should be displayed, see See ⟨undefined⟩ [scrollbar], page ⟨undefined⟩.

```
(.scrollbar :set a1 a2 a3 a4)
```

If on the other hand `.scale` is a scale (see ⟨undefined⟩ [scale], page ⟨undefined⟩), then we have

```
(.scale :set a1 )
```

only one numeric argument should be supplied, in order to position the scale.

1.5.2 Widget Constructor Argument Lists

These are

```
(widget-constructor pathname :keyword1 value1 :keyword2 value2 ...)
```

to create the widget whose name is *pathname*. The possible keywords allowed are specified in the corresponding section of See ⟨undefined⟩ [Widgets], page ⟨undefined⟩.

1.5.3 Concatenation Using ':' in Argument List

What has been said so far about arguments is not quite true. A special string concatenation construction is allowed in argument lists for widgets, widget constructors and control functions.

First we introduce the function `tk-conc` which takes an arbitrary number of arguments, which may be symbols, strings or numbers, and concatenates these into a string. The print names of symbols are converted to lower case, and package names are ignored.

```
(tk-conc "a" 1 :b 'cd "e") ==> "a1bcde"
```

One could use `tk-conc` to construct arguments for widget functions. But even though `tk-conc` has been made quite efficient, it still would involve the creation of a string. The `:` construct avoids this. In a call to a widget function, a widget constructor, or a control function you may remove the call to `tk-conc` and place `:` in between each of its arguments. Those functions are able to understand this and treat the extra arguments as if they were glued together in one string, but without the extra cost of actually forming that string.

```
(tk-conc a b c .. w) <==> a : b : c : ... w
(setq i 10)
(.hello :configure :text i : " pies")
(.hello :configure :text (tk-conc i  " pies"))
(.hello :configure :text (format nil "~a pies" i))
```

The last three examples would all result in the text string being `"10 pies"`, but the first method is the most efficient. That call will be made with no string or cons creation. The **GC Monitor** example, is written in such a way that there is no creation of `cons` or `string`

types during normal operation. This is particularly useful in that case, since one is trying to monitor usage of conses by other programs, not its own usage.

1.6 Lisp Functions Invoked from Graphics

It is possible to make certain areas of a window mouse sensitive, or to run commands on reception of certain events such as keystrokes, while the focus is in a certain window. This is done by having a lisp function invoked or some lisp form evaluated. We shall refer to such a lisp function or form as a *command*.

For example

```
(button '.button :text "Hello" :command '(print "hi"))
(button '.jim :text "Call Jim" :command 'call-jim)
```

In the first case when the window `.button` is clicked on, the word "hi" will be printed in the lisp to standard output. In the second case `call-jim` will be funcalled with no arguments.

A command must be one of the following three types. What happens depends on which type it is:

'`function`'
: If the value satisfies `functionp` then it will be called with a number of arguments which is dependent on the way it was bound, to graphics.

'`string`'
: If the command is a string, then it is passed directly to **TCL/TK** for evaluation on that side. Lisp will not be required for the evaluation when the command is invoked.

'`lisp form`'
: Any other lisp object is regarded as a lisp form to be eval'd, and this will be done when the command is invoked.

The following keywords accept as their value a command:

```
:command
:yscroll    :yscrollcommand
:xscroll    :xscrollcommand
:scrollcommand
:bind
```

and in addition `bind` takes a command as its third argument, see See ⟨undefined⟩ [bind], page ⟨undefined⟩.

Below we give three different examples using the 3 possibilities for a command: functionp, string, and lisp form. They all accomplish exactly the same thing. For given a frame `.frame` we could construct a listbox in it as:

```
(listbox '.frame.listbox :yscroll 'joe)
```

Then whenever the listbox view position changes, or text is inserted, so that something changes, the function `joe` will be invoked with 4 arguments giving the totalsize of the text, maximum number of units the window can display, the index of the top unit, and finally the index of the bottom unit. What these arguments are is specific to the widget `listbox` and is documented See ⟨undefined⟩ [listbox], page ⟨undefined⟩.

`joe` might be used to do anything, but a common usage is to have `joe` alter the position of some other window, such as a scroll bar window. Indeed if `.scrollbar` is a scrollbar then the function

```
(defun joe (a b c d)
  (.scrollbar :set a b c d))
```

would look after sizing the scrollbar appropriately for the percentage of the window visible, and positioning it.

A second method of accomplishing this identical, using a string (the second type of command),

```
(listbox '.frame.listbox :yscroll ".scrollbar set")
```

and this will not involve a call back to lisp. It uses the fact that the **TK** graphics side understands the window name `.scrollbar` and that it takes the *option* `set`. Note that it does not get the `:` before the keyword in this case.

In the case of a command which is a *lisp form* but is not installed via `bind` or `:bind`, then the form will be installed as

`#'(lambda (&rest *arglist*)` *lisp-form*`)`

where the *lisp-form* might wish to access the elements of the special variable `*arglist*`. Most often this list will be empty, but for example if the command was setup for `.scale` which is a *scale*, then the command will be supplied one argument which is the new numeric value which is the scale position. A third way of accomplishing the scrollbar setting using a lisp form is:

```
(listbox '.frame.listbox :yscroll '(apply '.scrollbar :set *arglist*))
```

The `bind` command and `:bind` keyword, have an additional wrinkle, see See ⟨undefined⟩ [bind], page ⟨undefined⟩. These are associated to an event in a particular window, and the lisp function or form to be evaled must have access to that information. For example the x y position, the window name, the key pressed, etc. This is done via *percent symbols* which are specified, see See ⟨undefined⟩ [bind], page ⟨undefined⟩.

```
(bind "Entry" "<Control-KeyPress>" '(emacs-move  %W %A ))
```

will cause the function emacs-move to be be invoked whenever a control key is pressed (unless there are more key specific or window specific bindings of said key). It will be invoked with two arguments, the first %W indicating the window in which it was invoked, and the second being a string which is the ascii keysym which was pressed at the same time as the control key.

These *percent constructs* are only permitted in commands which are invoked via `bind` or `:bind`. The lisp form which is passed as the command, is searched for the percent constructs, and then a function

```
#'(lambda (%W %A) (emacs-move %W %A))
```

will be invoked with two arguments, which will be supplied by the **TK** graphics server, at the time the command is invoked. The `*arglist*` construct is not available for these commands.

1.7 Linked Variables

It is possible to link lisp variables to **TK** variables. In general when the **TK** variable is changed, by for instance clicking on a radiobutton, the linked lisp variable will be changed. Conversely changing the lisp variable will be noticed by the **TK** graphics side, if one does the assignment in lisp using `setk` instead of `setq`.

```
(button '.hello :textvariable '*message* :text "hi there")
(pack '.hello)
```

This causes linking of the global variable `*message*` in lisp to a corresponding variable in **TK**. Moreover the message that is in the button `.hello` will be whatever the value of this global variable is (so long as the **TK** side is notified of the change!).

Thus if one does

```
(setk *message* "good bye")
```

then the button will change to have *good bye* as its text. The lisp macro `setk` expands into

```
(prog1 (setf *message* "good bye") (notice-text-variables))
```

which does the assignment, and then goes thru the linked variables checking for those that have changed, and updating the **TK** side should there be any. Thus if you have a more complex program which might have done the assignment of your global variable, you may include the call to `notice-text-variables` at the end, to assure that the graphics side knows about the changes.

A variable which is linked using the keyword `:textvariable` is always a variable containing a string.

However it is possible to have other types of variables.

```
(checkbutton '.checkbutton1 :text "A button" :variable '(boolean *joe*))
(checkbutton '.checkbutton2 :text "A button" :variable '*joe*)
(checkbutton '.checkbutton3 :text "Debugging" :variable '(t *debug*)
             :onvalue 100 :offvalue -1)
```

The first two examples are the same in that the default variable type for a checkbutton is `boolean`. Notice that the specification of a variable type is by (*type* `variable`). The types which are permissible are those which have coercion-fucntions, See ⟨undefined⟩ [Return Values], page ⟨undefined⟩. In the first example a variable `*joe*` will be linked, and its default initial value will be set to nil, since the default initial state of the check button is off, and the default off value is nil. Actually on the **TK** side, the corresponding boolean values are `"1"` and `"0"`, but the `boolean` type makes these become `t` and `nil`.

In the third example the variable *debug* may have any lisp value (here *type* is `t`). The initial value will be made to be `-1`, since the checkbutton is off. Clicking on `.checkbutton3` will result in the value of `*debug*` being changed to 100, and the light in the button will be toggled to on, See ⟨undefined⟩ [checkbutton], page ⟨undefined⟩. You may set the variable to be another value besides 100.

You may also call

```
(link-text-variable '*joe* 'boolean)
```

to cause the linking of a variable named *joe*. This is done automatically whenever the variable is specified after one of the keys

```
:variable   :textvariable.
```

Just as one must be cautious about using global variables in lisp, one must be cautious in making such linked variables. In particular note that the **TK** side, uses variables for various purposes. If you make a checkbutton with pathname `.a.b.c` then unless you specify a `:variable` option, the variable `c` will become associated to the **TK** value of the checkbutton. We do NOT link this variable by default, feeling that one might inadvertently alter global variables, and that they would not typically use the lisp convention of being of the form `*c*`. You must specify the `:variable` option, or call `link-variable`.

1.8 tkconnect

tkconnect `&key host display can-rsh gcltksrv`

This function provides a connection to a graphics server process, which in turn connects to possibly several graphics display screens. The graphics server process, called '`gcltksrv`' may or may not run on the same machine as the lisp to which it is attached. `display` indicates the name of the default display to connect to, and this in turn defaults to the value of the environment variable `DISPLAY`.

When *tkconnect* is invoked, a socket is opened and it waits for a graphics process to connect to it. If the host argument is not supplied, then a process will be spawned which will connect back to the lisp process. The name of the command for invoking the process is the value of the '`gcltksrv`' argument, which defaults to the value of the environment variable `GCL_TK_SERVER`. If that variable is not set, then the lisp `*lib-directory*` is searched for an entry '`gcl-tk/gcltksrv`'.

If `host` is supplied, then a command to run on the remote machine will be printed on standard output. If `can-rsh` is not nil, then the command will not be printed, but rather an attempt will be made to rsh to the machine, and to run the command.

Thus

```
(tkconnect)
```

would start the process on the local machine, and use for `display` the value of the environment variable `DISPLAY`.

```
(tkconnect :host "max.ma.utexas.edu" :can-rsh t)
```

would cause an attempt to rsh to `max` and to run the command there, to connect back to the appropriate port on the localhost.

You may indicate that different *toplevel* windows be on different displays, by using the `:display` argument when creating the window, See ⟨undefined⟩ [toplevel], page ⟨undefined⟩.

Clearly you must have a copy of the program '`gcltksrv`' and **TK** libraries installed on the machine where you wish to run the server.

2 Widgets

2.1 button

button \- Create and manipulate button widgets

Synopsis

button *pathName ?options?*

Standard Options

activeBackground	bitmap	font	relief
activeForeground	borderWidth	foreground	text
anchor	cursor	padX	textVariable
background	disabledForeground	padY	

See ⟨undefined⟩ [options], page ⟨undefined⟩, for more information.

Arguments for Button

:command

Name="**command**" Class="**Command**"

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

:height

Name="**height**" Class="**Height**"

Specifies a desired height for the button. If a bitmap is being displayed in the button then the value is in screen units (i.e. any of the forms acceptable to **Tk_GetPixels**); for text it is in lines of text. If this option isn't specified, the button's desired height is computed from the size of the bitmap or text being displayed in it.

:state

Name="**state**" Class="**State**"

Specifies one of three states for the button: **normal**, **active**, or **disabled**. In normal state the button is displayed using the **foreground** and **background** options. The active state is typically used when the pointer is over the button. In active state the button is displayed using the **activeForeground** and **activeBackground** options. Disabled state means that the button is insensitive: it doesn't activate and doesn't respond to mouse button presses. In this state the **disabledForeground** and **background** options determine how the button is displayed.

:width

Name="**width**" Class="**Width**"

Specifies a desired width for the button. If a bitmap is being displayed in the button then the value is in screen units (i.e. any of the forms acceptable to **Tk_GetPixels**); for text it is in characters. If this option isn't specified, the button's desired width is computed from the size of the bitmap or text being displayed in it.

Description

The **button** command creates a new window (given by the *pathName* argument) and makes it into a button widget. Additional options, described above, may be specified on the command line or in the option database to configure aspects of the button such as its colors, font, text, and initial relief. The **button** command returns its *pathName* argument. At the time this command is invoked, there must not exist a window named *pathName*, but *pathName*'s parent must exist.

A button is a widget that displays a textual string or bitmap. It can display itself in either of three different ways, according to the **state** option; it can be made to appear raised, sunken, or flat; and it can be made to flash. When a user invokes the button (by pressing mouse button 1 with the cursor over the button), then the Tcl command specified in the **:command** option is invoked.

A Button Widget's Arguments

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 **:activate**

Change the button's state to **active** and redisplay the button using its active foreground and background colors instead of normal colors. This command is ignored if the button's state is **disabled**. This command is obsolete and will eventually be removed; use "*pathName* **:configure :state active**" instead.

pathName **:configure** ?*option*? ?*value option value ...*?

Query or modify the configuration options of the widget. If no *option* is specified, returns a list describing all of the available options for *pathName* (see **Tk_ConfigureInfo** for information on the format of this list). If *option* is specified with no *value*, then the command returns a list describing the one named option (this list will be identical to the corresponding sublist of the value returned if no *option* is specified). If one or more *option:value* pairs are specified, then the command modifies the given widget option(s) to have the given value(s); in this case the command returns an empty string. *Option* may have any of the values accepted by the **button** command.

pathName **:deactivate**
Change the button's state to **normal** and redisplay the button using its normal foreground and background colors. This command is ignored if the button's state is **disabled**. This command is obsolete and will eventually be removed; use "*pathName* **:configure :state normal**" instead.

pathName **:flash**
Flash the button. This is accomplished by redisplaying the button several times, alternating between active and normal colors. At the end of the flash the button is left in the same normal/active state as when the command was invoked. This command is ignored if the button's state is **disabled**.

pathName **:invoke**
Invoke the Tcl command associated with the button, if there is one. The return value is the return value from the Tcl command, or an empty string if there is no command associated with the button. This command is ignored if the button's state is **disabled**.

"Default Bindings"

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

[1] The button activates whenever the mouse passes over it and deactivates whenever the mouse leaves the button.

[2] The button's relief is changed to sunken whenever mouse button 1 is pressed over the button, and the relief is restored to its original value when button 1 is later released.

[3] If mouse button 1 is pressed over the button and later released over the button, the button is invoked. However, if the mouse is not over the button when button 1 is released, then no invocation occurs.

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

2.2 listbox

listbox \- Create and manipulate listbox widgets

Synopsis

listbox *pathName* ?*options*?

Standard Options

background	foreground	selectBackground	xScrollCommand
borderWidth	font	selectBorderWidth	yScrollCommand
cursor	geometry	selectForeground	
exportSelection	relief	setGrid	

See ⟨undefined⟩ [options], page ⟨undefined⟩, for more information.

Arguments for Listbox

None.

Description

The **listbox** command creates a new window (given by the *pathName* argument) and makes it into a listbox widget. Additional options, described above, may be specified on the command line or in the option database to configure aspects of the listbox such as its colors, font, text, and relief. The **listbox** command returns its *pathName* argument. At the time this command is invoked, there must not exist a window named *pathName*, but *pathName*'s parent must exist.

A listbox is a widget that displays a list of strings, one per line. When first created, a new listbox has no elements in its list. Elements may be added or deleted using widget commands described below. In addition, one or more elements may be selected as described below. If a listbox is exporting its selection (see **exportSelection** option), then it will observe the standard X11 protocols for handling the selection; listbox selections are available as type **STRING**, consisting of a Tcl list with one entry for each selected element.

For large lists only a subset of the list elements will be displayed in the listbox window at once; commands described below may be used to change the view in the window. Listboxes allow scrolling in both directions using the standard **xScrollCommand** and **yScrollCommand** options. They also support scanning, as described below.

A Listbox's Arguments

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 **:configure** *?option? ?value option value ...?*

Query or modify the configuration options of the widget. If no *option* is specified, returns a list describing all of the available options for *pathName* (see **Tk_ConfigureInfo** for information on the format of this list). If *option* is specified with no *value*, then the command returns a list describing the one named option (this list will be identical to the corresponding sublist of the value returned if no *option* is specified). If one or more *option:value* pairs are specified, then the command modifies the given widget option(s) to have the given

value(s); in this case the command returns an empty string. *Option* may have any of the values accepted by the **listbox** command.

pathName **:curselection**
Returns a list containing the 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?*
Delete one or more elements of the listbox. *First* and *last* give the integer indices of the first and last elements in the range to be deleted. If *last* isn't specified it defaults to *first*, i.e. a single element is deleted. An index of **0** corresponds to the first element in the listbox. Either *first* or *last* may be specified as **end**, in which case it refers to the last element of the listbox. This command returns an empty string

pathName **:get** *index*
Return the contents of the listbox element indicated by *index*. *Index* must be a non-negative integer (0 corresponds to the first element in the listbox), or it may also be specified as **end** to indicate the last element in the listbox.

pathName **:insert** *index ?element element ...?*
Insert zero or more new elements in the list just before the element given by *index*. If *index* is specified as **end** then the new elements are added to the end of the list. Returns an empty string.

pathName **:nearest** *y*
Given a y-coordinate within the listbox window, this command returns the index of the (visible) listbox element nearest to that y-coordinate.

pathName **:scan** *option args*
This command is used to implement scanning on listboxes. It has two forms, depending on *option*:

pathName **:scan :mark** *x y*
Records *x* and *y* and the current view in the listbox window; used in conjunction with later **scan dragto** commands. Typically this command is associated with a mouse button press in the widget. It returns an empty string.

pathName **:scan :dragto** *x y.*
This command computes the difference between its *x* and *y* arguments and the *x* and *y* arguments to the last **scan mark** command for the widget. It then adjusts the view by 10 times the difference in coordinates. This command is typically associated with mouse motion events in the widget, to produce the effect of dragging the list at high speed through the window. The return value is an empty string.

pathName **:select** *option arg*
This command is used to adjust the selection within a listbox. It has several forms, depending on *option.* In all of the forms the index **end** refers to the last element in the listbox.

pathName **:select :adjust** *index*
Locate the end of the selection nearest to the element 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 listbox, then this command is identical to the **select from** widget command. Returns an empty string.

pathName **:select :clear**
If the selection is in this listbox then it is cleared so that none of the listbox's elements are selected anymore.

pathName **:select :from** *index*
Set the selection to consist of element *index*, and make *index* the anchor point for future **select to** widget commands. Returns an empty string.

pathName **:select :to** *index*
Set the selection to consist of the elements from the anchor point to element *index*, inclusive. 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, this command is identical to **select from**. Returns an empty string.

pathName **:size**
Returns a decimal string indicating the total number of elements in the listbox.

pathName **:xview** *index*
Adjust the view in the listbox so that character position *index* is displayed at the left edge of the widget. Returns an empty string.

pathName **:yview** *index*
Adjust the view in the listbox so that element *index* is displayed at the top of the widget. If *index* is specified as **end** it indicates the last element of the listbox. Returns an empty string.

"Default Bindings"

Tk automatically creates class bindings for listboxes that give them the following default behavior:

[1] When button 1 is pressed over a listbox, the element underneath the mouse cursor is selected. The mouse can be dragged to select a range of elements.

[2] The ends of the selection can be adjusted by dragging with mouse button 1 while the shift key is down; this will adjust the end of the selection that was nearest to the mouse cursor when button 1 was pressed.

[3] The view in the listbox can be adjusted by dragging with mouse button 2.

The behavior of listboxes can be changed by defining new bindings for individual widgets or by redefining the class bindings. In addition, the procedure **tk_listboxSingleSelect** may be invoked to change listbox behavior so that only a single element may be selected at once.

Keywords

listbox, widget

2.3 scale

scale \- Create and manipulate scale widgets

Synopsis

scale *pathName* ?*options*?

Standard Options

`activeForeground`	`borderWidth`	`font`	`orient`
`background`	`cursor`	`foreground`	`relief`

See ⟨undefined⟩ [options], page ⟨undefined⟩, for more information.

Arguments for Scale

:command

Name="**command**" Class="**Command**"

Specifies the prefix of a Tcl command to invoke whenever the value of the scale is changed interactively. The actual command consists of this option followed by a space and a number. The number indicates the new value of the scale.

:from

Name="**from**" Class="**From**"

Specifies the value corresponding to the left or top end of the scale. Must be an integer.

:label

Name="**label**" Class="**Label**"

Specifies a string to displayed as a label for the scale. For vertical scales the label is displayed just to the right of the top end of the scale. For horizontal scales the label is displayed just above the left end of the scale.

:length

Name="**length**" Class="**Length**"

Specifies the desired long dimension of the scale in screen units, that is in any of the forms acceptable to **Tk_GetPixels**. For vertical scales this is the scale's height; for horizontal scales it is the scale's width.

:showvalue

Name="**showValue**" Class="**ShowValue**"

Specifies a boolean value indicating whether or not the current value of the scale is to be displayed.

:sliderforeground

Name="**sliderForeground**" Class="**sliderForeground**"

Specifies the color to use for drawing the slider under normal conditions. When the mouse is in the slider window then the slider's color is determined by the **activeForeground** option.

:sliderlength

Name="**sliderLength**" Class="**SliderLength**"

Specfies the size of the slider, measured in screen units along the slider's long dimension. The value may be specified in any of the forms acceptable to **Tk_GetPixels**.

:state

Name="**state**" Class="**State**"

Specifies one of two states for the scale: **normal** or **disabled**. If the scale is disabled then the value may not be changed and the scale won't activate when the mouse enters it.

:tickinterval

Name="**tickInterval**" Class="**TickInterval**"

Must be an integer value. Determines the spacing between numerical tick-marks displayed below or to the left of the slider. If specified as 0, then no tick-marks will be displayed.

:to

Name="**to**" Class="**To**"

Specifies the value corresponding to the right or bottom end of the scale. Must be an integer. This value may be either less than or greater than the **from** option.

:width

Name="**width**" Class="**Width**"

Specifies the desired narrow dimension of the scale in screen units (i.e. any of the forms acceptable to **Tk_GetPixels**). For vertical scales this is the scale's width; for horizontal scales this is the scale's height.

Description

The **scale** command creates a new window (given by the *pathName* argument) and makes it into a scale widget. Additional options, described above, may be specified on the command line or in the option database to configure aspects of the scale such as its colors, orientation, and relief. The **scale** command returns its *pathName* argument. At the time this command is invoked, there must not exist a window named *pathName*, but *pathName*'s parent must exist.

A scale is a widget that displays a rectangular region and a small *slider*. The rectangular region corresponds to a range of integer values (determined by the **from** and **to** options), and the position of the slider selects a particular integer value. The slider's position (and hence the scale's value) may be adjusted by clicking or dragging with the mouse as described in the BINDINGS section below. Whenever the scale's value is changed, a Tcl command is invoked (using the **command** option) to notify other interested widgets of the change.

Three annotations may be displayed in a scale widget: a label appearing at the top-left of the widget (top-right for vertical scales), a number displayed just underneath the slider (just to the left of the slider for vertical scales), and a collection of numerical tick-marks just underneath the current value (just to the left of the current value for vertical scales). Each of these three annotations may be selectively enabled or disabled using the configuration options.

A Scale's"Argumentsommand"

The **scale** command creates a new Tcl command whose name is *pathName*. This command may be used to invoke various operations on the widget. It has the following general form:

pathName option ?arg arg ...?

Option and the *arg*s determine the exact behavior of the command. The following commands are possible for scale widgets:

pathName **:configure** ?*option*? ?*value option value ...*?
: Query or modify the configuration options of the widget. If no *option* is specified, returns a list describing all of the available options for *pathName* (see **Tk_ConfigureInfo** for information on the format of this list). If *option* is specified with no *value*, then the command returns a list describing the one named option (this list will be identical to the corresponding sublist of the value returned if no *option* is specified). If one or more *option:value* pairs are specified, then the command modifies the given widget option(s) to have the given value(s); in this case the command returns an empty string. *Option* may have any of the values accepted by the **scale** command.

pathName **:get**
: Returns a decimal string giving the current value of the scale.

pathName **:set** *value*
: This command is invoked to change the current value of the scale, and hence the position at which the slider is displayed. *Value* gives the new value for the scale.

Bindings

When a new scale is created, it is given the following initial behavior by default:

<Enter> Change the slider display to use **activeForeground** instead of **sliderForeground**.

<Leave> Reset the slider display to use **sliderForeground** instead of **activeForeground**.

<ButtonPress-1>
Change the slider display so that the slider appears sunken rather than raised. Move the slider (and adjust the scale's value) to correspond to the current mouse position.

<Button1-Motion>
Move the slider (and adjust the scale's value) to correspond to the current mouse position.

<ButtonRelease-1>
Reset the slider display so that the slider appears raised again.

Keywords

scale, widget

2.4 canvas

canvas \- Create and manipulate canvas widgets

Synopsis

canvas *pathName* ?*options*?

Standard Options

background	insertBorderWidth	relief	xScrollCommand
borderWidth	insertOffTime	selectBackground	yScrollCommand
cursor	insertOnTime	selectBorderWidth	
insertBackground	insertWidth	selectForeground	

See ⟨undefined⟩ [options], page ⟨undefined⟩, for more information.

Arguments for Canvas

:closeenough

Name="**closeEnough**" Class="**CloseEnough**"

Specifies a floating-point value indicating how close the mouse cursor must be to an item before it is considered to be "inside" the item. Defaults to 1.0.

:confine

Name="**confine**" Class="**Confine**"

Specifies a boolean value that indicates whether or not it should be allowable to set the canvas's view outside the region defined by the **scrollRegion** argument. Defaults to true, which means that the view will be constrained within the scroll region.

:height

Name="**height**" Class="**Height**"

Specifies a desired window height that the canvas widget should request from its geometry manager. The value may be specified in any of the forms described in the COORDINATES section below.

:scrollincrement

Name="**scrollIncrement**" Class="**ScrollIncrement**"

Specifies a distance used as increment during scrolling: when one of the arrow buttons on an associated scrollbar is pressed, the picture will shift by this distance. The distance may be specified in any of the forms described in the COORDINATES section below.

:scrollregion

Name="**scrollRegion**" Class="**ScrollRegion**"

Specifies a list with four coordinates describing the left, top, right, and bottom coordinates of a rectangular region. This region is used for scrolling purposes and is considered to be the boundary of the information in the canvas. Each of the coordinates may be specified in any of the forms given in the COORDINATES section below.

:width

Name="**width**" Class="**width**"

Specifies a desired window width that the canvas widget should request from its geometry manager. The value may be specified in any of the forms described in the COORDINATES section below.

Introduction

The **canvas** command creates a new window (given by the *pathName* argument) and makes it into a canvas widget. Additional options, described above, may be specified on the command line or in the option database to configure aspects of the canvas such as its colors and 3-D relief. The **canvas** command returns its *pathName* argument. At the time this command is invoked, there must not exist a window named *pathName*, but *pathName*'s parent must exist.

Canvas widgets implement structured graphics. A canvas displays any number of *items*, which may be things like rectangles, circles, lines, and text. Items may be manipulated

(e.g. moved or re-colored) and commands may be associated with items in much the same way that the **bind** command allows commands to be bound to widgets. For example, a particular command may be associated with the <Button-1> event so that the command is invoked whenever button 1 is pressed with the mouse cursor over an item. This means that items in a canvas can have behaviors defined by the Tcl scripts bound to them.

Display List

The items in a canvas are ordered for purposes of display, with the first item in the display list being displayed first, followed by the next item in the list, and so on. Items later in the display list obscure those that are earlier in the display list and are sometimes referred to as being "on top" of earlier items. When a new item is created it is placed at the end of the display list, on top of everything else. Widget commands may be used to re-arrange the order of the display list.

Item Ids And Tags

Items in a canvas widget may be named in either of two ways: by id or by tag. Each item has a unique identifying number which is assigned to that item when it is created. The id of an item never changes and id numbers are never re-used within the lifetime of a canvas widget.

Each item may also have any number of *tags* associated with it. A tag is just a string of characters, and it may take any form except that of an integer. For example, "x123" is OK but "123" isn't. The same tag may be associated with many different items. This is commonly done to group items in various interesting ways; for example, all selected items might be given the tag "selected".

The tag **all** is implicitly associated with every item in the canvas; it may be used to invoke operations on all the items in the canvas.

The tag **current** is managed automatically by Tk; it applies to the *current item*, which is the topmost item whose drawn area covers the position of the mouse cursor. If the mouse is not in the canvas widget or is not over an item, then no item has the **current** tag.

When specifying items in canvas widget commands, if the specifier is an integer then it is assumed to refer to the single item with that id. If the specifier is not an integer, then it is assumed to refer to all of the items in the canvas that have a tag matching the specifier. The symbol *tagOrId* is used below to indicate that an argument specifies either an id that selects a single item or a tag that selects zero or more items. Some widget commands only operate on a single item at a time; if *tagOrId* is specified in a way that names multiple items, then the normal behavior is for the command to use the first (lowest) of these items in the display list that is suitable for the command. Exceptions are noted in the widget command descriptions below.

Coordinates

All coordinates related to canvases are stored as floating-point numbers. Coordinates and distances are specified in screen units, which are floating-point numbers optionally followed by one of several letters. If no letter is supplied then the distance is in pixels. If the letter is **m** then the distance is in millimeters on the screen; if it is **c** then the distance is in

centimeters; **i** means inches, and **p** means printers points (1/72 inch). Larger y-coordinates refer to points lower on the screen; larger x-coordinates refer to points farther to the right.

Transformations

Normally the origin of the canvas coordinate system is at the upper-left corner of the window containing the canvas. It is possible to adjust the origin of the canvas coordinate system relative to the origin of the window using the **xview** and **yview** widget commands; this is typically used for scrolling. Canvases do not support scaling or rotation of the canvas coordinate system relative to the window coordinate system.

Indidividual items may be moved or scaled using widget commands described below, but they may not be rotated.

Indices

Text items support the notion of an *index* for identifying particular positions within the item. Indices are used for commands such as inserting text, deleting a range of characters, and setting the insertion cursor position. An index may be specified in any of a number of ways, and different types of items may support different forms for specifying indices. Text items support the following forms for an index; if you define new types of text-like items, it would be advisable to support as many of these forms as practical. Note that it is possible to refer to the character just after the last one in the text item; this is necessary for such tasks as inserting new text at the end of the item.

number A decimal number giving the position of the desired character within the text item. 0 refers to the first character, 1 to the next character, and so on. A number less than 0 is treated as if it were zero, and a number greater than the length of the text item is treated as if it were equal to the length of the text item.

end Refers to the character just after the last one in the item (same as the number of characters in the item).

insert Refers to the character just before which the insertion cursor is drawn in this item.

sel.first Refers to the first selected character in the item. If the selection isn't in this item then this form is illegal.

sel.last Refers to the last selected character in the item. If the selection isn't in this item then this form is illegal.

`@`*x,y* Refers to the character at the point given by x and y, where x and y are specified in the coordinate system of the canvas. If x and y lie outside the coordinates covered by the text item, then they refer to the first or last character in the line that is closest to the given point.

A Canvas Widget's Arguments

The **canvas** command creates a new Tcl command whose name is *pathName*. This command may be used to invoke various operations on the widget. It has the following general form:

pathName option ?arg arg ...?

Option and the *arg*s determine the exact behavior of the command. The following widget commands are possible for canvas widgets:

pathName **:addtag** *tag searchSpec ?arg arg ...?*

For each item that meets the constraints specified by *searchSpec* and the *arg*s, add *tag* to the list of tags associated with the item if it isn't already present on that list. It is possible that no items will satisfy the constraints given by *searchSpec and arg*s, in which case the command has no effect. This command returns an empty string as result. *SearchSpec* and *arg*'s may take any of the following forms:

above *tagOrId*

Selects the item just after (above) the one given by *tagOrId* in the display list. If *tagOrId* denotes more than one item, then the last (topmost) of these items in the display list is used.

all Selects all the items in the canvas.

below *tagOrId*

Selects the item just before (below) the one given by *tagOrId* in the display list. If *tagOrId* denotes more than one item, then the first (lowest) of these items in the display list is used.

closest *x y ?halo? ?start?*

Selects the item closest to the point given by *x* and *y*. If more than one item is at the same closest distance (e.g. two items overlap the point), then the top-most of these items (the last one in the display list) is used. If *halo* is specified, then it must be a non-negative value. Any item closer than *halo* to the point is considered to overlap it. The *start* argument may be used to step circularly through all the closest items. If *start* is specified, it names an item using a tag or id (if by tag, it selects the first item in the display list with the given tag). Instead of selecting the topmost closest item, this form will select the topmost closest item that is below *start* in the display list; if no such item exists, then the selection behaves as if the *start* argument had not been specified.

enclosed *x1 y1 x2 y2*

Selects all the items completely enclosed within the rectangular region given by *x1*, *y1*, *x2*, and *y2*. *X1* must be no greater then *x2* and *y1* must be no greater than *y2*.

overlapping *x1 y1 x2 y2*

Selects all the items that overlap or are enclosed within the rectangular region given by *x1*, *y1*, *x2*, and *y2*. *X1* must be no greater then *x2* and *y1* must be no greater than *y2*.

withtag *tagOrId*

Selects all the items given by *tagOrId*.

pathName **:bbox** *tagOrId ?tagOrId tagOrId ...?*

Returns a list with four elements giving an approximate bounding box for all the items named by the *tagOrId* arguments. The list has the form "*x1 y1 x2 y2*" such that the drawn areas of all the named elements are within the region bounded by *x1* on the left, *x2* on the right, *y1* on the top, and *y2* on the bottom. The return value may overestimate the actual bounding box by a few pixels. If no items match any of the *tagOrId* arguments then an empty string is returned.

pathName **:bind** *tagOrId ?sequence? ?command?*

This command associates *command* with all the items given by *tagOrId* such that whenever the event sequence given by *sequence* occurs for one of the items the command will be invoked. This widget command is similar to the **bind** command except that it operates on items in a canvas rather than entire widgets. See the **bind** manual entry for complete details on the syntax of *sequence* and the substitutions performed on *command* before invoking it. If all arguments are specified then a new binding is created, replacing any existing binding for the same *sequence* and *tagOrId* (if the first character of *command* is "+" then *command* augments an existing binding rather than replacing it). In this case the return value is an empty string. If *command* is omitted then the command returns the *command* associated with *tagOrId* and *sequence* (an error occurs if there is no such binding). If both *command* and *sequence* are omitted then the command returns a list of all the sequences for which bindings have been defined for *tagOrId*.

The only events for which bindings may be specified are those related to the mouse and keyboard, such as **Enter**, **Leave**, **ButtonPress**, **Motion**, and **KeyPress**. The handling of events in canvases uses the current item defined in ITEM IDS AND TAGS above. **Enter** and **Leave** events trigger for an item when it becomes the current item or ceases to be the current item; note that these events are different than **Enter** and **Leave** events for windows. Mouse-related events are directed to the current item, if any. Keyboard-related events are directed to the focus item, if any (see the **focus** widget command below for more on this).

It is possible for multiple commands to be bound to a single event sequence for a single object. This occurs, for example, if one command is associated with the item's id and another is associated with one of the item's tags. When this occurs, the first matching binding is used. A binding for the item's id has highest priority, followed by the oldest tag for the item and proceeding through all of the item's tags up through the most-recently-added one. If a binding is associated with the tag **all**, the binding will have lower priority than all other bindings associated with the item.

pathName **:canvasx** *screenx ?gridspacing?*

Given a screen x-coordinate *screenx* this command returns the canvas x-coordinate that is displayed at that location. If *gridspacing* is specified, then the canvas coordinate is rounded to the nearest multiple of *gridspacing* units.

pathName **:canvasy** *screeny ?gridspacing?*

Given a screen y-coordinate *screeny* this command returns the canvas y-coordinate that is displayed at that location. If *gridspacing* is specified, then the canvas coordinate is rounded to the nearest multiple of *gridspacing* units.

pathName **:configure ?***option*? ?*value*? ?*option value ...*?
Query or modify the configuration options of the widget. If no *option* is specified, returns a list describing all of the available options for *pathName* (see **Tk_ConfigureInfo** for information on the format of this list). If *option* is specified with no *value*, then the command returns a list describing the one named option (this list will be identical to the corresponding sublist of the value returned if no *option* is specified). If one or more *option:value* pairs are specified, then the command modifies the given widget option(s) to have the given value(s); in this case the command returns an empty string. *Option* may have any of the values accepted by the **canvas** command.

pathName **:coords** *tagOrId* ?*x0 y0 ...*?
Query or modify the coordinates that define an item. If no coordinates are specified, this command returns a list whose elements are the coordinates of the item named by *tagOrId*. If coordinates are specified, then they replace the current coordinates for the named item. If *tagOrId* refers to multiple items, then the first one in the display list is used.

pathName **:create** *type x y* ?*x y ...*? ?*option value ...*?
Create a new item in *pathName* of type *type*. The exact format of the arguments after **type** depends on **type**, but usually they consist of the coordinates for one or more points, followed by specifications for zero or more item options. See the subsections on individual item types below for more on the syntax of this command. This command returns the id for the new item.

pathName **:dchars** *tagOrId first* ?*last*?
For each item given by *tagOrId*, delete the characters in the range given by *first* and *last*, inclusive. If some of the items given by *tagOrId* don't support text operations, then they are ignored. *First* and *last* are indices of characters within the item(s) as described in INDICES above. If *last* is omitted, it defaults to *first*. This command returns an empty string.

pathName **:delete** ?*tagOrId tagOrId ...*?
Delete each of the items given by each *tagOrId*, and return an empty string.

pathName **:dtag** *tagOrId* ?tagToDelete?
For each of the items given by *tagOrId*, delete the tag given by *tagToDelete* from the list of those associated with the item. If an item doesn't have the tag *tagToDelete* then the item is unaffected by the command. If *tagToDelete* is omitted then it defaults to *tagOrId*. This command returns an empty string.

pathName **:find** *searchCommand* ?*arg arg ...*?
This command returns a list consisting of all the items that meet the constraints specified by *searchCommand* and *arg*'s. *SearchCommand* and *args* have any of the forms accepted by the **addtag** command.

pathName **:focus** ?*tagOrId*?
Set the keyboard focus for the canvas widget to the item given by *tagOrId*. If *tagOrId* refers to several items, then the focus is set to the first such item in the display list that supports the insertion cursor. If *tagOrId* doesn't refer to any items, or if none of them support the insertion cursor, then the focus isn't

changed. If *tagOrId* is an empty string, then the focus item is reset so that no item has the focus. If *tagOrId* is not specified then the command returns the id for the item that currently has the focus, or an empty string if no item has the focus.

Once the focus has been set to an item, the item will display the insertion cursor and all keyboard events will be directed to that item. The focus item within a canvas and the focus window on the screen (set with the **focus** command) are totally independent: a given item doesn't actually have the input focus unless (a) its canvas is the focus window and (b) the item is the focus item within the canvas. In most cases it is advisable to follow the **focus** widget command with the **focus** command to set the focus window to the canvas (if it wasn't there already).

pathName **:gettags** *tagOrId*
Return a list whose elements are the tags associated with the item given by *tagOrId*. If *tagOrId* refers to more than one item, then the tags are returned from the first such item in the display list. If *tagOrId* doesn't refer to any items, or if the item contains no tags, then an empty string is returned.

pathName **:icursor** *tagOrId index*
Set the position of the insertion cursor for the item(s) given by *tagOrId* to just before the character whose position is given by *index*. If some or all of the items given by *tagOrId* don't support an insertion cursor then this command has no effect on them. See INDICES above for a description of the legal forms for *index*. Note: the insertion cursor is only displayed in an item if that item currently has the keyboard focus (see the widget command **focus**, below), but the cursor position may be set even when the item doesn't have the focus. This command returns an empty string.

pathName **:index** *tagOrId index*
This command returns a decimal string giving the numerical index within *tagOrId* corresponding to *index*. *Index* gives a textual description of the desired position as described in INDICES above. The return value is guaranteed to lie between 0 and the number of characters within the item, inclusive. If *tagOrId* refers to multiple items, then the index is processed in the first of these items that supports indexing operations (in display list order).

pathName **:insert** *tagOrId beforeThis string*
For each of the items given by *tagOrId*, if the item supports text insertion then *string* is inserted into the item's text just before the character whose index is *beforeThis*. See INDICES above for information about the forms allowed for *beforeThis*. This command returns an empty string.

pathName **:itemconfigure** *tagOrId* ?*option*? ?*value*? ?*option value ...*?
This command is similar to the **configure** widget command except that it modifies item-specific options for the items given by *tagOrId* instead of modifying options for the overall canvas widget. If no *option* is specified, returns a list describing all of the available options for the first item given by *tagOrId* (see **Tk_ConfigureInfo** for information on the format of this list). If *option* is specified with no *value*, then the command returns a list describing the one named

option (this list will be identical to the corresponding sublist of the value returned if no *option* is specified). If one or more *option:value* pairs are specified, then the command modifies the given widget option(s) to have the given value(s) in each of the items given by *tagOrId*; in this case the command returns an empty string. The *option*s and *value*s are the same as those permissible in the **create** widget command when the item(s) were created; see the sections describing individual item types below for details on the legal options.

pathName **:lower** *tagOrId* ?*belowThis*?
Move all of the items given by *tagOrId* to a new position in the display list just before the item given by *belowThis*. If *tagOrId* refers to more than one item then all are moved but the relative order of the moved items will not be changed. *BelowThis* is a tag or id; if it refers to more than one item then the first (lowest) of these items in the display list is used as the destination location for the moved items. This command returns an empty string.

pathName **:move** *tagOrId xAmount yAmount*
Move each of the items given by *tagOrId* in the canvas coordinate space by adding *xAmount* to the x-coordinate of each point associated with the item and *yAmount* to the y-coordinate of each point associated with the item. This command returns an empty string.

pathName **:postscript** ?*option value option value ...*?
Generate a Postscript representation for part or all of the canvas. If the **:file** option is specified then the Postscript is written to a file and an empty string is returned; otherwise the Postscript is returned as the result of the command. The Postscript is created in Encapsulated Postscript form using version 3.0 of the Document Structuring Conventions. The *option*\-*value* argument pairs provide additional information to control the generation of Postscript. The following options are supported:

:colormap *varName*
VarName must be the name of a global array variable that specifies a color mapping to use in the Postscript. Each element of *varName* must consist of Postscript code to set a particular color value (e.g. "**1.0 1.0 0.0 setrgbcolor**"). When outputting color information in the Postscript, Tk checks to see if there is an element of *varName* with the same name as the color. If so, Tk uses the value of the element as the Postscript command to set the color. If this option hasn't been specified, or if there isn't an entry in *varName* for a given color, then Tk uses the red, green, and blue intensities from the X color.

:colormode *mode*
Specifies how to output color information. *Mode* must be either **color** (for full color output), **gray** (convert all colors to their gray-scale equivalents) or **mono** (convert all colors to black or white).

:file *fileName*
Specifies the name of the file in which to write the Postscript. If this option isn't specified then the Postscript is returned as the result of the command instead of being written to a file.

:fontmap *varName*
VarName must be the name of a global array variable that specifies a font mapping to use in the Postscript. Each element of *varName* must consist of a Tcl list with two elements, which are the name and point size of a Postscript font. When outputting Postscript commands for a particular font, Tk checks to see if *varName* contains an element with the same name as the font. If there is such an element, then the font information contained in that element is used in the Postscript. Otherwise Tk attempts to guess what Postscript font to use. Tk's guesses generally only work for well-known fonts such as Times and Helvetica and Courier, and only if the X font name does not omit any dashes up through the point size. For example, \fB\-*\-Courier\-Bold\-R\-Normal\-\-*\-120\-* will work but \fB*Courier\-Bold\-R\-Normal*120* will not; Tk needs the dashes to parse the font name).

:height *size*
Specifies the height of the area of the canvas to print. Defaults to the height of the canvas window.

:pageanchor *anchor*
Specifies which point of the printed area should be appear over the positioning point on the page (which is given by the **:pagex** and **:pagey** options). For example, **:pageanchor n** means that the top center of the printed area should be over the positioning point. Defaults to **center**.

:pageheight *size*
Specifies that the Postscript should be scaled in both x and y so that the printed area is *size* high on the Postscript page. *Size* consists of a floating-point number followed by **c** for centimeters, **i** for inches, **m** for millimeters, or **p** or nothing for printer's points (1/72 inch). Defaults to the height of the printed area on the screen. If both **:pageheight** and **:pagewidth** are specified then the scale factor from the later option is used (non-uniform scaling is not implemented).

:pagewidth *size*
Specifies that the Postscript should be scaled in both x and y so that the printed area is *size* wide on the Postscript page. *Size* has the same form as for **:pageheight**. Defaults to the width of the printed area on the screen. If both **:pageheight** and **:pagewidth** are specified then the scale factor from the later option is used (non-uniform scaling is not implemented).

:pagex *position*

Position gives the x-coordinate of the positioning point on the Postscript page, using any of the forms allowed for **:pageheight**. Used in conjunction with the **:pagey** and **:pageanchor** options to determine where the printed area appears on the Postscript page. Defaults to the center of the page.

:pagey *position*

Position gives the y-coordinate of the positioning point on the Postscript page, using any of the forms allowed for **:pageheight**. Used in conjunction with the **:pagex** and **:pageanchor** options to determine where the printed area appears on the Postscript page. Defaults to the center of the page.

:rotate *boolean*

Boolean specifies whether the printed area is to be rotated 90 degrees. In non-rotated output the x-axis of the printed area runs along the short dimension of the page ("portrait" orientation); in rotated output the x-axis runs along the long dimension of the page ("landscape" orientation). Defaults to non-rotated.

:width *size*

Specifies the width of the area of the canvas to print. Defaults to the width of the canvas window.

:x *position* Specifies the x-coordinate of the left edge of the area of the canvas that is to be printed, in canvas coordinates, not window coordinates. Defaults to the coordinate of the left edge of the window.

:y *position* Specifies the y-coordinate of the top edge of the area of the canvas that is to be printed, in canvas coordinates, not window coordinates. Defaults to the coordinate of the top edge of the window.

pathName **:raise** *tagOrId* ?*aboveThis*?

Move all of the items given by *tagOrId* to a new position in the display list just after the item given by *aboveThis*. If *tagOrId* refers to more than one item then all are moved but the relative order of the moved items will not be changed. *AboveThis* is a tag or id; if it refers to more than one item then the last (topmost) of these items in the display list is used as the destination location for the moved items. This command returns an empty string.

pathName **:scale** *tagOrId xOrigin yOrigin xScale yScale*

Rescale all of the items given by *tagOrId* in canvas coordinate space. *XOrigin* and *yOrigin* identify the origin for the scaling operation and *xScale* and *yScale* identify the scale factors for x- and y-coordinates, respectively (a scale factor of 1.0 implies no change to that coordinate). For each of the points defining each item, the x-coordinate is adjusted to change the distance from *xOrigin* by a factor of *xScale*. Similarly, each y-coordinate is adjusted to change the distance from *yOrigin* by a factor of *yScale*. This command returns an empty string.

pathName **:scan** *option args*

This command is used to implement scanning on canvases. It has two forms, depending on *option*:

pathName **:scan :mark** *x y*

Records *x* and *y* and the canvas's current view; used in conjunction with later **scan dragto** commands. Typically this command is associated with a mouse button press in the widget and *x* and *y* are the coordinates of the mouse. It returns an empty string.

pathName **:scan :dragto** *x y*.

This command computes the difference between its *x* and *y* arguments (which are typically mouse coordinates) and the *x* and *y* arguments to the last **scan mark** command for the widget. It then adjusts the view by 10 times the difference in coordinates. This command is typically associated with mouse motion events in the widget, to produce the effect of dragging the canvas at high speed through its window. The return value is an empty string.

pathName **:select** *option* ?*tagOrId arg*?

Manipulates the selection in one of several ways, depending on *option*. The command may take any of the forms described below. In all of the descriptions below, *tagOrId* must refer to an item that supports indexing and selection; if it refers to multiple items then the first of these that supports indexing and the selection is used. *Index* gives a textual description of a position within *tagOrId*, as described in INDICES above.

pathName **:select :adjust** *tagOrId index*

Locate the end of the selection in *tagOrId* nearest to the character given by *index*, and adjust that end of the selection to be at *index* (i.e. including but not going beyond *index*). The other end of the selection is made the anchor point for future **select to** commands. If the selection isn't currently in *tagOrId* then this command behaves the same as the **select to** widget command. Returns an empty string.

pathName **:select :clear**

Clear the selection if it is in this widget. If the selection isn't in this widget then the command has no effect. Returns an empty string.

pathName **:select :from** *tagOrId index*

Set the selection anchor point for the widget to be just before the character given by *index* in the item given by *tagOrId*. This command doesn't change the selection; it just sets the fixed end of the selection for future **select to** commands. Returns an empty string.

pathName **:select :item**

Returns the id of the selected item, if the selection is in an item in this canvas. If the selection is not in this canvas then an empty string is returned.

pathName **:select :to** *tagOrId index*
Set the selection to consist of those characters of *tagOrId* between the selection anchor point and *index*. The new selection will include the character given by *index*; it will include the character given by the anchor point only if *index* is greater than or equal to the anchor point. The anchor point is determined by the most recent **select adjust** or **select from** command for this widget. If the selection anchor point for the widget isn't currently in *tagOrId*, then it is set to the same character given by *index*. Returns an empty string.

pathName **:type** *tagOrId*
Returns the type of the item given by *tagOrId*, such as **rectangle** or **text**. If *tagOrId* refers to more than one item, then the type of the first item in the display list is returned. If *tagOrId* doesn't refer to any items at all then an empty string is returned.

pathName **:xview** *index*
Change the view in the canvas so that the canvas position given by *index* appears at the left edge of the window. This command is typically used by scrollbars to scroll the canvas. *Index* counts in units of scroll increments (the value of the **scrollIncrement** option): a value of 0 corresponds to the left edge of the scroll region (as defined by the **scrollRegion** option), a value of 1 means one scroll unit to the right of this, and so on. The return value is an empty string.

pathName **:yview** *index*
Change the view in the canvas so that the canvas position given by *index* appears at the top edge of the window. This command is typically used by scrollbars to scroll the canvas. *Index* counts in units of scroll increments (the value of the **scrollIncrement** option): a value of 0 corresponds to the top edge of the scroll region (as defined by the **scrollRegion** option), a value of 1 means one scroll unit below this, and so on. The return value is an empty string.

Overview Of Item Types

The sections below describe the various types of items supported by canvas widgets. Each item type is characterized by two things: first, the form of the **create** command used to create instances of the type; and second, a set of configuration options for items of that type, which may be used in the **create** and **itemconfigure** widget commands. Most items don't support indexing or selection or the commands related to them, such as **index** and **insert**. Where items do support these facilities, it is noted explicitly in the descriptions below (at present, only text items provide this support).

Arc Items

Items of type **arc** appear on the display as arc-shaped regions. An arc is a section of an oval delimited by two angles (specified by the **:start** and **:extent** options) and displayed in one of several ways (specified by the **:style** option). Arcs are created with widget commands of the following form:

pathName **:create arc** *x1 y1 x2 y2 ?option value option value ...?*

The arguments *x1*, *y1*, *x2*, and *y2* give the coordinates of two diagonally opposite corners of a rectangular region enclosing the oval that defines the arc. After the coordinates there may be any number of *option-value* pairs, each of which sets one of the configuration options for the item. These same *option\-value* pairs may be used in **itemconfigure** widget commands to change the item's configuration. The following options are supported for arcs:

:extent *degrees*

Specifies the size of the angular range occupied by the arc. The arc's range extends for *degrees* degrees counter-clockwise from the starting angle given by the **:start** option. *Degrees* may be negative.

:fill *color* Fill the region of the arc with *color*. *Color* may have any of the forms accepted by **Tk_GetColor**. If *color* is an empty string (the default), then then the arc will not be filled.

:outline *color*

Color specifies a color to use for drawing the arc's outline; it may have any of the forms accepted by **Tk_GetColor**. This option defaults to **black**. If the arc's style is **arc** then this option is ignored (the section of perimeter is filled using the **:fill** option). If *color* is specified as an empty string then no outline is drawn for the arc.

:start *degrees*

Specifies the beginning of the angular range occupied by the arc. *Degrees* is given in units of degrees measured counter-clockwise from the 3-o'clock position; it may be either positive or negative.

:stipple *bitmap*

Indicates that the arc should be filled in a stipple pattern; *bitmap* specifies the stipple pattern to use, in any of the forms accepted by **Tk_GetBitmap**. If the **:fill** option hasn't been specified then this option has no effect. If *bitmap* is an empty string (the default), then filling is done in a solid fashion.

:style *type* Specifies how to draw the arc. If *type* is **pieslice** (the default) then the arc's region is defined by a section of the oval's perimeter plus two line segments, one between the center of the oval and each end of the perimeter section. If *type* is **chord** then the arc's region is defined by a section of the oval's perimeter plus a single line segment connecting the two end points of the perimeter section. If *type* is **arc** then the arc's region consists of a section of the perimeter alone. In this last case there is no outline for the arc and the **:outline** option is ignored.

:tags *tagList*

Specifies a set of tags to apply to the item. *TagList* consists of a list of tag names, which replace any existing tags for the item. *TagList* may be an empty list.

:width *outlineWidth*
Specifies the width of the outline to be drawn around the arc's region, in any of the forms described in the COORDINATES section above. If the **:outline** option has been specified as an empty string then this option has no effect. Wide outlines will be drawn centered on the edges of the arc's region. This option defaults to 1.0.

Bitmap Items

Items of type **bitmap** appear on the display as images with two colors, foreground and background. Bitmaps are created with widget commands of the following form:

pathName **:create bitmap** *x y ?option value option value ...?*
The arguments *x* and *y* specify the coordinates of a point used to position the bitmap on the display (see the **:anchor** option below for more information on how bitmaps are displayed). After the coordinates there may be any number of *option-value* pairs, each of which sets one of the configuration options for the item. These same *option\-value* pairs may be used in **itemconfigure** widget commands to change the item's configuration. The following options are supported for bitmaps:

:anchor *anchorPos*
AnchorPos tells how to position the bitmap relative to the positioning point for the item; it may have any of the forms accepted by **Tk_GetAnchor**. For example, if *anchorPos* is **center** then the bitmap is centered on the point; if *anchorPos* is **n** then the bitmap will be drawn so that its top center point is at the positioning point. This option defaults to **center**.

:background *color*
Specifies a color to use for each of the bitmap pixels whose value is 0. *Color* may have any of the forms accepted by **Tk_GetColor**. If this option isn't specified, or if it is specified as an empty string, then the background color for the canvas is used.

:bitmap *bitmap*
Specifies the bitmap to display in the item. *Bitmap* may have any of the forms accepted by **Tk_GetBitmap**.

:foreground *color*
Specifies a color to use for each of the bitmap pixels whose value is 1. *Color* may have any of the forms accepted by **Tk_GetColor** and defaults to **black**.

:tags *tagList*
Specifies a set of tags to apply to the item. *TagList* consists of a list of tag names, which replace any existing tags for the item. *TagList* may be an empty list.

Line Items

Items of type **line** appear on the display as one or more connected line segments or curves. Lines are created with widget commands of the following form:

pathName **:create line** *x1 y1... xn yn ?option value option value ...?*

The arguments *x1* through *yn* give the coordinates for a series of two or more points that describe a series of connected line segments. After the coordinates there may be any number of *option-value* pairs, each of which sets one of the configuration options for the item. These same *option\-value* pairs may be used in **itemconfigure** widget commands to change the item's configuration. The following options are supported for lines:

:arrow *where*

Indicates whether or not arrowheads are to be drawn at one or both ends of the line. *Where* must have one of the values **none** (for no arrowheads), **first** (for an arrowhead at the first point of the line), **last** (for an arrowhead at the last point of the line), or **both** (for arrowheads at both ends). This option defaults to **none**.

:arrowshape *shape*

This option indicates how to draw arrowheads. The *shape* argument must be a list with three elements, each specifying a distance in any of the forms described in the COORDINATES section above. The first element of the list gives the distance along the line from the neck of the arrowhead to its tip. The second element gives the distance along the line from the trailing points of the arrowhead to the tip, and the third element gives the distance from the outside edge of the line to the trailing points. If this option isn't specified then Tk picks a "reasonable" shape.

:capstyle *style*

Specifies the ways in which caps are to be drawn at the endpoints of the line. *Style* may have any of the forms accepted by **Tk_GetCapStyle** (**butt**, **projecting**, or **round**). If this option isn't specified then it defaults to **butt**. Where arrowheads are drawn the cap style is ignored.

:fill *color* *Color* specifies a color to use for drawing the line; it may have any of the forms acceptable to **Tk_GetColor**. It may also be an empty string, in which case the line will be transparent. This option defaults to **black**.

:joinstyle *style*

Specifies the ways in which joints are to be drawn at the vertices of the line. *Style* may have any of the forms accepted by **Tk_GetCapStyle** (**bevel**, **miter**, or **round**). If this option isn't specified then it defaults to **miter**. If the line only contains two points then this option is irrelevant.

:smooth *boolean*

Boolean must have one of the forms accepted by **Tk_GetBoolean**. It indicates whether or not the line should be drawn as a curve. If so, the line is rendered as a set of Bezier splines: one spline is drawn for the first and second line segments, one for the second and third, and so on. Straight-line segments can be generated within a curve by duplicating the end-points of the desired line segment.

:splinesteps *number*

Specifies the degree of smoothness desired for curves: each spline will be approximated with *number* line segments. This option is ignored unless the **:smooth** option is true.

:stipple *bitmap*

Indicates that the line should be filled in a stipple pattern; *bitmap* specifies the stipple pattern to use, in any of the forms accepted by **Tk_GetBitmap**. If *bitmap* is an empty string (the default), then filling is done in a solid fashion.

:tags *tagList*

Specifies a set of tags to apply to the item. *TagList* consists of a list of tag names, which replace any existing tags for the item. *TagList* may be an empty list.

:width *lineWidth*

LineWidth specifies the width of the line, in any of the forms described in the COORDINATES section above. Wide lines will be drawn centered on the path specified by the points. If this option isn't specified then it defaults to 1.0.

Oval Items

Items of type **oval** appear as circular or oval regions on the display. Each oval may have an outline, a fill, or both. Ovals are created with widget commands of the following form:

pathName **:create oval** *x1 y1 x2 y2 ?option value option value ...?*

The arguments *x1*, *y1*, *x2*, and *y2* give the coordinates of two diagonally opposite corners of a rectangular region enclosing the oval. The oval will include the top and left edges of the rectangle not the lower or right edges. If the region is square then the resulting oval is circular; otherwise it is elongated in shape. After the coordinates there may be any number of *option-value* pairs, each of which sets one of the configuration options for the item. These same *option\-value* pairs may be used in **itemconfigure** widget commands to change the item's configuration. The following options are supported for ovals:

:fill *color* Fill the area of the oval with *color*. *Color* may have any of the forms accepted by **Tk_GetColor**. If *color* is an empty string (the default), then then the oval will not be filled.

:outline *color*

Color specifies a color to use for drawing the oval's outline; it may have any of the forms accepted by **Tk_GetColor**. This option de-

faults to **black**. If *color* is an empty string then no outline will be drawn for the oval.

:stipple *bitmap*
Indicates that the oval should be filled in a stipple pattern; *bitmap* specifies the stipple pattern to use, in any of the forms accepted by **Tk_GetBitmap**. If the **:fill** option hasn't been specified then this option has no effect. If *bitmap* is an empty string (the default), then filling is done in a solid fashion.

:tags *tagList*
Specifies a set of tags to apply to the item. *TagList* consists of a list of tag names, which replace any existing tags for the item. *TagList* may be an empty list.

:width *outlineWidth*
outlineWidth specifies the width of the outline to be drawn around the oval, in any of the forms described in the COORDINATES section above. If the **:outline** option hasn't been specified then this option has no effect. Wide outlines are drawn centered on the oval path defined by *x1*, *y1*, *x2*, and *y2*. This option defaults to 1.0.

Polygon Items

Items of type **polygon** appear as polygonal or curved filled regions on the display. Polygons are created with widget commands of the following form:

pathName **:create polygon** *x1 y1 ... xn yn ?option value option value ...?*
The arguments *x1* through *yn* specify the coordinates for three or more points that define a closed polygon. The first and last points may be the same; whether they are or not, Tk will draw the polygon as a closed polygon. After the coordinates there may be any number of *option-value* pairs, each of which sets one of the configuration options for the item. These same *option\-value* pairs may be used in **itemconfigure** widget commands to change the item's configuration. The following options are supported for polygons:

:fill *color* *Color* specifies a color to use for filling the area of the polygon; it may have any of the forms acceptable to **Tk_GetColor**. If *color* is an empty string then the polygon will be transparent. This option defaults to **black**.

:smooth *boolean*
Boolean must have one of the forms accepted by **Tk_GetBoolean** It indicates whether or not the polygon should be drawn with a curved perimeter. If so, the outline of the polygon becomes a set of Bezier splines, one spline for the first and second line segments, one for the second and third, and so on. Straight-line segments can be generated in a smoothed polygon by duplicating the end-points of the desired line segment.

:splinesteps *number*
Specifies the degree of smoothness desired for curves: each spline will be approximated with *number* line segments. This option is ignored unless the **:smooth** option is true.

:stipple *bitmap*
Indicates that the polygon should be filled in a stipple pattern; *bitmap* specifies the stipple pattern to use, in any of the forms accepted by **Tk_GetBitmap**. If *bitmap* is an empty string (the default), then filling is done in a solid fashion.

:tags *tagList*
Specifies a set of tags to apply to the item. *TagList* consists of a list of tag names, which replace any existing tags for the item. *TagList* may be an empty list.

Rectangle Items

Items of type **rectangle** appear as rectangular regions on the display. Each rectangle may have an outline, a fill, or both. Rectangles are created with widget commands of the following form:

pathName **:create rectangle** *x1 y1 x2 y2 ?option value option value ...?*
The arguments *x1*, *y1*, *x2*, and *y2* give the coordinates of two diagonally opposite corners of the rectangle (the rectangle will include its upper and left edges but not its lower or right edges). After the coordinates there may be any number of *option-value* pairs, each of which sets one of the configuration options for the item. These same *option\-value* pairs may be used in **itemconfigure** widget commands to change the item's configuration. The following options are supported for rectangles:

:fill *color* Fill the area of the rectangle with *color*, which may be specified in any of the forms accepted by **Tk_GetColor**. If *color* is an empty string (the default), then then the rectangle will not be filled.

:outline *color*
Draw an outline around the edge of the rectangle in *color*. *Color* may have any of the forms accepted by **Tk_GetColor**. This option defaults to **black**. If *color* is an empty string then no outline will be drawn for the rectangle.

:stipple *bitmap*
Indicates that the rectangle should be filled in a stipple pattern; *bitmap* specifies the stipple pattern to use, in any of the forms accepted by **Tk_GetBitmap**. If the **:fill** option hasn't been specified then this option has no effect. If *bitmap* is an empty string (the default), then filling is done in a solid fashion.

:tags *tagList*
Specifies a set of tags to apply to the item. *TagList* consists of a list of tag names, which replace any existing tags for the item. *TagList* may be an empty list.

:width *outlineWidth*

OutlineWidth specifies the width of the outline to be drawn around the rectangle, in any of the forms described in the COORDINATES section above. If the **:outline** option hasn't been specified then this option has no effect. Wide outlines are drawn centered on the rectangular path defined by *x1*, *y1*, *x2*, and *y2*. This option defaults to 1.0.

Text Items

A text item displays a string of characters on the screen in one or more lines. Text items support indexing and selection, along with the following text-related canvas widget commands: **dchars**, **focus**, **icursor**, **index**, **insert**, **select**. Text items are created with widget commands of the following form:

pathName **:create text** *x y ?option value option value ...?*

The arguments *x* and *y* specify the coordinates of a point used to position the text on the display (see the options below for more information on how text is displayed). After the coordinates there may be any number of *option-value* pairs, each of which sets one of the configuration options for the item. These same *option*\-*value* pairs may be used in **itemconfigure** widget commands to change the item's configuration. The following options are supported for text items:

:anchor *anchorPos*

AnchorPos tells how to position the text relative to the positioning point for the text; it may have any of the forms accepted by **Tk_GetAnchor**. For example, if *anchorPos* is **center** then the text is centered on the point; if *anchorPos* is **n** then the text will be drawn such that the top center point of the rectangular region occupied by the text will be at the positioning point. This option defaults to **center**.

:fill *color* *Color* specifies a color to use for filling the text characters; it may have any of the forms accepted by **Tk_GetColor**. If this option isn't specified then it defaults to **black**.

:font *fontName*

Specifies the font to use for the text item. *FontName* may be any string acceptable to **Tk_GetFontStruct**. If this option isn't specified, it defaults to a system-dependent font.

:justify *how*

Specifies how to justify the text within its bounding region. *How* must be one of the values **left**, **right**, or **center**. This option will only matter if the text is displayed as multiple lines. If the option is omitted, it defaults to **left**.

:stipple *bitmap*

Indicates that the text should be drawn in a stippled pattern rather than solid; *bitmap* specifies the stipple pattern to use, in any of the

forms accepted by **Tk_GetBitmap**. If *bitmap* is an empty string (the default) then the text is drawn in a solid fashion.

:tags *tagList*
Specifies a set of tags to apply to the item. *TagList* consists of a list of tag names, which replace any existing tags for the item. *TagList* may be an empty list.

:text *string*
String specifies the characters to be displayed in the text item. Newline characters cause line breaks. The characters in the item may also be changed with the **insert** and **delete** widget commands. This option defaults to an empty string.

:width *lineLength*
Specifies a maximum line length for the text, in any of the forms described in the COORDINATES section abov. If this option is zero (the default) the text is broken into lines only at newline characters. However, if this option is non-zero then any line that would be longer than *lineLength* is broken just before a space character to make the line shorter than *lineLength*; the space character is treated as if it were a newline character.

Window Items

Items of type **window** cause a particular window to be displayed at a given position on the canvas. Window items are created with widget commands of the following form:

pathName **:create window** *x y ?option value option value ...*?

The arguments *x* and *y* specify the coordinates of a point used to position the window on the display (see the **:anchor** option below for more information on how bitmaps are displayed). After the coordinates there may be any number of *option-value* pairs, each of which sets one of the configuration options for the item. These same *option\-value* pairs may be used in **itemconfigure** widget commands to change the item's configuration. The following options are supported for window items:

:anchor *anchorPos*
AnchorPos tells how to position the window relative to the positioning point for the item; it may have any of the forms accepted by **Tk_GetAnchor**. For example, if *anchorPos* is **center** then the window is centered on the point; if *anchorPos* is **n** then the window will be drawn so that its top center point is at the positioning point. This option defaults to **center**.

:height *pixels*
Specifies the height to assign to the item's window. *Pixels* may have any of the forms described in the COORDINATES section above. If this option isn't specified, or if it is specified as an empty string, then the window is given whatever height it requests internally.

:tags *tagList*
: Specifies a set of tags to apply to the item. *TagList* consists of a list of tag names, which replace any existing tags for the item. *TagList* may be an empty list.

:width *pixels*
: Specifies the width to assign to the item's window. *Pixels* may have any of the forms described in the COORDINATES section above. If this option isn't specified, or if it is specified as an empty string, then the window is given whatever width it requests internally.

:window *pathName*
: Specifies the window to associate with this item. The window specified by *pathName* must either be a child of the canvas widget or a child of some ancestor of the canvas widget. *PathName* may not refer to a top-level window.

Application-Defined Item Types

It is possible for individual applications to define new item types for canvas widgets using C code. The interfaces for this mechanism are not presently documented, and it's possible they may change, but you should be able to see how they work by examining the code for some of the existing item types.

Bindings

In the current implementation, new canvases are not given any default behavior: you'll have to execute explicit Tcl commands to give the canvas its behavior.

Credits

Tk's canvas widget is a blatant ripoff of ideas from Joel Bartlett's *ezd* program. *Ezd* provides structured graphics in a Scheme environment and preceded canvases by a year or two. Its simple mechanisms for placing and animating graphical objects inspired the functions of canvases.

Keywords

canvas, widget

2.5 menu

menu \- Create and manipulate menu widgets

Synopsis

menu *pathName* ?*options*?

Standard Options

activeBackground	background	disabledForeground
activeBorderWidth	borderWidth	font
activeForeground	cursor	foreground

See ⟨undefined⟩ [options], page ⟨undefined⟩, for more information.

Arguments for Menu

:postcommand

Name="**postCommand**" Class="**Command**"

If this option is specified then it provides a Tcl command to execute each time the menu is posted. The command is invoked by the **post** widget command before posting the menu.

:selector

Name="**selector**" Class="**Foreground**"

For menu entries that are check buttons or radio buttons, this option specifies the color to display in the selector when the check button or radio button is selected.

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 text or a bitmap, which is determined by the **:label** or **:bitmap** option for the entry. If the **:accelerator** option is specified for an entry then a second textual field is displayed to the right of the label. The accelerator typically describes a keystroke sequence that may be typed in the application to cause the same result as invoking the menu entry. The third field is a *selector*. The selector is present only for check-button or radio-button entries. It indicates whether the entry is selected or not, and is displayed to the left of the entry's string.

In normal use, an entry becomes active (displays itself differently) whenever the mouse pointer is over the entry. If a mouse button is released over the entry then the entry is *invoked*. The effect of invocation is different for each type of entry; these effects are described below in the sections on individual entries.

Entries may be *disabled*, which causes their labels and accelerators to be displayed with dimmer colors. A disabled entry cannot 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.

Check-Button Entries

A check-button menu entry behaves much like a check-button 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. A selector box is displayed to the left of the label in a check-button entry. If the entry is selected then the box's center is displayed in the color given by the **selector** option for the menu; otherwise the box's center is displayed in the background color for the menu. If a **:command** option is specified for a check-button 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.

Radio-Button Entries

A radio-button menu entry behaves much like a radio-button widget. Radio-button 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 radio-button entries is determined by their associated variables: if two entries have the same associated variable then they are in the same group. A selector diamond is displayed to the left of the label in each radio-button entry. If the entry is selected then the diamond's center is displayed in the color given by the **selector** option for the menu; otherwise the diamond's center is displayed in the background color for the menu. If a **:command** option is specified for a radio-button 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. When the entry is activated,

the associated menu is posted just to the right of the entry; that menu remains posted until the higher-level menu is unposted or until some other entry is activated in the higher-level menu. The associated menu should normally be a child of the menu containing the cascade entry, 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, *x* and *y* are the root-window coordinates of the upper-right corner of the cascade entry, and *group* is the name of the menu's group (as determined in its last **post** widget command). 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 each time the associated menu is posted (the evaluation occurs before the menu is posted).

A Menu Widget's Arguments

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.

last 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.

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 spanning that y-coordinate is used. For example, "**`@0`**" indicates the top-most entry in the window. If *number* is outside the range of the window then this form is equivalent to **none**.

pattern If the index doesn't satisfy one of the above forms then this form is used. *Pattern* is pattern-matched against the label of each entry in the menu, in order from

the top down, until a matching entry is found. The rules of **Tcl_StringMatch** are used.

The following widget commands are possible for menu widgets:

pathName **:activate** *index*
Change the state of the entry indicated by *index* to **active** and redisplay it using its active colors. Any previously-active entry is deactivated. If *index* is specified as **none**, or if the specified entry is disabled, then the menu ends up with no active entry. Returns an empty string.

pathName **:add** *type ?option value option value ...?*
Add a new entry to the bottom of the menu. The new entry's type is given by *type* and must be one of **cascade**, **checkbutton**, **command**, **radiobutton**, or **separator**, or a unique abbreviation of one of the above. If additional arguments are present, they specify any of the following options:

:activebackground *value*
Specifies a background color to use for displaying this entry when it is active. If this option is specified as an empty string (the default), then the **activeBackground** option for the overall menu is used. 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.

: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.

:bitmap *value*
Specifies a bitmap to display in the menu instead of a textual label, in any of the forms accepted by **Tk_GetBitmap**. This option overrides the **:label** option but may be reset to an empty string to enable a textual label to be displayed. 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.

:font *value*
Specifies the font to use when drawing the label or accelerator string in this entry. If this option is specified as an empty string (the

default) then the **font** option for the overall menu is used. This option is not available for separator entries.

: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 menu associated with this entry.

:offvalue *value*
Available only for check-button entries. Specifies the value to store in the entry's associated variable when the entry is deselected.

:onvalue *value*
Available only for check-button entries. Specifies the value to store in the entry's associated variable when the entry is selected.

:state *value*
Specifies one of three states for the entry: **normal**, **active**, or **disabled**. In normal state the entry is displayed using the **foreground** option for the menu and the **background** option from the entry or the menu. The active state is typically used when the pointer is over the entry. In active state the entry is displayed using the **activeForeground** option for the menu along with the **activebackground** option from the entry. Disabled state means that the entry is insensitive: it doesn't activate and doesn't respond to mouse button presses or releases. In this state the entry is displayed according to the **disabledForeground** option for the menu and the **background** option from the entry. This option is not available for separator entries.

:underline *value*
Specifies the integer index of a character to underline in the entry. This option is typically used to indicate keyboard traversal characters. 0 corresponds to the first character of the text displayed in the entry, 1 to the next character, and so on. If a bitmap is displayed in the entry then this option is ignored. This option is not available for separator entries.

:value *value*
Available only for radio-button entries. Specifies the value to store in the entry's associated variable when the entry is selected.

:variable *value*
Available only for check-button and radio-button entries. Specifies the name of a global value to set when the entry is selected. For check-button entries the variable is also set when the entry is deselected. For radio-button entries, changing the variable causes the currently-selected entry to deselect itself.

The **add** widget command returns an empty string.

pathName **:configure** ?*option*? ?*value option value ...*?
Query or modify the configuration options of the widget. If no *option* is specified, returns a list describing all of the available options for *pathName* (see **Tk_ConfigureInfo** for information on the format of this list). If *option* is specified with no *value*, then the command returns a list describing the one named option (this list will be identical to the corresponding sublist of the value returned if no *option* is specified). If one or more *option:value* pairs are specified, then the command modifies the given widget option(s) to have the given value(s); in this case the command returns an empty string. *Option* may have any of the values accepted by the **menu** command.

pathName **:delete** *index1* ?*index2*?
Delete all of the menu entries between *index1* and *index2* inclusive. If *index2* is omitted then it defaults to *index1*. Returns an empty string.

pathName **:disable** *index*
Change the state of the entry given by *index* to **disabled** and redisplay the entry using its disabled colors. Returns an empty string. This command is obsolete and will eventually be removed; use "*pathName* **:entryconfigure** *index* :state disabled" instead.

pathName **:enable** *index*
Change the state of the entry given by *index* to **normal** and redisplay the entry using its normal colors. Returns an empty string. This command is obsolete and will eventually be removed; use "*pathName* **:entryconfigure** *index* :state normal" instead.

pathName **:entryconfigure** *index* ?*options*?
This command is similar to the **configure** command, except that it applies to the options for an individual entry, whereas **configure** applies to the options for the menu as a whole. *Options* may have any of the values accepted by the **add** widget command. If *options* are specified, options are modified as indicated in the command and the command returns an empty string. If no *options* are specified, returns a list describing the current options for entry *index* (see **Tk_ConfigureInfo** for information on the format of this list).

pathName **:index** *index*
Returns the numerical index corresponding to *index*, or **none** if *index* was specified as **none**.

pathName **: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. Normally the associated menubutton will take care of unposting the menu.

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 **: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 topmost pixel in the entry specified by *index*.

Default Bindings

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

[1] When the mouse cursor enters a menu, the entry underneath the mouse cursor is activated; as the mouse moves around the menu, the active entry changes to track the mouse.

[2] When button 1 is released over a menu, the active entry (if any) is invoked.

[3] A menu can be repositioned on the screen by dragging it with mouse button 2.

[4] A number of other bindings are created to support keyboard menu traversal. See the manual entry for **tk_bindForTraversal** for details on these bindings.

Disabled menu entries are non-responsive: they don't activate and 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

2.6 scrollbar

scrollbar \- Create and manipulate scrollbar widgets

Synopsis

scrollbar *pathName ?options?*

Standard Options

activeForeground	cursor	relief
background	foreground	repeatDelay
borderWidth	orient	repeatInterval

See ⟨undefined⟩ [options], page ⟨undefined⟩, for more information.

Arguments for Scrollbar

:command

Name="**command**" Class="**Command**"

Specifies the prefix of a Tcl command to invoke to change the view in the widget associated with the scrollbar. When a user requests a view change by manipulating the scrollbar, a Tcl command is invoked. The actual command consists of this option followed by a space and a number. The number indicates the logical unit that should appear at the top of the associated window.

:width

Name="**width**" Class="**Width**"

Specifies the desired narrow dimension of the scrollbar window, not including 3-D border, if any. For vertical scrollbars this will be the width and for horizontal scrollbars this will be the height. The value may have any of the forms acceptable to **Tk_GetPixels**.

Description

The **scrollbar** command creates a new window (given by the *pathName* argument) and makes it into a scrollbar widget. Additional options, described above, may be specified on the command line or in the option database to configure aspects of the scrollbar such as its colors, orientation, and relief. The **scrollbar** command returns its *pathName* argument. At the time this command is invoked, there must not exist a window named *pathName*, but *pathName*'s parent must exist.

A scrollbar is a widget that displays two arrows, one at each end of the scrollbar, and a *slider* in the middle portion of the scrollbar. A scrollbar is used to provide information about what is visible in an *associated window* that displays an object of some sort (such as a file being edited or a drawing). The position and size of the slider indicate which portion of the object 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 object.

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.

A Scrollbar Widget's Arguments

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 **:configure** ?*option*? ?*value option value ...*?
Query or modify the configuration options of the widget. If no *option* is specified, returns a list describing all of the available options for *pathName* (see **Tk_ConfigureInfo** for information on the format of this list). If *option* is specified with no *value*, then the command returns a list describing the one named option (this list will be identical to the corresponding sublist of the value returned if no *option* is specified). If one or more *option:value* pairs are specified, then the command modifies the given widget option(s) to have the given value(s); in this case the command returns an empty string. *Option* may have any of the values accepted by the **scrollbar** command.

pathName **:get**
Returns a Tcl list containing four decimal values, which are the current *totalUnits*, *widnowUnits*, *firstUnit*, and *lastUnit* values for the scrollbar. These are the values from the most recent **set** widget command on the scrollbar.

pathName **:set** *totalUnits windowUnits firstUnit lastUnit*
This command is invoked to give the scrollbar information about the widget associated with the scrollbar. *TotalUnits* is an integer value giving the total size of the object being displayed in the associated widget. The meaning of one unit depends on the associated widget; for example, in a text editor widget units might correspond to lines of text. *WindowUnits* indicates the total number of units that can fit in the associated window at one time. *FirstUnit* and *lastUnit* give the indices of the first and last units currently visible in the associated window (zero corresponds to the first unit of the object). This command should be invoked by the associated widget whenever its object or window changes size and whenever it changes the view in its window.

Bindings

The description below assumes a vertically-oriented scrollbar. For a horizontally-oriented scrollbar replace the words "up", "down", "top", and "bottom" with "left", "right", "left", and "right", respectively

A scrollbar widget is divided into five distinct areas. From top to bottom, they are: the top arrow, the top gap (the empty space between the arrow and the slider), the slider, the bottom gap, and the bottom arrow. Pressing mouse button 1 in each area has a different effect:

top arrow Causes the view in the associated window to shift up by one unit (i.e. the object appears to move down one unit in its window). If the button is held down the action will auto-repeat.

top gap Causes the view in the associated window to shift up by one less than the number of units in the window (i.e. the portion of the object that used to appear at the very top of the window will now appear at the very bottom). If the button is held down the action will auto-repeat.

slider Pressing button 1 in this area has no immediate effect except to cause the slider to appear sunken rather than raised. However, if the mouse is moved with the button down then the slider will be dragged, adjusting the view as the mouse is moved.

bottom gap
Causes the view in the associated window to shift down by one less than the number of units in the window (i.e. the portion of the object that used to appear at the very bottom of the window will now appear at the very top). If the button is held down the action will auto-repeat.

bottom arrow
Causes the view in the associated window to shift down by one unit (i.e. the object appears to move up one unit in its window). If the button is held down the action will auto-repeat.

Note: none of the actions described above has an immediate impact on the position of the slider in the scrollbar. It simply invokes the command specified in the **command** option to notify the associated widget that a change in view is desired. If the view is actually changed then the associated widget must invoke the scrollbar's **set** widget command to change what is displayed in the scrollbar.

Keywords

scrollbar, widget

2.7 checkbutton

checkbutton \- Create and manipulate check-button widgets

Synopsis

checkbutton *pathName* ?*options*?

Standard Options

activeBackground	bitmap	font	relief
activeForeground	borderWidth	foreground	text
anchor	cursor	padX	textVariable
background	disabledForeground	padY	

See ⟨undefined⟩ [options], page ⟨undefined⟩, for more information.

Arguments for Checkbutton

:command

Name="**command**" Class="**Command**"

Specifies a Tcl command to associate with the button. This command is typically invoked when mouse button 1 is released over the button window. The button's global variable (**:variable** option) will be updated before the command is invoked.

:height

Name="**height**" Class="**Height**"

Specifies a desired height for the button. If a bitmap is being displayed in the button then the value is in screen units (i.e. any of the forms acceptable to **Tk_GetPixels**); for text it is in lines of text. If this option isn't specified, the button's desired height is computed from the size of the bitmap or text being displayed in it.

:offvalue

Name="**offValue**" Class="**Value**"

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

:onvalue

Name="**onValue**" Class="**Value**"

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

:selector

Name="**selector**" Class="**Foreground**"

Specifies the color to draw in the selector when this button is selected. If specified as an empty string then no selector is drawn for the button.

:state

Name="**state**" Class="**State**"

Specifies one of three states for the check button: **normal**, **active**, or **disabled**. In normal state the check button is displayed using the **foreground** and **background** options. The active state is typically used when the pointer is over the check button. In active state the check button is displayed using the **activeForeground** and **activeBackground** options. Disabled state means that the check button is insensitive: it doesn't activate and doesn't respond to mouse button presses. In this state the **disabledForeground** and **background** options determine how the check button is displayed.

:variable

Name="**variable**" Class="**Variable**"

Specifies name of global variable to set to indicate whether or not this button is selected. Defaults to the name of the button within its parent (i.e. the last element of the button window's path name).

:width

Name="**width**" Class="**Width**"

Specifies a desired width for the button. If a bitmap is being displayed in the button then the value is in screen units (i.e. any of the forms acceptable to **Tk_GetPixels**); for text it is in characters. If this option isn't specified, the button's desired width is computed from the size of the bitmap or text being displayed in it.

Description

The **checkbutton** command creates a new window (given by the *pathName* argument) and makes it into a check-button widget. Additional options, described above, may be specified on the command line or in the option database to configure aspects of the check button 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 check button is a widget that displays a textual string or bitmap and a square called a *selector*. A check button has all of the behavior of a simple button, including the following: it can display itself in either of three different ways, according to the **state** option; it can be made to appear raised, sunken, or flat; it can be made to flash; and it invokes a Tcl command whenever mouse button 1 is clicked over the check button.

In addition, check buttons can be *selected*. If a check button is selected then a special highlight appears in the selector, and a Tcl variable associated with the check button is set to a particular value (normally 1). If the check button is not selected, then the selector is drawn in a different fashion and the associated variable is set to a different value (typically 0). By default, the name of the variable associated with a check button is the same as the *name* used to create the check button. 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 check button is configured to select and deselect itself on alternate button clicks. In addition, each check button monitors its associated variable and automatically selects and deselects itself when the variables value changes to and from the button's "on" value.

A Checkbutton Widget's Arguments

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 check button widgets:

pathName **:activate**
Change the check button's state to **active** and redisplay the button using its active foreground and background colors instead of normal colors. This command is ignored if the check button's state is **disabled**. This command is obsolete and will eventually be removed; use "*pathName* **:configure :state active**" instead.

pathName **:configure** *?option? ?value option value ...?*
Query or modify the configuration options of the widget. If no *option* is specified, returns a list describing all of the available options for *pathName* (see **Tk_ConfigureInfo** for information on the format of this list). If *option* is specified with no *value*, then the command returns a list describing the one named option (this list will be identical to the corresponding sublist of the value returned if no *option* is specified). If one or more *option:value* pairs are specified, then the command modifies the given widget option(s) to have the given value(s); in this case the command returns an empty string. *Option* may have any of the values accepted by the **checkbutton** command.

pathName **:deactivate**
Change the check button's state to **normal** and redisplay the button using its normal foreground and background colors. This command is ignored if the check button's state is **disabled**. This command is obsolete and will eventually be removed; use "*pathName* **:configure :state normal**" instead.

pathName **:deselect**
Deselect the check button: redisplay it without a highlight in the selector and set the associated variable to its "off" value.

pathName **:flash**
Flash the check button. This is accomplished by redisplaying the check button several times, alternating between active and normal colors. At the end of the flash the check button is left in the same normal/active state as when the command was invoked. This command is ignored if the check button's state is **disabled**.

pathName **:invoke**
Does just what would have happened if the user invoked the check button with the mouse: toggle the selection state of the button and invoke the Tcl command associated with the check 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 check button. This command is ignored if the check button's state is **disabled**.

pathName **:select**
Select the check button: display it with a highlighted selector and set the associated variable to its "on" value.

pathName **:toggle**
Toggle the selection state of the button, redisplaying it and modifying its associated variable to reflect the new state.

Bindings

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

[1] The check button activates whenever the mouse passes over it and deactivates whenever the mouse leaves the check button.

[2] The check button's relief is changed to sunken whenever mouse button 1 is pressed over it, and the relief is restored to its original value when button 1 is later released.

[3] If mouse button 1 is pressed over the check button and later released over the check button, the check button is invoked (i.e. its selection state toggles and the command associated with the button is invoked, if there is one). However, if the mouse is not over the check button when button 1 is released, then no invocation occurs.

If the check button's state is **disabled** then none of the above actions occur: the check button is completely non-responsive.

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

Keywords

check button, widget

2.8 menubutton

menubutton \- Create and manipulate menubutton widgets

Synopsis

menubutton *pathName* ?*options*?

Standard Options

`activeBackground`	`bitmap`	`font`	`relief`
`activeForeground`	`borderWidth`	`foreground`	`text`
`anchor`	`cursor`	`padX`	`textVariable`
`background`	`disabledForeground`	`padY`	`underline`

See ⟨undefined⟩ [options], page ⟨undefined⟩, for more information.

Arguments for Menubutton

:height

Name="**height**" Class="**Height**"

Specifies a desired height for the menu button. If a bitmap is being displayed in the menu button then the value is in screen units (i.e. any of the forms acceptable to **Tk_GetPixels**); for text it is in lines of text. If this option isn't specified, the menu button's desired height is computed from the size of the bitmap or text being displayed in it.

:menu

Name="**menu**" Class="**MenuName**"

Specifies the path name of the menu associated with this menubutton. The menu must be a descendant of the menubutton in order for normal pull-down operation to work via the mouse.

:state

Name="**state**" Class="**State**"

Specifies one of three states for the menu button: **normal**, **active**, or **disabled**. In normal state the menu button is displayed using the **foreground** and **background** options. The active state is typically used when the pointer is over the menu button. In active state the menu button is displayed using the **activeForeground** and **activeBackground** options. Disabled state means that the menu button is insensitive: it doesn't activate and doesn't respond to mouse button presses. In this state the **disabledForeground** and **background** options determine how the button is displayed.

:width

Name="**width**" Class="**Width**"

Specifies a desired width for the menu button. If a bitmap is being displayed in the menu button then the value is in screen units (i.e. any of the forms acceptable to **Tk_GetPixels**); for text it is in characters. If this option isn't specified, the menu button's desired width is computed from the size of the bitmap or text being displayed in it.

Introduction

The **menubutton** command creates a new window (given by the *pathName* argument) and makes it into a menubutton widget. Additional options, described above, may be specified on the command line or in the option database to configure aspects of the menubutton such as its colors, font, text, and initial relief. The **menubutton** command returns its *pathName* argument. At the time this command is invoked, there must not exist a window named *pathName*, but *pathName*'s parent must exist.

A menubutton is a widget that displays a textual string or bitmap and is associated with a menu widget. In normal usage, pressing mouse button 1 over the menubutton causes the associated menu to be posted just underneath the menubutton. If the mouse is moved over

the menu before releasing the mouse button, the button release causes the underlying menu entry to be invoked. When the button is released, the menu is unposted.

Menubuttons are typically organized into groups called menu bars that allow scanning: if the mouse button is pressed over one menubutton (causing it to post its menu) and the mouse is moved over another menubutton in the same menu bar without releasing the mouse button, then the menu of the first menubutton is unposted and the menu of the new menubutton is posted instead. The **tk-menu-bar** procedure is used to set up menu bars for scanning; see that procedure for more details.

A Menubutton Widget's Arguments

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 **:activate**
Change the menu button's state to **active** and redisplay the menu button using its active foreground and background colors instead of normal colors. The command returns an empty string. This command is ignored if the menu button's state is **disabled**. This command is obsolete and will eventually be removed; use "*pathName* **:configure :state active**" instead.

pathName **:configure** *?option? ?value option value ...?*
Query or modify the configuration options of the widget. If no *option* is specified, returns a list describing all of the available options for *pathName* (see **Tk_ConfigureInfo** for information on the format of this list). If *option* is specified with no *value*, then the command returns a list describing the one named option (this list will be identical to the corresponding sublist of the value returned if no *option* is specified). If one or more *option:value* pairs are specified, then the command modifies the given widget option(s) to have the given value(s); in this case the command returns an empty string. *Option* may have any of the values accepted by the **menubutton** command.

pathName **:deactivate**
Change the menu button's state to **normal** and redisplay the menu button using its normal foreground and background colors. The command returns an empty string. This command is ignored if the menu button's state is **disabled**. This command is obsolete and will eventually be removed; use "*pathName* **:configure :state normal**" instead.

"Default Bindings"

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

[1] A menu button activates whenever the mouse passes over it and deactivates whenever the mouse leaves it.

[2] A menu button's relief is changed to raised whenever mouse button 1 is pressed over it, and the relief is restored to its original value when button 1 is later released or the mouse is dragged into another menu button in the same menu bar.

[3] When mouse button 1 is pressed over a menu button, or when the mouse is dragged into a menu button with mouse button 1 pressed, the associated menu is posted; the mouse can be dragged across the menu and released over an entry in the menu to invoke that entry. The menu is unposted when button 1 is released outside either the menu or the menu button. The menu is also unposted when the mouse is dragged into another menu button in the same menu bar.

[4] If mouse button 1 is pressed and released within the menu button, then the menu stays posted and keyboard traversal is possible as described in the manual entry for **tk-menu-bar**.

[5] Menubuttons may also be posted by typing characters on the keyboard. See the manual entry for **tk-menu-bar** for full details on keyboard menu traversal.

[6] If mouse button 2 is pressed over a menu button then the associated menu is posted and also *torn off*: it can then be dragged around on the screen with button 2 and the menu will not automatically unpost when entries in it are invoked. To close a torn off menu, click mouse button 1 over the associated menu button.

If the menu button's state is **disabled** then none of the above actions occur: the menu button is completely non-responsive.

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

Keywords

menubutton, widget

2.9 text

text \- Create and manipulate text widgets

Synopsis

text *pathName* ?*options*?

Standard Options

background	foreground	insertWidth	selectBorderWidth
borderWidth	insertBackground	padX	selectForeground
cursor	insertBorderWidth	padY	setGrid
exportSelection	insertOffTime	relief	yScrollCommand
font	insertOnTime	selectBackground	

See ⟨undefined⟩ [options], page ⟨undefined⟩, for more information.

Arguments for Text

:height

Name="**height**" Class="**Height**"

Specifies the desired height for the window, in units of characters. Must be at least one.

:state

Name="**state**" Class="**State**"

Specifies one of two states for the text: **normal** or **disabled**. If the text is disabled then characters may not be inserted or deleted and no insertion cursor will be displayed, even if the input focus is in the widget.

:width

Name="**width**" Class="**Width**"

Specifies the desired width for the window in units of characters. If the font doesn't have a uniform width then the width of the character "0" is used in translating from character units to screen units.

:wrap

Name="**wrap**" Class="**Wrap**"

Specifies how to handle lines in the text that are too long to be displayed in a single line of the text's window. The value must be **none** or **char** or **word**. A wrap mode of **none** means that each line of text appears as exactly one line on the screen; extra characters that don't fit on the screen are not displayed. In the other modes each line of text will be broken up into several screen lines if necessary to keep all the characters visible. In **char** mode a screen line break may occur after any character; in **word** mode a line break will only be made at word boundaries.

Description

The **text** command creates a new window (given by the *pathName* argument) and makes it into a text widget. Additional options, described above, may be specified on the command line or in the option database to configure aspects of the text such as its default background color and relief. The **text** command returns the path name of the new window.

A text widget displays one or more lines of text and allows that text to be edited. Text widgets support three different kinds of annotations on the text, called tags, marks, and windows. Tags allow different portions of the text to be displayed with different fonts and colors. In addition, Tcl commands can be associated with tags so that commands are invoked when particular actions such as keystrokes and mouse button presses occur in particular ranges of the text. See TAGS below for more details.

The second form of annotation consists of marks, which are floating markers in the text. Marks are used to keep track of various interesting positions in the text as it is edited. See MARKS below for more details.

The third form of annotation allows arbitrary windows to be displayed in the text widget. See WINDOWS 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 pixel whose x and y coordinates within the text's window are *x* and *y*.

end Indicates the last character in the text, which is always a newline character.

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 "\fBend \- 1 chars" refers to the next-to-last character in the text and "\fBinsert 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. There may be any number of tags associated with characters in a text. Each tag may refer to a single character, a range of characters, or several ranges of characters. An individual character may have any number of tags associated with it.

A priority order is defined among tags, and this order is used in implementing some of the tag-related functions described below. When a tag is defined (by associating it with characters or setting its display options or binding commands to it), it is given a priority higher than any existing tag. The priority order of tags may be redefined using the "*pathName* **:tag :raise**" and "*pathName* **:tag :lower**" widget commands.

Tags serve three purposes in text widgets. First, they control the way information is displayed on the screen. By default, characters are displayed as determined by the **background**, **font**, and **foreground** options for the text widget. However, display options may

be associated with individual tags using the "*pathName* **:tag configure**" widget command. If a character has been tagged, then the display options associated with the tag override the default display style. The following options are currently supported for tags:

:background *color*

Color specifies the background color to use for characters associated with the tag. It may have any of the forms accepted by **Tk_GetColor**.

:bgstipple *bitmap*

Bitmap specifies a bitmap that is used as a stipple pattern for the background. It may have any of the forms accepted by **Tk_GetBitmap**. If *bitmap* hasn't been specified, or if it is specified as an empty string, then a solid fill will be used for the background.

:borderwidth *pixels*

Pixels specifies the width of a 3-D border to draw around the background. It may have any of the forms accepted by **Tk_GetPixels**. This option is used in conjunction with the **:relief** option to give a 3-D appearance to the background for characters; it is ignored unless the **:background** option has been set for the tag.

:fgstipple *bitmap*

Bitmap specifies a bitmap that is used as a stipple pattern when drawing text and other foreground information such as underlines. It may have any of the forms accepted by **Tk_GetBitmap**. If *bitmap* hasn't been specified, or if it is specified as an empty string, then a solid fill will be used.

:font *fontName*

FontName is the name of a font to use for drawing characters. It may have any of the forms accepted by **Tk_GetFontStruct**.

:foreground *color*

Color specifies the color to use when drawing text and other foreground information such as underlines. It may have any of the forms accepted by **Tk_GetColor**.

:relief *relief*

\fIRelief specifies the 3-D relief to use for drawing backgrounds, in any of the forms accepted by **Tk_GetRelief**. This option is used in conjunction with the **:borderwidth** option to give a 3-D appearance to the background for characters; it is ignored unless the **:background** option has been set for the tag.

:underline *boolean*

Boolean specifies whether or not to draw an underline underneath characters. It may have any of the forms accepted by **Tk_GetBoolean**.

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 optionl, then the default style for the widget will be used.

The second purpose for tags is event bindings. You can associate bindings with a tag in much the same way you can associate bindings with a widget class:

whenever particular X events occur on characters with the given tag, a given Tcl command will be executed. Tag bindings can be used to give behaviors to ranges of characters; among other things, this allows hypertext-like features to be implemented. For details, see the description of the **tag bind** widget command below.

The third use for tags is in managing the selection. See THE SELECTION below.

Marks

The second form of annotation in text widgets is a mark. Marks are used for remembering particular places in a text. They are something like tags, in that they have names and they refer to places in the file, but a mark isn't associated with particular characters. Instead, a mark is associated with the gap between two characters. Only a single position may be associated with a mark at any given time. If the characters around a mark are deleted the mark will still remain; it will just have new neighbor characters. In contrast, if the characters containing a tag are deleted then the tag will no longer have an association with characters in the file. Marks may be manipulated with the "*pathName* **:mark**" widget command, and their current locations may be determined by using the mark name as an index in widget commands.

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 unset.

Windows

The third form of annotation in text widgets is a window. Window support isn't implemented yet, but when it is it will be described here.

The Selection

Text widgets support the standard X selection. Selection support is implemented via tags. If the **exportSelection** option for the text widget is true then the **sel** tag will be associated with the selection:

[1] Whenever characters are tagged with **sel** the text widget will claim ownership of the selection.

[2] Attempts to retrieve the selection will be serviced by the text widget, returning all the charaters with the **sel** tag.

[3] If the selection is claimed away by another application or by another window within this application, then the **sel** tag will be removed from all characters in the text.

The **sel** tag is automatically defined when a text widget is created, and it may not be deleted with the "*pathName* **:tag delete**" widget command. Furthermore, the **selectBackground**, **selectBorderWidth**, and **selectForeground** options for the text widget are tied to the **:background**, **:borderwidth**, and **:foreground** options for the **sel** tag: changes in either will automatically be reflected in the other.

The Insertion Cursor

The mark named **insert** has special significance in text widgets. It is defined automatically when a text widget is created and it may not be unset with the "*pathName* **:mark unset**" widget command. The **insert** mark represents the position of the insertion cursor, and the insertion cursor will automatically be drawn at this point whenever the text widget has the input focus.

A Text Widget's Arguments

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 **:compare** *index1 op index2*

Compares the indices given by *index1* and *index2* according to the relational operator given by *op*, and returns 1 if the relationship is satisfied and 0 if it isn't. *Op* must be one of the operators <, <=, ==, >=, >, or !=. If *op* is == then 1 is returned if the two indices refer to the same character, if *op* is < then 1 is returned if *index1* refers to an earlier character in the text than *index2*, and so on.

pathName **:configure** ?*option*? *?value option value ...?*

Query or modify the configuration options of the widget. If no *option* is specified, returns a list describing all of the available options for *pathName* (see **Tk_ConfigureInfo** for information on the format of this list). If *option* is specified with no *value*, then the command returns a list describing the one named option (this list will be identical to the corresponding sublist of the value returned if no *option* is specified). If one or more *option:value* pairs are specified, then the command modifies the given widget option(s) to have the given value(s); in this case the command returns an empty string. *Option* may have any of the values accepted by the **text** command.

pathName **:debug** ?*boolean*?

If *boolean* is specified, then it must have one of the true or false values accepted by Tcl_GetBoolean. If the value is a true one then internal consistency checks will be turned on in the B-tree code associated with text widgets. If *boolean* has a false value then the debugging checks will be turned off. In either case the command returns an empty string. If *boolean* is not specified then the command

returns **on** or **off** to indicate whether or not debugging is turned on. There is a single debugging switch shared by all text widgets: turning debugging on or off in any widget turns it on or off for all widgets. For widgets with large amounts of text, the consistency checks may cause a noticeable slow-down.

pathName **:delete** *index1* ?*index2*?
Delete a range of characters from the text. If both *index1* and *index2* are specified, then delete all the characters starting with the one given by *index1* and stopping just before *index2* (i.e. the character at *index2* is not deleted). If *index2* doesn't specify a position later in the text than *index1* then no characters are deleted. If *index2* isn't specified then the single character at *index1* is deleted. It is not allowable to delete characters in a way that would leave the text without a newline as the last character. The command returns an empty string.

pathName **: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** \fIindex chars
Inserts *chars* into the text just before the character at *index* and returns an empty string. It is not possible to insert characters after the last newline of the text.

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 :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 **:scan** *option args*
This command is used to implement scanning on texts. It has two forms, depending on *option*:

pathName **:scan :mark** *y*
Records *y* and the current view in the text window; used in conjunction with later **scan dragto** commands. Typically this command is associated with a mouse button press in the widget. It returns an empty string.

pathName **:scan :dragto** *y*
This command computes the difference between its *y* argument and the *y* argument to the last **scan mark** command for the widget. It then adjusts the view up or down by 10 times the difference in y-coordinates. This command is typically associated with mouse motion events in the widget, to produce the effect of dragging the text at high speed through the window. The return value is an empty string.

pathName **: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*?
Associate the tag *tagName* with all of the characters starting with *index1* and ending just before *index2* (the character at *index2* isn't tagged). If *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 **:tag :bind** *tagName* ?*sequence*? ?*command*?
This command associates *command* with the tag given by *tagName*. Whenever the event sequence given by *sequence* occurs for a character that has been tagged with *tagName*, the command will be invoked. This widget command is similar to the **bind** command except that it operates on characters in a text rather than entire widgets. See the **bind** manual entry for complete details on the syntax of *sequence* and the substitutions performed on *command* before invoking it. If all arguments are specified then a new binding is created, replacing any existing binding for the same *sequence* and *tagName* (if the first character of *command* is "+" then *command* augments an existing binding rather than replacing it). In this case the return value is an empty string. If *command* is omitted then the command returns the *command* associated with *tagName* and

sequence (an error occurs if there is no such binding). If both *command* and *sequence* are omitted then the command returns a list of all the sequences for which bindings have been defined for *tagName*.

The only events for which bindings may be specified are those related to the mouse and keyboard, such as **Enter**, **Leave**, **ButtonPress**, **Motion**, and **KeyPress**. Event bindings for a text widget use the **current** mark described under MARKS above. **Enter** events trigger for a character when it becomes the current character (i.e. the **current** mark moves to just in front of that character). **Leave** events trigger for a character when it ceases to be the current item (i.e. the **current** mark moves away from that character, or the character is deleted). These events are different than **Enter** and **Leave** events for windows. Mouse and keyboard events are directed to the current character.

It is possible for the current character to have multiple tags, and for each of them to have a binding for a particular event sequence. When this occurs, the binding from the highest priority tag is used. If a particular tag doesn't have a binding that matches an event, then the tag is ignored and tags with lower priority will be checked.

If bindings are created for the widget as a whole using the **bind** command, then those bindings will supplement the tag bindings. This means that a single event can trigger two Tcl scripts, one for a widget-level binding and one for a tag-level binding.

pathName **:tag :configure** *tagName* ?*option*? ?*value*? ?*option value ...*?
This command is similar to the **configure** widget command except that it modifies options associated with the tag given by *tagName* instead of modifying options for the overall text widget. If no *option* is specified, the command returns a list describing all of the available options for *tagName* (see **Tk_ConfigureInfo** for information on the format of this list). If *option* is specified with no *value*, then the command returns a list describing the one named option (this list will be identical to the corresponding sublist of the value returned if no *option* is specified). If one or more *option:value* pairs are specified, then the command modifies the given option(s) to have the given value(s) in *tagName*; in this case the command returns an empty string. See TAGS above for details on the options available for tags.

pathName **:tag :delete** *tagName* ?*tagName ...*?
Deletes all tag information for each of the *tagName* arguments. The command removes the tags from all characters in the file and also deletes any other information associated with the tags, such as bindings and display information. The command returns an empty string.

pathName **:tag :lower** *tagName* ?*belowThis*?
Changes the priority of tag *tagName* so that it is just lower in priority than the tag whose name is *belowThis*. If *belowThis* is omitted, then *tagName*'s priority is changed to make it lowest priority of all tags.

pathName **:tag :names** ?*index*?
Returns a list whose elements are the names of all the tags that are active at the character position given by *index*. If *index* is omitted, then the return value will describe all of the tags that exist for the text (this includes all tags that have been named in a "*pathName* **:tag**" widget command but haven't been deleted by a "*pathName* **:tag :delete**" widget command, even if no characters are currently marked with the tag). The list will be sorted in order from lowest priority to highest priority.

pathName **:tag :nextrange** *tagName index1* ?*index2*?
This command searches the text for a range of characters tagged with *tagName* where the first character of the range is no earlier than the character at *index1* and no later than the character just before *index2* (a range starting at *index2* will not be considered). If several matching ranges exist, the first one is chosen. The command's return value is a list containing two elements, which are the index of the first character of the range and the index of the character just after the last one in the range. If no matching range is found then the return value is an empty string. If *index2* is not given then it defaults to the end of the text.

pathName **:tag :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*?
Remove the tag *tagName* from all of the characters starting at *index1* and ending just before *index2* (the character at *index2* isn't affected). If *index2* is omitted then the single character at *index1* is untagged. 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 **:yview** ?**:pickplace**? *what*

This command changes the view in the widget's window so that the line given by *what* is visible in the window. *What* may be either an absolute line number, where 0 corresponds to the first line of the file, or an index with any of the forms described under INDICES above. The first form (absolute line number) is used in the commands issued by scrollbars to control the widget's view. If the **:pickplace** option isn't specified then *what* will appear at the top of the window. If **:pickplace** is specified then the widget chooses where *what* appears in the window:

[1] If *what* is already visible somewhere in the window then the command does nothing.

[2] If *what* is only a few lines off-screen above the window then it will be positioned at the top of the window.

[3] If *what* is only a few lines off-screen below the window then it will be positioned at the bottom of the window.

[4] Otherwise, *what* will be centered in the window.

The **:pickplace** option is typically used after inserting text to make sure that the insertion cursor is still visible on the screen. This command returns an empty string.

Bindings

Tk automatically creates class bindings for texts that give them the following default behavior:

[1] Pressing mouse button 1 in an text positions the insertion cursor just before the character underneath the mouse cursor and sets the input focus to this widget.

[2] Dragging with mouse button 1 strokes out a selection between the insertion cursor and the character under the mouse.

[3] If you double-press mouse button 1 then the word under the mouse cursor will be selected, the insertion cursor will be positioned at the beginning of the word, and dragging the mouse will stroke out a selection whole words at a time.

[4] If you triple-press mouse button 1 then the line under the mouse cursor will be selected, the insertion cursor will be positioned at the beginning of the line, and dragging the mouse will stroke out a selection whole line at a time.

[5] The ends of the selection can be adjusted by dragging with mouse button 1 while the shift key is down; this will adjust the end of the selection that was nearest to the mouse cursor when button 1 was pressed. If the selection was made in word or line mode then it will be adjusted in this same mode.

[6] The view in the text can be adjusted by dragging with mouse button 2.

[7] If the input focus is in a text widget and characters are typed on the keyboard, the characters are inserted just before the insertion cursor.

[8] Control+h and the Backspace and Delete keys erase the character just before the insertion cursor.

[9] Control+v inserts the current selection just before the insertion cursor.

[10] Control+d deletes the selected characters; an error occurs if the selection is not in this widget.

If the text is disabled using the **state** option, then the text's view can still be adjusted and text in the 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

2.10 entry

entry \- Create and manipulate entry widgets

Synopsis

entry *pathName ?options?*

Standard Options

`background`	`foreground`	`insertWidth`	`selectForeground`
`borderWidth`	`insertBackground`	`relief`	`textVariable`
`cursor`	`insertBorderWidth`	`scrollCommand`	
`exportSelection`	`insertOffTime`	`selectBackground`	
`font`	`insertOnTime`	`selectBorderWidth`	

See ⟨undefined⟩ [options], page ⟨undefined⟩, for more information.

Arguments for Entry

:state

Name="**state**" Class="**State**"

Specifies one of two states for the entry: **normal** or **disabled**. If the entry is disabled then the value may not be changed using widget commands and no insertion cursor will be displayed, even if the input focus is in the widget.

:width

Name="**width**" Class="**Width**"

Specifies an integer value indicating the desired width of the entry window, in average-size characters of the widget's font.

Description

The **entry** command creates a new window (given by the *pathName* argument) and makes it into an entry widget. Additional options, described above, may be specified on the command line or in the option database to configure aspects of the entry such as its colors, font, and relief. The **entry** command returns its *pathName* argument. At the time this command is invoked, there must not exist a window named *pathName*, but *pathName*'s parent must exist.

An entry is a widget that displays a one-line text string and allows that string to be edited using widget commands described below, which are typically bound to keystrokes and mouse actions. When first created, an entry's string is empty. A portion of the entry may be selected as described below. If an entry is exporting its selection (see the **exportSelection** option), then it will observe the standard X11 protocols for handling the selection; entry selections are available as type **STRING**. Entries also observe the standard Tk rules for dealing with the input focus. When an entry has the input focus it displays an *insertion cursor* to indicate where new characters will be inserted.

Entries are capable of displaying strings that are too long to fit entirely within the widget's window. In this case, only a portion of the string will be displayed; commands described below may be used to change the view in the window. Entries use the standard **scrollCommand** mechanism for interacting with scrollbars (see the description of the **scrollCommand** option for details). They also support scanning, as described below.

A Entry Widget's Arguments

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.

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 last character 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 **:configure** ?*option*? ?*value option value ...*?
Query or modify the configuration options of the widget. If no *option* is specified, returns a list describing all of the available options for *pathName* (see **Tk_ConfigureInfo** for information on the format of this list). If *option* is specified with no *value*, then the command returns a list describing the one named option (this list will be identical to the corresponding sublist of the value returned if no *option* is specified). If one or more *option:value* pairs are specified, then the command modifies the given widget option(s) to have the given value(s); in this case the command returns an empty string. *Option* may have any of the values accepted by the **entry** command.

pathName **:delete** *first* ?*last*?
Delete one or more elements of the entry. *First* and *last* are indices of of the first and last characters in the range to be deleted. If *last* isn't specified it defaults to *first*, i.e. a single character is deleted. This command returns an empty string.

pathName **:get**
Returns the entry's string.

pathName **:icursor** *index*
Arrange for the insertion cursor to be displayed just before the character given by *index*. Returns an empty string.

pathName **:index** *index*
Returns the numerical index corresponding to *index*.

pathName **:insert** *index string*
Insert the characters of *string* just before the character indicated by *index*. Returns an empty string.

pathName **:scan** *option args*

This command is used to implement scanning on entries. It has two forms, depending on *option*:

pathName **:scan :mark** *x*

Records *x* and the current view in the entry window; used in conjunction with later **scan dragto** commands. Typically this command is associated with a mouse button press in the widget. It returns an empty string.

pathName **:scan :dragto** *x*

This command computes the difference between its *x* argument and the *x* argument to the last **scan mark** command for the widget. It then adjusts the view left or right by 10 times the difference in x-coordinates. This command is typically associated with mouse motion events in the widget, to produce the effect of dragging the entry at high speed through the window. The return value is an empty string.

pathName **:select** *option arg*

This command is used to adjust the selection within an entry. It has several forms, depending on *option*:

pathName **:select :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 **:select :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 **:select :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 **:select :to** *index*

Set the selection to consist of the elements from the anchor point to element *index*, inclusive. 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 **:view** *index*

Adjust the view in the entry so that element *index* is at the left edge of the window. Returns an empty string.

"Default Bindings"

Tk automatically creates class bindings for entries that give them the following default behavior:

[1] Clicking mouse button 1 in an entry positions the insertion cursor just before the character underneath the mouse cursor and sets the input focus to this widget.

[2] Dragging with mouse button 1 strokes out a selection between the insertion cursor and the character under the mouse.

[3] The ends of the selection can be adjusted by dragging with mouse button 1 while the shift key is down; this will adjust the end of the selection that was nearest to the mouse cursor when button 1 was pressed.

[4] The view in the entry can be adjusted by dragging with mouse button 2.

[5] If the input focus is in an entry widget and characters are typed on the keyboard, the characters are inserted just before the insertion cursor.

[6] Control-h and the Backspace and Delete keys erase the character just before the insertion cursor.

[7] Control-w erases the word just before the insertion cursor.

[8] Control-u clears the entry to an empty string.

[9] Control-v inserts the current selection just before the insertion cursor.

[10] Control-d deletes the selected characters; an error occurs if the selection is not in this widget.

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

2.11 message

message \- Create and manipulate message widgets

Synopsis

message *pathName ?options?*

Standard Options

`anchor`	`cursor`	`padX`	`text`
`background`	`font`	`padY`	`textVariable`
`borderWidth`	`foreground`	`relief`	`width`

See ⟨undefined⟩ [options], page ⟨undefined⟩, for more information.

Arguments for Message

:aspect

Name="**aspect**" Class="**Aspect**"

Specifies a non-negative integer value indicating desired aspect ratio for the text. The aspect ratio is specified as 100*width/height. 100 means the text should be as wide as it is tall, 200 means the text should be twice as wide as it is tall, 50 means the text should be twice as tall as it is wide, and so on. Used to choose line length for text if **width** option isn't specified. Defaults to 150.

:justify

Name="**justify**" Class="**Justify**"

Specifies how to justify lines of text. Must be one of **left**, **center**, or **right**. Defaults to **left**. This option works together with the **anchor**, **aspect**, **padX**, **padY**, and **width** options to provide a variety of arrangements of the text within the window. The **aspect** and **width** options determine the amount of screen space needed to display the text. The **anchor**, **padX**, and **padY** options determine where this rectangular area is displayed within the widget's window, and the **justify** option determines how each line is displayed within that rectangular region. For example, suppose **anchor** is **e** and **justify** is **left**, and that the message window is much larger than needed for the text. The the text will displayed so that the left edges of all the lines line up and the right edge of the longest line is **padX** from the right side of the window; the entire text block will be centered in the vertical span of the window.

:width

Name="**width**" Class="**Width**"

Specifies the length of lines in the window. The value may have any of the forms acceptable to **Tk_GetPixels**. If this option has a value greater than zero then the **aspect** option is ignored and the **width** option determines the line length. If this option has a value less than or equal to zero, then the **aspect** option determines the line length.

Description

The **message** command creates a new window (given by the *pathName* argument) and makes it into a message widget. Additional options, described above, may be specified on the command line or in the option database to configure aspects of the message such as its colors, font, text, and initial relief. The **message** command returns its *pathName* argument. At the time this command is invoked, there must not exist a window named *pathName*, but *pathName*'s parent must exist.

A message is a widget that displays a textual string. A message widget has three special features. First, it breaks up its string into lines in order to produce a given aspect ratio for the window. The line breaks are chosen at word boundaries wherever possible (if not

even a single word would fit on a line, then the word will be split across lines). Newline characters in the string will force line breaks; they can be used, for example, to leave blank lines in the display.

The second feature of a message widget is justification. The text may be displayed left-justified (each line starts at the left side of the window), centered on a line-by-line basis, or right-justified (each line ends at the right side of the window).

The third feature of a message widget is that it handles control characters and non-printing characters specially. Tab characters are replaced with enough blank space to line up on the next 8-character boundary. Newlines cause line breaks. Other control characters (ASCII code less than 0x20) and characters not defined in the font are displayed as a four-character sequence \fB\ex*hh* where *hh* is the two-digit hexadecimal number corresponding to the character. In the unusual case where the font doesn't contain all of the characters in "0123456789abcdef\ex" then control characters and undefined characters are not displayed at all.

A Message Widget's Arguments

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 **:configure** ?*option*? ?*value option value ...*?

Query or modify the configuration options of the widget. If no *option* is specified, returns a list describing all of the available options for *pathName* (see **Tk_ConfigureInfo** for information on the format of this list). If *option* is specified with no *value*, then the command returns a list describing the one named option (this list will be identical to the corresponding sublist of the value returned if no *option* is specified). If one or more *option:value* pairs are specified, then the command modifies the given widget option(s) to have the given value(s); in this case the command returns an empty string. *Option* may have any of the values accepted by the **message** command.

"Default Bindings"

When a new message is created, it has no default event bindings: messages are intended for output purposes only.

Bugs

Tabs don't work very well with text that is centered or right-justified. The most common result is that the line is justified wrong.

Keywords

message, widget

2.12 frame

frame \- Create and manipulate frame widgets

Synopsis

frame *pathName* ?**:class** *className*? ?*options*?

Standard Options

background	cursor	relief
borderWidth	geometry	

See ⟨undefined⟩ [options], page ⟨undefined⟩, for more information.

Arguments for Frame

:height

Name="**height**" Class="**Height**"

Specifies the desired height for the window in any of the forms acceptable to **Tk_GetPixels**. This option is only used if the **:geometry** option is unspecified. If this option is less than or equal to zero (and **:geometry** is not specified) then the window will not request any size at all.

:width

Name="**width**" Class="**Width**"

Specifies the desired width for the window in any of the forms acceptable to **Tk_GetPixels**. This option is only used if the **:geometry** option is unspecified. If this option is less than or equal to zero (and **:geometry** is not specified) then the window will not request any size at all.

Description

The **frame** command creates a new window (given by the *pathName* argument) and makes it into a frame widget. Additional options, described above, may be specified on the command line or in the option database to configure aspects of the frame such as its background color and relief. The **frame** command returns the path name of the new window.

A frame is a simple widget. Its primary purpose is to act as a spacer or container for complex window layouts. The only features of a frame are its background color and an optional 3-D border to make the frame appear raised or sunken.

In addition to the standard options listed above, a **:class** option may be specified on the command line. If it is specified, then the new widget's class will be set to *className* instead of **Frame**. Changing the class of a frame widget may be useful in order to use a special class name in database options referring to this widget and its children. Note: **:class** is handled differently than other command-line options and cannot be specified using the option database (it has to be processed before the other options are even looked up, since the new class name will affect the lookup of the other options). In addition, the **:class** option may not be queried or changed using the **config** command described below.

A Frame Widget's Arguments

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 **:configure** ?*option*? *?value option value ...?*

Query or modify the configuration options of the widget. If no *option* is specified, returns a list describing all of the available options for *pathName* (see **Tk_ConfigureInfo** for information on the format of this list). If *option* is specified with no *value*, then the command returns a list describing the one named option (this list will be identical to the corresponding sublist of the value returned if no *option* is specified). If one or more *option:value* pairs are specified, then the command modifies the given widget option(s) to have the given value(s); in this case the command returns an empty string. *Option* may have any of the values accepted by the **frame** command.

Bindings

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

Keywords

frame, widget

2.13 label

label \- Create and manipulate label widgets

Synopsis

label *pathName ?options?*

Standard Options

anchor	borderWidth	foreground	relief
background	cursor	padX	text
bitmap	font	padY	textVariable

See ⟨undefined⟩ [options], page ⟨undefined⟩, for more information.

Arguments for Label

:height

Name="**height**" Class="**Height**"

Specifies a desired height for the label. If a bitmap is being displayed in the label then the value is in screen units (i.e. any of the forms acceptable to **Tk_GetPixels**); for text it is in lines of text. If this option isn't specified, the label's desired height is computed from the size of the bitmap or text being displayed in it.

:width

Name="**width**" Class="**Width**"

Specifies a desired width for the label. If a bitmap is being displayed in the label then the value is in screen units (i.e. any of the forms acceptable to **Tk_GetPixels**); for text it is in characters. If this option isn't specified, the label's desired width is computed from the size of the bitmap or text being displayed in it.

Description

The **label** command creates a new window (given by the *pathName* argument) and makes it into a label widget. Additional options, described above, may be specified on the command line or in the option database to configure aspects of the label such as its colors, font, text, and initial relief. The **label** command returns its *pathName* argument. At the time this command is invoked, there must not exist a window named *pathName*, but *pathName*'s parent must exist.

A label is a widget that displays a textual string or bitmap. The label can be manipulated in a few simple ways, such as changing its relief or text, using the commands described below.

A Label Widget's Arguments

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 **:configure** *?option? ?value option value ...?*

Query or modify the configuration options of the widget. If no *option* is specified, returns a list describing all of the available options for *pathName* (see **Tk_ConfigureInfo** for information on the format of this list). If *option* is specified with no *value*, then the command returns a list describing the one named option (this list will be identical to the corresponding sublist of the value returned if no *option* is specified). If one or more *option:value* pairs are specified, then the command modifies the given widget option(s) to have the given

value(s); in this case the command returns an empty string. *Option* may have any of the values accepted by the **label** command.

Bindings

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

Keywords

label, widget

2.14 radiobutton

radiobutton \- Create and manipulate radio-button widgets

Synopsis

radiobutton *pathName* ?*options*?

Standard Options

activeBackground	bitmap	font	relief
activeForeground	borderWidth	foreground	text
anchor	cursor	padX	textVariable
background	disabledForeground	padX	

See ⟨undefined⟩ [options], page ⟨undefined⟩, for more information.

Arguments for Radiobutton

:command

Name="**command**" Class="**Command**"

Specifies a Tcl command to associate with the button. This command is typically invoked when mouse button 1 is released over the button window. The button's global variable (**:variable** option) will be updated before the command is invoked.

:height

Name="**height**" Class="**Height**"

Specifies a desired height for the button. If a bitmap is being displayed in the button then the value is in screen units (i.e. any of the forms acceptable to **Tk_GetPixels**); for text it is in lines of text. If this option isn't specified, the button's desired height is computed from the size of the bitmap or text being displayed in it.

:selector

Name="**selector**" Class="**Foreground**"

Specifies the color to draw in the selector when this button is selected. If specified as an empty string then no selector is drawn for the button.

:state

Name="**state**" Class="**State**"

Specifies one of three states for the radio button: **normal**, **active**, or **disabled**. In normal state the radio button is displayed using the **foreground** and **background** options. The active state is typically used when the pointer is over the radio button. In active state the radio button is displayed using the **activeForeground** and **activeBackground** options. Disabled state means that the radio button is insensitive: it doesn't activate and doesn't respond to mouse button presses. In this state the **disabledForeground** and **background** options determine how the radio button is displayed.

:value

Name="**value**" Class="**Value**"

Specifies value to store in the button's associated variable whenever this button is selected. Defaults to the name of the radio button.

:variable

Name="**variable**" Class="**Variable**"

Specifies name of global variable to set whenever this button is selected. Changes in this variable also cause the button to select or deselect itself. Defaults to the value **selectedButton**.

:width

Name="**width**" Class="**Width**"

Specifies a desired width for the button. If a bitmap is being displayed in the button then the value is in screen units (i.e. any of the forms acceptable to **Tk_GetPixels**); for text it is in characters. If this option isn't specified, the button's desired width is computed from the size of the bitmap or text being displayed in it.

Description

The **radiobutton** command creates a new window (given by the *pathName* argument) and makes it into a radiobutton widget. Additional options, described above, may be specified on the command line or in the option database to configure aspects of the radio button such as its colors, font, text, and initial relief. The **radiobutton** command returns its *pathName* argument. At the time this command is invoked, there must not exist a window named *pathName*, but *pathName*'s parent must exist.

A radio button is a widget that displays a textual string or bitmap and a diamond called a *selector*. A radio button has all of the behavior of a simple button: it can display itself in either of three different ways, according to the **state** option; it can be made to appear raised, sunken, or flat; it can be made to flash; and it invokes a Tcl command whenever mouse button 1 is clicked over the check button.

In addition, radio buttons can be *selected*. If a radio button is selected then a special highlight appears in the selector and a Tcl variable associated with the radio button is set to a particular value. If the radio button is not selected then the selector is drawn in a different fashion. Typically, several radio buttons share a single variable and the value of the variable indicates which radio button is to be selected. When a radio button is selected it sets the value of the variable to indicate that fact; each radio button 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 radio button, plus the variable to be stored into it, may be modified with options on the command line or in the option database. By default a radio button is configured to select itself on button clicks.

A Radiobutton Widget's Arguments

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 radio-button widgets:

pathName **:activate**
: Change the radio button's state to **active** and redisplay the button using its active foreground and background colors instead of normal colors. This command is ignored if the radio button's state is **disabled**. This command is obsolete and will eventually be removed; use "*pathName* **:configure :state active**" instead.

pathName **:configure** ?*option*? ?*value option value ...*?
: Query or modify the configuration options of the widget. If no *option* is specified, returns a list describing all of the available options for *pathName* (see **Tk_ConfigureInfo** for information on the format of this list). If *option* is specified with no *value*, then the command returns a list describing the one named option (this list will be identical to the corresponding sublist of the value returned if no *option* is specified). If one or more *option:value* pairs are specified, then the command modifies the given widget option(s) to have the given value(s); in this case the command returns an empty string. *Option* may have any of the values accepted by the **radiobutton** command.

pathName **:deactivate**
: Change the radio button's state to **normal** and redisplay the button using its normal foreground and background colors. This command is ignored if the radio button's state is **disabled**. This command is obsolete and will eventually be removed; use "*pathName* **:configure :state normal**" instead.

pathName **:deselect**
Deselect the radio button: redisplay it without a highlight in the selector and set the associated variable to an empty string. If this radio button was not currently selected, then the command has no effect.

pathName **:flash**
Flash the radio button. This is accomplished by redisplaying the radio button several times, alternating between active and normal colors. At the end of the flash the radio button is left in the same normal/active state as when the command was invoked. This command is ignored if the radio button's state is **disabled**.

pathName **:invoke**
Does just what would have happened if the user invoked the radio button with the mouse: select the button and invoke 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 radio button. This command is ignored if the radio button's state is **disabled**.

pathName **:select**
Select the radio button: display it with a highlighted selector and set the associated variable to the value corresponding to this widget.

Bindings

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

[1] The radio button activates whenever the mouse passes over it and deactivates whenever the mouse leaves the radio button.

[2] The radio button's relief is changed to sunken whenever mouse button 1 is pressed over it, and the relief is restored to its original value when button 1 is later released.

[3] If mouse button 1 is pressed over the radio button and later released over the radio button, the radio button is invoked (i.e. it is selected and the command associated with the button is invoked, if there is one). However, if the mouse is not over the radio button when button 1 is released, then no invocation occurs.

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

Keywords

radio button, widget

2.15 toplevel

toplevel \- Create and manipulate toplevel widgets

Synopsis

toplevel *pathName* ?**:screen** *screenName*? ?**:class** *className*? ?*options*?

Standard Options

`background`	`geometry`
`borderWidth`	`relief`

See ⟨undefined⟩ [options], page ⟨undefined⟩, for more information.

Arguments for Toplevel

Description

The **toplevel** command creates a new toplevel widget (given by the *pathName* argument). Additional options, described above, may be specified on the command line or in the option database to configure aspects of the toplevel such as its background color and relief. The **toplevel** command returns the path name of the new window.

A toplevel is similar to a frame except that it is created as a top-level window: its X parent is the root window of a screen rather than the logical parent from its path name. The primary purpose of a toplevel is to serve as a container for dialog boxes and other collections of widgets. The only features of a toplevel are its background color and an optional 3-D border to make the toplevel appear raised or sunken.

Two special command-line options may be provided to the **toplevel** command: **:class** and **:screen**. If **:class** is specified, then the new widget's class will be set to *className* instead of **Toplevel**. Changing the class of a toplevel widget may be useful in order to use a special class name in database options referring to this widget and its children. The **:screen** option may be used to place the window on a different screen than the window's logical parent. Any valid screen name may be used, even one associated with a different display.

Note: **:class** and **:screen** are handled differently than other command-line options. They may not be specified using the option database (these options must have been processed before the new window has been created enough to use the option database; in particular, the new class name will affect the lookup of options in the database). In addition, **:class** and **:screen** may not be queried or changed using the **config** command described below. However, the **winfo :class** command may be used to query the class of a window, and **winfo :screen** may be used to query its screen.

A Toplevel Widget's Arguments

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 **:configure** ?*option*? ?*value option value ...*?

Query or modify the configuration options of the widget. If no *option* is specified, returns a list describing all of the available options for *pathName* (see

Tk_ConfigureInfo for information on the format of this list). If *option* is specified with no *value*, then the command returns a list describing the one named option (this list will be identical to the corresponding sublist of the value returned if no *option* is specified). If one or more *option:value* pairs are specified, then the command modifies the given widget option(s) to have the given value(s); in this case the command returns an empty string. *Option* may have any of the values accepted by the **toplevel** command.

Bindings

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

Keywords

toplevel, widget

3 Control

3.1 after

after - Execute a command after a time delay

Synopsis

after *ms ?arg1 arg2 arg3 ...?*

Description

This command is used to delay execution of the program or to execute a command in background after a delay. The *ms* argument gives a time in milliseconds. If *ms* is the only argument to **after** then the command sleeps for *ms* milliseconds and returns. While the command is sleeping the application does not respond to X events and other events.

If additional arguments are present after *ms*, then a Tcl command is formed by concatenating all the additional arguments in the same fashion as the **concat** command. **After** returns immediately but arranges for the command to be executed *ms* milliseconds later in background. 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 always returns an empty string.

See ⟨undefined⟩ [tkerror], page ⟨undefined⟩.

Keywords

delay, sleep, time

3.2 bind

bind \- Arrange for X events to invoke Tcl commands

Synopsis

bind *windowSpec*

bind *windowSpec sequence*

bind *windowSpec sequence command*
bind *windowSpec sequence* **+***command*

Description

If all three arguments are specified, **bind** will arrange for *command* (a Tcl command) to be executed whenever the sequence of events given by *sequence* occurs in the window(s) identified by *windowSpec*. If *command* is prefixed with a "+", then it is appended to any existing binding for *sequence*; otherwise *command* replaces the existing binding, if any. If *command* is an empty string then the current binding for *sequence* is destroyed, leaving *sequence* unbound. In all of the cases where a *command* argument is provided, **bind** returns an empty string.

If *sequence* is specified without a *command*, then the command currently bound to *sequence* is returned, or an empty string if there is no binding for *sequence*. If neither *sequence* nor *command* is specified, then the return value is a list whose elements are all the sequences for which there exist bindings for *windowSpec*.

The *windowSpec* argument selects which window(s) the binding applies to. It may have one of three forms. If *windowSpec* is the path name for a window, then the binding applies to that particular window. If *windowSpec* is the name of a class of widgets, then the binding applies to all widgets in that class. Lastly, *windowSpec* may have the value **all**, in which case the binding applies to all windows in the application.

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:

<*modifier-modifier-type-detail*>

The entire event pattern is surrounded by angle brackets. Inside the angle brackets are zero or more modifiers, an event type, and an extra piece of information (*detail*) identifying a particular button or keysym. Any of the fields may be omitted, as long as at least one of *type* and *detail* is present. The fields must be separated by white space or dashes.

Modifiers may consist of any of the values in the following list:

```
Control                 Any
Shift                   Double
Lock                    Triple
Button1, B1             Mod1, M1, Meta, M
Button2, B2             Mod2, M2, Alt
Button3, B3             Mod3, M3
Button4, B4             Mod4, M4
Button5, B5             Mod5, M5
```

Where more than one value is listed, separated by commas, the values are equivalent. All of the modifiers except **Any**, **Double**, and **Triple** have the obvious X meanings. For example, **Button1** requires that button 1 be depressed when the event occurs. Under normal conditions the button and modifier state at the time of the event must match exactly those specified in the **bind** command. If no modifiers are specified, then events will match only if no modifiers are present. If the **Any** modifier is specified, then additional modifiers may be present besides those specified explicitly. For example, if button 1 is pressed while the shift

and control keys are down, the specifier **<Any-Control-Button-1>** will match the event, but the specifier **<Control-Button-1>** will not.

The **Double** and **Triple** modifiers are a convenience for specifying double mouse clicks and other repeated events. They cause a particular event pattern to be repeated 2 or 3 times, and also place a time and space requirement on the sequence: for a sequence of events to match a **Double** or **Triple** pattern, all of the events must occur close together in time and without substantial mouse motion in between. For example, **<Double-Button-1>** is equivalent to **<Button-1><Button-1>** with the extra time and space requirement.

The *type* field may be any of the standard X event types, with a few extra abbreviations. Below is a list of all the valid types; where two name appear together, they are synonyms.

```
ButtonPress, Button     Expose              Leave
ButtonRelease           FocusIn             Map
Circulate               FocusOut            Property
CirculateRequest        Gravity             Reparent
Colormap                Keymap              ResizeRequest
Configure               KeyPress, Key       Unmap
ConfigureRequest        KeyRelease          Visibility
Destroy                 MapRequest
Enter                   Motion
```

The last part of a long event specification is *detail*. In the case of a **ButtonPress** or **ButtonRelease** event, it is the number of a button (1-5). If a button number is given, then only an event on that particular button will match; if no button number is given, then an event on any button will match. Note: giving a specific button number is different than specifying a button modifier; in the first case, it refers to a button being pressed or released, while in the second it refers to some other button that is already depressed when the matching event occurs. If a button number is given then *type* may be omitted: if will default to **ButtonPress**. For example, the specifier **<1>** is equivalent to **<ButtonPress-1>**.

If the event type is **KeyPress** or **KeyRelease**, then *detail* may be specified in the form of an X keysym. Keysyms are textual specifications for particular keys on the keyboard; they include all the alphanumeric ASCII characters (e.g. "a" is the keysym for the ASCII character "a"), plus descriptions for non-alphanumeric characters ("comma" is the keysym for the comma character), plus descriptions for all the non-ASCII keys on the keyboard ("Shift_L" is the keysm for the left shift key, and "F1" is the keysym for the F1 function key, if it exists). The complete list of keysyms is not presented here; it should be available in other X documentation. If necessary, you can use the **%K** notation described below to print out the keysym name for an arbitrary key. If a keysym *detail* is given, then the *type* field may be omitted; it will default to **KeyPress**. For example, **<Control-comma>** is equivalent to **<Control-KeyPress-comma>**. If a keysym *detail* is specified then the **Shift** modifier need not be specified and will be ignored if specified: each keysym already implies a particular state for the shift key.

The *command* argument to **bind** is a Tcl command string, 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. If *command* contains any **%** characters, then the command string will not be executed directly. Instead, a new command string will be generated by replacing each **%**, and the character following it, with information from the current event. The replacement depends on the character following the **%**, as defined in the

list below. Unless otherwise indicated, the replacement string is the decimal value of the given field from the current event. Some of the substitutions are only valid for certain types of events; if they are used for other types of events the value substituted is undefined.

%% Replaced with a single percent.

|%#| The number of the last client request processed by the server (the *serial* field from the event). Valid for all event types.

|%a| The *above* field from the event. Valid only for **ConfigureNotify** events.

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

|%c| The *count* field from the event. Valid only for **Expose**, **GraphicsExpose**, and **MappingNotify** events.

|%d| The *detail* field from the event. The **|%d|** is replaced by a string identifying the detail. For **EnterNotify**, **LeaveNotify**, **FocusIn**, and **FocusOut** events, the string will be one of the following:

```
NotifyAncestor          NotifyNonlinearVirtual
NotifyDetailNone        NotifyPointer
NotifyInferior          NotifyPointerRoot
NotifyNonlinear         NotifyVirtual
```

For **ConfigureRequest** events, the substituted string will be one of the following:

```
Above                   Opposite
Below                   TopIf
BottomIf
```

For events other than these, the substituted string is undefined. .RE

|%f| The *focus* field from the event (**0** or **1**). Valid only for **EnterNotify** and **LeaveNotify** events.

|%h| The *height* field from the event. Valid only for **Configure**, **ConfigureNotify**, **Expose**, **GraphicsExpose**, and **ResizeRequest** events.

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

|%m| The *mode* field from the event. The substituted string is one of **NotifyNormal**, **NotifyGrab**, **NotifyUngrab**, or **NotifyWhileGrabbed**. Valid only for **EnterWindow**, **FocusIn**, **FocusOut**, and **LeaveWindow** events.

|%o| The *override_redirect* field from the event. Valid only for **CreateNotify**, **MapNotify**, **ReparentNotify**, and **ConfigureNotify** events.

|%p| The *place* field from the event, substituted as one of the strings **PlaceOnTop** or **PlaceOnBottom**. Valid only for **CirculateNotify** and **CirculateRequest** events.

|%s| The *state* field from the event. For **ButtonPress**, **ButtonRelease**, **EnterNotify**, **KeyPress**, **KeyRelease**, **LeaveNotify**, and **MotionNotify** events, a decimal string is substituted. For **VisibilityNotify**, one of the strings **VisibilityUnobscured**, **VisibilityPartiallyObscured**, and **VisibilityFullyObscured** is substituted.

|%t| The *time* field from the event. Valid only for events that contain a *time* field.

|%v| The *value_mask* field from the event. Valid only for **ConfigureRequest** events.

|%w| The *width* field from the event. Valid only for **Configure**, **ConfigureRequest**, **Expose**, **GraphicsExpose**, and **ResizeRequest** events.

|%x| The *x* field from the event. Valid only for events containing an *x* field.

|%y| The *y* field from the event. Valid only for events containing a *y* field.

%A Substitutes the ASCII character corresponding to the event, or the empty string if the event doesn't correspond to an ASCII character (e.g. the shift key was pressed). **XLookupString** does all the work of translating from the event to an ASCII character. Valid only for **KeyPress** and **KeyRelease** events.

%B The *border_width* field from the event. Valid only for **ConfigureNotify** and **CreateWindow** events.

%D The *display* field from the event. Valid for all event types.

%E The *send_event* field from the event. Valid for all event types.

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

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

%R The *root* window identifier from the event. Valid only for events containing a *root* field.

%S The *subwindow* window identifier from the event. Valid only for events containing a *subwindow* field.

%T The *type* field from the event. Valid for all event types.

%W The path name of the window to which the event was reported (the *window* field from the event). Valid for all event types.

%X The *x_root* field from the event. If a virtual-root window manager is being used then the substituted value is the corresponding x-coordinate in the virtual root. Valid only for **ButtonPress**, **ButtonRelease**, **KeyPress**, **KeyRelease**, and **MotionNotify** events.

%Y The *y_root* field from the event. If a virtual-root window manager is being used then the substituted value is the corresponding y-coordinate in the virtual root. Valid only for **ButtonPress**, **ButtonRelease**, **KeyPress**, **KeyRelease**, and **MotionNotify** events.

If the replacement string for a %-replacement contains characters that are interpreted specially by the Tcl parser (such as backslashes or square brackets or spaces) additional backslashes are added during replacement so that the result after parsing is the original replacement string. For example, if *command* is

```
insert %A
```

and the character typed is an open square bracket, then the command actually executed will be

```
insert \e[
```

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 command correctly.

At most one binding will trigger for any given X event. If several bindings match the recent events, the most specific binding is chosen and its command will be executed. The following tests are applied, in order, to determine which of several matching sequences is more specific: (a) a binding whose *windowSpec* names a particular window is more specific than a binding for a class, which is more specific than a binding whose *windowSpec* is **all**; (b) a longer sequence (in terms of number of events matched) is more specific than a shorter sequence; (c) an event pattern that specifies a specific button or key is more specific than one that doesn't; (e) an event pattern that requires a particular modifier is more specific than one that doesn't require the modifier; (e) an event pattern specifying the **Any** modifier is less specific than one that doesn't. If the matching sequences contain more than one event, then tests (c)-(e) are applied in order from the most recent event to the least recent event in the sequences. If these tests fail to determine a winner, then the most recently registered sequence is the winner.

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

When a *sequence* specified in a **bind** command contains more than one event pattern, then its command is executed whenever the recent events (leading up to and including the current event) match the given sequence. This means, for example, that if button 1 is clicked repeatedly the sequence **<Double-ButtonPress-1>** will match each button press but the first. If extraneous events that would prevent a match occur in the middle of an event sequence then the extraneous events are ignored unless they are **KeyPress** or **ButtonPress** events. For example, **<Double-ButtonPress-1>** will match a sequence of presses of button 1, even though there will be **ButtonRelease** events (and possibly **MotionNotify** events) between the **ButtonPress** events. Furthermore, a **KeyPress** event may be preceded by any number of other **KeyPress** events for modifier keys without the modifier keys preventing a match. For example, the event sequence **aB** will match a press of the **a** key, a release of the **a** key, a press of the **Shift** key, and a press of the **b** key: the press of **Shift** is ignored because it is a modifier key. Finally, if several **MotionNotify** events occur in a row, only the last one is used for purposes of matching binding sequences.

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

See ⟨undefined⟩ [tkerror], page ⟨undefined⟩.

Keywords

form, manual

3.3 destroy

destroy \- Destroy one or more windows

Synopsis

destroy ?*window window ...*?

Description

This command deletes the windows given by the *window* arguments, plus all of their descendants. If a *window* "." is deleted then the entire application will be destroyed. The *window*s are destroyed in order, and if an error occurs in destroying a window the command aborts without destroying the remaining windows.

Keywords

application, destroy, window

3.4 tk-dialog

tk-dialog \- Create modal dialog and wait for response

Synopsis

tk-dialog *window title text bitmap default string string ...*

Description

This procedure is part of the Tk script library. Its arguments describe a dialog box:

window Name of top-level window to use for dialog. Any existing window by this name is destroyed.

title Text to appear in the window manager's title bar for the dialog.

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

bitmap If non-empty, specifies a bitmap to display in the top portion of the dialog, to the left of the text. If this is an empty string then no bitmap is displayed in the dialog.

default If this is an integer greater than or equal to zero, then it gives the index of the button that is to be the default button for the dialog (0 for the leftmost button, and so on). If less than zero or an empty string then there won't be any default button.

string There will be one button for each of these arguments. Each *string* specifies text to display in a button, in order from left to right.

After creating a dialog box, **tk-dialog** waits for the user to select one of the buttons either by clicking on the button with the mouse or by typing return to invoke the default button (if any). Then it returns the index of the selected button: 0 for the leftmost button, 1 for the button next to it, and so on.

While waiting for the user to respond, **tk-dialog** sets a local grab. This prevents the user from interacting with the application in any way except to invoke the dialog box.

Keywords

bitmap, dialog, modal

3.5 exit

exit \- Exit the process

Synopsis

exit ?*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 removing application names from the global registry of applications.

Keywords

exit, process

3.6 focus

focus \- Direct keyboard events to a particular window

Synopsis

focus

focus *window*
focus *option* ?*arg arg ...*?

Description

The **focus** command is used to manage the Tk input focus. At any given time, one window in an application is designated as the focus window for that application; any key press or key release events directed to any window in the application will be redirected instead to the focus window. If there is no focus window for an application then keyboard events are discarded. Typically, windows that are prepared to deal with the focus (e.g. entries and other widgets that display editable text) will claim the focus when mouse button 1 is pressed in them. When an application is created its main window is initially given the focus.

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

focus If invoked with no arguments, **focus** returns the path name of the current focus window, or **none** if there is no focus window.

focus *window*
If invoked with a single argument consisting of a window's path name, **focus** sets the input focus to that window. The return value is an empty string.

focus :default ?*window*?
If *window* is specified, it becomes the default focus window (the window that receives the focus whenever the focus window is deleted) and the command returns an empty string. If *window* isn't specified, the command returns the path name of the current default focus window, or **none** if there is no default. *Window* may be specified as **none** to clear its existing value. The default window is initially **none**.

focus :none
Clears the focus window, so that keyboard input to this application will be discarded.

"Focus Events"

Tk's model of the input focus is different than X's model, and the focus window set with the **focus** command is not usually the same as the X focus window. Tk never explicitly changes the official X focus window. It waits for the window manager to direct the X input focus to and from the application's top-level windows, and it intercepts **FocusIn** and **FocusOut** events coming from the X server to detect these changes. All of the focus events received from X are discarded by Tk; they never reach the application. Instead, Tk generates a different stream of **FocusIn** and **FocusOut** for the application. This means that **FocusIn** and and **FocusOut** events seen by the application will not obey the conventions described in the documentation for Xlib.

Tk applications receive two kinds of **FocusIn** and **FocusOut** events, which can be distinguished by their *detail* fields. Events with a *detail* of **NotifyAncestor** are directed to the current focus window when it becomes active or inactive. A window is the active focus whenever two conditions are simultaneously true: (a) the window is the focus window for its application, and (b) some top-level window in the application has received the X focus. When this happens Tk generates a **FocusIn** event for the focus window with detail **NotifyAncestor**. When a window loses the active focus (either because the window manager removed the focus from the application or because the focus window changed within the application) then it receives a **FocusOut** event with detail **NotifyAncestor**.

The events described above are directed to the application's focus window regardless of which top-level window within the application has received the focus. The second kind of focus event is provided for applications that need to know which particular top-level window has the X focus. Tk generates **FocusIn** and **FocusOut** events with detail **NotifyVirtual** for top-level windows whenever they receive or lose the X focus. These events are generated regardless of which window in the application has the Tk input focus. They do not imply that keystrokes will be directed to the window that receives the event; they simply indicate which top-level window is active as far as the window manager is concerned. If a top-level window is also the application's focus window, then it will receive both **NotifyVirtual** and **NotifyAncestor** events when it receives or loses the X focus.

Tk does not generate the hierarchical chains of **FocusIn** and **FocusOut** events described in the Xlib documentation (e.g. a window can get a **FocusIn** or **FocusOut** event without all of its ancestors getting events too). Furthermore, the *mode* field in focus events is always **NotifyNormal** and the only values ever present in the *detail* field are **NotifyAncestor** and **NotifyVirtual**.

Keywords

events, focus, keyboard, top-level, window manager

3.7 grab

grab \- Confine pointer and keyboard events to a window sub-tree

Synopsis

grab ?**:global**? *window*
grab *option* ?arg arg ...?

Description

This command implements simple pointer and keyboard grabs for Tk. Tk's grabs are different than the grabs described in the Xlib documentation. When a grab is set for a particular window, Tk restricts all pointer events to the grab window and its descendants in Tk's window hierarchy. Whenever the pointer is within the grab window's subtree, the pointer will behave exactly the same as if there had been no grab at all and all events will be reported in the normal fashion. When the pointer is outside *window*'s tree, button presses and releases and mouse motion events are reported to *window*, and window entry and window exit events are ignored. The grab subtree "owns" the pointer: windows outside the grab subtree will be visible on the screen but they will be insensitive until the grab is released. The tree of windows underneath the grab window can include top-level windows, in which case all of those top-level windows and their descendants will continue to receive mouse events during the grab.

Two forms of grabs are possible: local and global. A local grab affects only the grabbing application: events will be reported to other applications as if the grab had never occurred. Grabs are local by default. A global grab locks out all applications on the screen, so that only the given subtree of the grabbing application will be sensitive to pointer events (mouse button presses, mouse button releases, pointer motions, window entries, and window exits). During global grabs the window manager will not receive pointer events either.

During local grabs, keyboard events (key presses and key releases) are delivered as usual: the window manager controls which application receives keyboard events, and if they are sent to any window in the grabbing application then they are redirected to the focus window. During a global grab Tk grabs the keyboard so that all keyboard events are always sent to the grabbing application. The **focus** command is still used to determine which window in the application receives the keyboard events. The keyboard grab is released when the grab is released.

Grabs apply to particular displays. If an application has windows on multiple displays then it can establish a separate grab on each display. The grab on a particular display affects only the windows on that display. It is possible for different applications on a single display to have simultaneous local grabs, but only one application can have a global grab on a given display at once.

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

grab ?**:global**? *window*
: Same as **grab :set**, described below.

grab :current ?*window*?
: If *window* is specified, returns the name of the current grab window in this application for *window*'s display, or an empty string if there is no such window. If *window* is omitted, the command returns a list whose elements are all of the windows grabbed by this application for all displays, or an empty string if the application has no grabs.

grab :release *window*
: Releases the grab on *window* if there is one, otherwise does nothing. Returns an empty string.

grab :set ?**:global**? *window*
: Sets a grab on *window*. If **:global** is specified then the grab is global, otherwise it is local. If a grab was already in effect for this application on *window*'s display then it is automatically released. If there is already a grab on *window* and it has the same global/local form as the requested grab, then the command does nothing. Returns an empty string.

grab :status *window*
: Returns **none** if no grab is currently set on *window*, **local** if a local grab is set on *window*, and **global** if a global grab is set.

Bugs

It took an incredibly complex and gross implementation to produce the simple grab effect described above. Given the current implementation, it isn't safe for applications to use the Xlib grab facilities at all except through the Tk grab procedures. If applications try to manipulate X's grab mechanisms directly, things will probably break.

If a single process is managing several different Tk applications, only one of those applications can have a local grab for a given display at any given time. If the applications are in different processes, this restriction doesn't exist.

Keywords

grab, keyboard events, pointer events, window

3.8 tk-listbox-single-select

tk-listbox-single-select \- Allow only one selected element in listbox(es)

Synopsis

tk-listbox-single-select *arg ?arg arg ...?*

Description

This command is a Tcl procedure provided as part of the Tk script library. It takes as arguments the path names of one or more listbox widgets, or the value **Listbox**. For each named widget, **tk-listbox-single-select** modifies the bindings of the widget so that only a single element may be selected at a time (the normal configuration allows multiple elements to be selected). If the keyword **Listbox** is among the *window* arguments, then the class bindings for listboxes are changed so that all listboxes have the one-selection-at-a-time behavior.

Keywords

listbox, selection

3.9 lower

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

3.10 tk-menu-bar

tk-menu-bar, tk_bindForTraversal \- Support for menu bars

Synopsis

tk-menu-bar *frame ?menu menu ...?*

tk_bindForTraversal *arg arg ...*

Description

These two commands are Tcl procedures in the Tk script library. They provide support for menu bars. A menu bar is a frame that contains a collection of menu buttons that work together, so that the user can scan from one menu to another with the mouse: if the mouse button is pressed over one menubutton (causing it to post its menu) and the mouse is moved over another menubutton in the same menu bar without releasing the mouse button, then the menu of the first menubutton is unposted and the menu of the new menubutton is posted instead. Menus in a menu bar can also be accessed using keyboard traversal (i.e. by typing keystrokes instead of using the mouse). In order for an application to use these procedures, it must do three things, which are described in the paragraphs below.

First, each application must call **tk-menu-bar** to provide information about the menubar. The *frame* argument gives the path name of the frame that contains all of the menu buttons, and the *menu* arguments give path names for all of the menu buttons associated with the menu bar. Normally *frame* is the parent of each of the *menu*'s. This need not be the case, but *frame* must be an ancestor of each of the *menu*'s in order for grabs to work correctly when the mouse is used to pull down menus. The order of the *menu* arguments determines the traversal order for the menu buttons. If **tk-menu-bar** is called without any *menu* arguments, it returns a list containing the current menu buttons for *frame*, or an empty string if *frame* isn't currently set up as a menu bar. If **tk-menu-bar** is called with a single *menu* argument consisting of an empty string, any menubar information for *frame* is removed; from now on the menu buttons will function independently without keyboard traversal. Only one menu bar may be defined at a time within each top-level window.

The second thing an application must do is to identify the traversal characters for menu buttons and menu entries. This is done by underlining those characters using the **:underline** options for the widgets. The menu traversal system uses this information to traverse the menus under keyboard control (see below).

The third thing that an application must do is to make sure that the input focus is always in a window that has been configured to support menu traversal. If the input focus is **none** then input characters will be discarded and no menu traversal will be possible. If you have no other place to set the focus, set it to the menubar widget: **tk-menu-bar** creates bindings for its *frame* argument to support menu traversal.

The Tk startup scripts configure all the Tk widget classes with bindings to support menu traversal, so menu traversal will be possible regardless of which widget has the focus. If your application defines new classes of widgets that support the input focus, then you should call **tk_bindForTraversal** for each of these classes. **Tk_bindForTraversal** takes any number of arguments, each of which is a widget path name or widget class name. It sets up bindings for all the named widgets and classes so that the menu traversal system will be invoked when appropriate keystrokes are typed in those widgets or classes.

"Menu Traversal Bindings"

Once an application has made the three arrangements described above, menu traversal will be available. At any given time, the only menus available for traversal are those associated with the top-level window containing the input focus. Menu traversal is initiated by one of the following actions:

[1] If <F10> is typed, then the first menu button in the list for the top-level window is posted and the first entry within that menu is selected.

[2] If <Alt-*key*> is pressed, then the menu button that has *key* as its underlined character is posted and the first entry within that menu is selected. The comparison between *key* and the underlined characters ignores case differences. If no menu button matches *key* then the keystroke has no effect.

[3] Clicking mouse button 1 on a menu button posts that menu and selects its first entry.

Once a menu has been posted, the input focus is switched to that menu and the following actions are possible:

[1] Typing <ESC> or clicking mouse button 1 outside the menu button or its menu will abort the menu traversal.

[2] If <Alt-*key*> is pressed, then the entry in the posted menu whose underlined character is *key* is invoked. This causes the menu to be unposted, the entry's action to be taken, and the menu traversal to end. The comparison between *key* and underlined characters ignores case differences. If no menu entry matches *key* then the keystroke is ignored.

[3] The arrow keys may be used to move among entries and menus. The left and right arrow keys move circularly among the available menus and the up and down arrow keys move circularly among the entries in the current menu.

[4] If <Return> is pressed, the selected entry in the posted menu is invoked, which causes the menu to be unposted, the entry's action to be taken, and the menu traversal to end.

When a menu traversal completes, the input focus reverts to the window that contained it when the traversal started.

Keywords

keyboard traversal, menu, menu bar, post

3.11 option

option \- Add/retrieve window options to/from the option database

Synopsis

option :add *pattern value ?priority?*

option :clear

option :get *window name class*

option :readfile *fileName ?priority?*

Description

The **option** command allows you to add entries to the Tk option database or to retrieve options from the database. The **add** form of the command adds a new option to the database. *Pattern* contains the option being specified, and consists of names and/or classes separated by asterisks or dots, in the usual X format. *Value* contains a text string to associate with *pattern*; this is the value that will be returned in calls to **Tk_GetOption** or by invocations of the **option :get** command. If *priority* is specified, it indicates the priority level for this option (see below for legal values); it defaults to **interactive**. This command always returns an empty string.

The **option :clear** command clears the option database. Default options (in the **RESOURCE_MANAGER** property or the **.Xdefaults** file) will be reloaded automatically the next time an option is added to the database or removed from it. This command always returns an empty string.

The **option :get** command returns the value of the option specified for *window* under *name* and *class*. If several entries in the option database match *window*, *name*, and *class*, then the command returns whichever was created with highest *priority* level. If there are several matching entries at the same priority level, then it returns whichever entry was most recently entered into the option database. If there are no matching entries, then the empty string is returned.

The **readfile** form of the command reads *fileName*, which should have the standard format for an X resource database such as **.Xdefaults**, and adds all the options specified in that file to the option database. If *priority* is specified, it indicates the priority level at which to enter the options; *priority* defaults to **interactive**.

The *priority* arguments to the **option** command are normally specified symbolically using one of the following values:

widgetDefault
: Level 20. Used for default values hard-coded into widgets.

startupFile
: Level 40. Used for options specified in application-specific startup files.

userDefault
: Level 60. Used for options specified in user-specific defaults files, such as **.Xdefaults**, resource databases loaded into the X server, or user-specific startup files.

interactive
: Level 80. Used for options specified interactively after the application starts running. If *priority* isn't specified, it defaults to this level.

Any of the above keywords may be abbreviated. In addition, priorities may be specified numerically using integers between 0 and 100, inclusive. The numeric form is probably a bad idea except for new priority levels other than the ones given above.

Keywords

database, option, priority, retrieve

3.12 options

options \- Standard options supported by widgets

Description

This manual entry describes the common configuration options supported by widgets in the Tk toolkit. Every widget does not necessarily support every option (see the manual entries for individual widgets for a list of the standard options supported by that widget), but if a widget does support an option with one of the names listed below, then the option has exactly the effect described below.

In the descriptions below, "Name" refers to the option's name in the option database (e.g. in .Xdefaults files). "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.

:activebackground

Name="**activeBackground**" Class="**Foreground**"

Specifies background color to use when drawing active elements. An element (a widget or portion of a widget) is active if the mouse cursor is positioned over the element and pressing a mouse button will cause some action to occur.

:activeborderwidth

Name="**activeBorderWidth**" Class="**BorderWidth**"

Specifies a non-negative value indicating the width of the 3-D border drawn around active elements. See above for definition of active elements. The value may have any of the forms acceptable to **Tk_GetPixels**. This option is typically only available in widgets displaying more than one element at a time (e.g. menus but not buttons).

:activeforeground

Name="**activeForeground**" Class="**Background**"

Specifies foreground color to use when drawing active elements. See above for definition of active elements.

:anchor

Name="**anchor**" Class="**Anchor**"

Specifies how the information in a widget (e.g. text or a bitmap) is to be displayed in the widget. Must be one of the values **n**, **ne**, **e**, **se**, **s**, **sw**, **w**, **nw**, or

center. For example, **nw** means display the information such that its top-left corner is at the top-left corner of the widget.

:background or :bg

Name="**background**" Class="**Background**"

Specifies the normal background color to use when displaying the widget.

:bitmap

Name="**bitmap**" Class="**Bitmap**"

Specifies a bitmap to display in the widget, in any of the forms acceptable to **Tk_GetBitmap**. The exact way in which the bitmap is displayed may be affected by other options such as **anchor** or **justify**. Typically, if this option is specified then it overrides other options that specify a textual value to display in the widget; the **bitmap** option may be reset to an empty string to re-enable a text display.

:borderwidth or :bd

Name="**borderWidth**" Class="**BorderWidth**"

Specifies a non-negative value indicating the width of the 3-D border to draw around the outside of the widget (if such a border is being drawn; the **relief** option typically determines this). The value may also be used when drawing 3-D effects in the interior of the widget. The value may have any of the forms acceptable to **Tk_GetPixels**.

:cursor

Name="**cursor**" Class="**Cursor**"

Specifies the mouse cursor to be used for the widget. The value may have any of the forms acceptable to **Tk_GetCursor**.

:cursorbackground

Name="**cursorBackground**" Class="**Foreground**"

Specifies the color to use as background in the area covered by the insertion cursor. This color will normally override either the normal background for the widget (or the selection background if the insertion cursor happens to fall in the selection). \fIThis option is obsolete and is gradually being replaced by the **insertBackground** option.

:cursorborderwidth

Name="**cursorBorderWidth**" Class="**BorderWidth**"

Specifies a non-negative value indicating the width of the 3-D border to draw around the insertion cursor. The value may have any of the forms acceptable to **Tk_GetPixels**. \fIThis option is obsolete and is gradually being replaced by the **insertBorderWidth** option.

:cursorofftime

Name="**cursorOffTime**" Class="**OffTime**"

Specifies a non-negative integer value indicating the number of milliseconds the cursor should remain "off" in each blink cycle. If this option is zero then the cursor doesn't blink: it is on all the time. \fIThis option is obsolete and is gradually being replaced by the **insertOffTime** option.

:cursorontime

Name="**cursorOnTime**" Class="**OnTime**"

Specifies a non-negative integer value indicating the number of milliseconds the cursor should remain "on" in each blink cycle. \fIThis option is obsolete and is gradually being replaced by the **insertOnTime** option.

:cursorwidth

Name="**cursorWidth**" Class="**CursorWidth**"

Specifies a value indicating the total width of the insertion cursor. The value may have any of the forms acceptable to **Tk_GetPixels**. If a border has been specified for the cursor (using the **cursorBorderWidth** option), the border will be drawn inside the width specified by the **cursorWidth** option. \fIThis option is obsolete and is gradually being replaced by the **insertWidth** option.

:disabledforeground

Name="**disabledForeground**" Class="**DisabledForeground**"

Specifies foreground color to use when drawing a disabled element. If the option is specified as an empty string (which is typically the case on monochrome displays), disabled elements are drawn with the normal fooreground color but they are dimmed by drawing them with a stippled fill pattern.

:exportselection

Name="**exportSelection**" Class="**ExportSelection**"

Specifies whether or not a selection in the widget should also be the X selection. The value may have any of the forms accepted by **Tcl_GetBoolean**, such as **true**, **false**, **0**, **1**, **yes**, or **no**. If the selection is exported, then selecting in the widget deselects the current X selection, selecting outside the widget deselects any widget selection, and the widget will respond to selection retrieval requests when it has a selection. The default is usually for widgets to export selections.

:font

Name="**font**" Class="**Font**"

Specifies the font to use when drawing text inside the widget.

:foreground or :fg

Name="**foreground**" Class="**Foreground**"

Specifies the normal foreground color to use when displaying the widget.

:geometry

Name="**geometry**" Class="**Geometry**"

Specifies the desired geometry for the widget's window, in the form *width***x***height*, where *width* is the desired width of the window and *height* is the desired height. The units for *width* and *height* depend on the particular widget. For widgets displaying text the units are usually the size of the characters in the font being displayed; for other widgets the units are usually pixels.

:insertbackground

Name="**insertBackground**" Class="**Foreground**"

Specifies the color to use as background in the area covered by the insertion cursor. This color will normally override either the normal background for the widget (or the selection background if the insertion cursor happens to fall in the selection).

:insertborderwidth

Name="**insertBorderWidth**" Class="**BorderWidth**"

Specifies a non-negative value indicating the width of the 3-D border to draw around the insertion cursor. The value may have any of the forms acceptable to **Tk_GetPixels**.

:insertofftime

Name="**insertOffTime**" Class="**OffTime**"

Specifies a non-negative integer value indicating the number of milliseconds the insertion cursor should remain "off" in each blink cycle. If this option is zero then the cursor doesn't blink: it is on all the time.

:insertontime

Name="**insertOnTime**" Class="**OnTime**"

Specifies a non-negative integer value indicating the number of milliseconds the insertion cursor should remain "on" in each blink cycle.

:insertwidth

Name="**insertWidth**" Class="**InsertWidth**"

Specifies a value indicating the total width of the insertion cursor. The value may have any of the forms acceptable to **Tk_GetPixels**. If a border has been specified for the insertion cursor (using the **insertBorderWidth** option), the border will be drawn inside the width specified by the **insertWidth** option.

:orient

Name="**orient**" Class="**Orient**"

For widgets that can lay themselves out with either a horizontal or vertical orientation, such as scrollbars, this option specifies which orientation should be used. Must be either **horizontal** or **vertical** or an abbreviation of one of these.

:padx

Name="**padX**" Class="**Pad**"

Specifies a non-negative value indicating how much extra space to request for the widget in the X-direction. The value may have any of the forms acceptable to **Tk_GetPixels**. When computing how large a window it needs, the widget will add this amount to the width it would normally need (as determined by the width of the things displayed in the widget); if the geometry manager can satisfy this request, the widget will end up with extra internal space to the left and/or right of what it displays inside.

:pady

Name="**padY**" Class="**Pad**"

Specifies a non-negative value indicating how much extra space to request for the widget in the Y-direction. The value may have any of the forms acceptable to **Tk_GetPixels**. When computing how large a window it needs, the widget will add this amount to the height it would normally need (as determined by the height of the things displayed in the widget); if the geometry manager can satisfy this request, the widget will end up with extra internal space above and/or below what it displays inside.

:relief

Name="**relief**" Class="**Relief**"

Specifies the 3-D effect desired for the widget. Acceptable values are **raised**, **sunken**, **flat**, **ridge**, and **groove**. The value indicates how the interior of the widget should appear relative to its exterior; for example, **raised** means the interior of the widget should appear to protrude from the screen, relative to the exterior of the widget.

:repeatdelay

Name="**repeatDelay**" Class="**RepeatDelay**"

Specifies the number of milliseconds a button or key must be held down before it begins to auto-repeat. Used, for example, on the up- and down-arrows in scrollbars.

:repeatinterval

Name="**repeatInterval**" Class="**RepeatInterval**"

Used in conjunction with **repeatDelay**: once auto-repeat begins, this option determines the number of milliseconds between auto-repeats.

:scrollcommand

Name=**"scrollCommand" Class="ScrollCommand"**

Specifies the prefix for a command used to communicate with scrollbar widgets. 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 four numbers. The four numbers are, in order: the total size of the widget's contents, in unspecified units ("unit" is a widget-specific term; for widgets displaying text, the unit is a line); the maximum number of units that may be displayed at once in the widget's window, given its current size; the index of the top-most or left-most unit currently visible in the window (index 0 corresponds to the first unit); and the index of the bottom-most or right-most unit currently visible in the window. This command is then passed to the Tcl interpreter for execution. Typically the **scrollCommand** 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.

The **scrollCommand** option is used for widgets that support scrolling in only one direction. For widgets that support scrolling in both directions, this option is replaced with the **xScrollCommand** and **yScrollCommand** options.

:selectbackground

Name=**"selectBackground" Class="Foreground"**

Specifies the background color to use when displaying selected items.

:selectborderwidth

Name=**"selectBorderWidth" Class="BorderWidth"**

Specifies a non-negative value indicating the width of the 3-D border to draw around selected items. The value may have any of the forms acceptable to **Tk_GetPixels**.

:selectforeground

Name=**"selectForeground" Class="Background"**

Specifies the foreground color to use when displaying selected items.

:setgrid

Name=**"setGrid" Class="SetGrid"**

Specifies a boolean value that determines whether this widget controls the resizing grid for its top-level window. This option is typically used in text widgets,

where the information in the widget has a natural size (the size of a character) and it makes sense for the window's dimensions to be integral numbers of these units. These natural window sizes form a grid. If the **setGrid** option is set to true then the widget will communicate with the window manager so that when the user interactively resizes the top-level window that contains the widget, the dimensions of the window will be displayed to the user in grid units and the window size will be constrained to integral numbers of grid units. See the section GRIDDED GEOMETRY MANAGEMENT in the **wm** manual entry for more details.

:text

Name=**"text" Class="Text"**

Specifies a string to be displayed inside the widget. The way in which the string is displayed depends on the particular widget and may be determined by other options, such as **anchor** or **justify**.

:textvariable

Name=**"textVariable" Class="Variable"**

Specifies the name of a variable. The value of the variable is a text string to be displayed inside the widget; if the variable value changes then the widget will automatically update itself to reflect the new value. The way in which the string is displayed in the widget depends on the particular widget and may be determined by other options, such as **anchor** or **justify**.

:underline

Name=**"underline" Class="Underline"**

Specifies the integer index of a character to underline in the widget. This option is typically used to indicate keyboard traversal characters in 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.

:xscrollcommand

Name=**"xScrollCommand" Class="ScrollCommand"**

Specifies the prefix for a command used to communicate with horizontal scrollbars. This option is treated in the same way as the **scrollCommand** option, except that it is used for horizontal scrollbars associated with widgets that support both horizontal and vertical scrolling. See the description of **scrollCommand** for complete details on how this option is used.

:yscrollcommand

Name=**"yScrollCommand" Class="ScrollCommand"**

Specifies the prefix for a command used to communicate with vertical scrollbars. This option is treated in the same way as the **scrollCommand** option, except

that it is used for vertical scrollbars associated with widgets that support both horizontal and vertical scrolling. See the description of **scrollCommand** for complete details on how this option is used.

Keywords

class, name, standard option, switch

3.13 pack-old

pack \- Obsolete syntax for packer geometry manager

Synopsis

pack after *sibling window options ?window options ...?*

pack append *parent window options ?window options ...?*

pack before *sibling window options ?window options ...?*

pack info *parent*

pack unpack *window*

Description

Note: this manual entry describes the syntax for the **pack**fI command as it before Tk version 3.3. Although this syntax continues to be supported for backward compatibility, it is obsolete and should not be used anymore. At some point in the future it may cease to be supported.

The packer is a geometry manager that arranges the children of a parent by packing them in order around the edges of the parent. The first child is placed against one side of the window, occupying the entire span of the window along that side. This reduces the space remaining for other children as if the side had been moved in by the size of the first child. Then the next child is placed against one side of the remaining cavity, and so on until all children have been placed or there is no space left in the cavity.

The **before**, **after**, and **append** forms of the **pack** command are used to insert one or more children into the packing order for their parent. The **before** form inserts the children before window *sibling* in the order; all of the other windows must be siblings of *sibling*. The **after** form inserts the windows after *sibling*, and the **append** form appends one or more windows to the end of the packing order for *parent*. If a *window* named in any of these commands is already packed in its parent, it is removed from its current position in the packing order and repositioned as indicated by the command. All of these commands return an empty string as result.

The **unpack** form of the **pack** command removes *window* from the packing order of its parent and unmaps it. After the execution of this command the packer will no longer manage *window*'s geometry.

The placement of each child is actually a four-step process; the *options* argument following each *window* consists of a list of one or more fields that govern the placement of that window. In the discussion below, the term *cavity* refers to the space left in a parent when a particular child is placed (i.e. all the space that wasn't claimed by earlier children in the packing order). The term *parcel* refers to the space allocated to a particular child; this is not necessarily the same as the child window's final geometry.

The first step in placing a child is to determine which side of the cavity it will lie against. Any one of the following options may be used to specify a side:

top Position the child's parcel against the top of the cavity, occupying the full width of the cavity.

bottom Position the child's parcel against the bottom of the cavity, occupying the full width of the cavity.

left Position the child's parcel against the left side of the cavity, occupying the full height of the cavity.

right Position the child's parcel against the right side of the cavity, occupying the full height of the cavity.

At most one of these options should be specified for any given window. If no side is specified, then the default is **top**.

The second step is to decide on a parcel for the child. For **top** and **bottom** windows, the desired parcel width is normally the cavity width and the desired parcel height is the window's requested height, as passed to **Tk_GeometryRequest**. For **left** and **right** windows, the desired parcel height is normally the cavity height and the desired width is the window's requested width. However, extra space may be requested for the window using any of the following options:

padx *num* Add *num* pixels to the window's requested width before computing the parcel size as described above.

pady *num* Add *num* pixels to the window's requested height before computing the parcel size as described above.

expand This option requests that the window's parcel absorb any extra space left over in the parent's cavity after packing all the children. The amount of space left over depends on the sizes requested by the other children, and may be zero. If several windows have all specified **expand** then the extra width will be divided equally among all the **left** and **right** windows that specified **expand** and the extra height will be divided equally among all the **top** and **bottom** windows that specified **expand**.

If the desired width or height for a parcel is larger than the corresponding dimension of the cavity, then the cavity's dimension is used instead.

The third step in placing the window is to decide on the window's width and height. The default is for the window to receive either its requested width and height or the those of the parcel, whichever is smaller. If the parcel is larger than the window's requested size, then the following options may be used to expand the window to partially or completely fill the parcel:

fill Set the window's size to equal the parcel size.

fillx Increase the window's width to equal the parcel's width, but retain the window's requested height.

filly Increase the window's height to equal the parcel's height, but retain the window's requested width.

The last step is to decide the window's location within its parcel. If the window's size equals the parcel's size, then the window simply fills the entire parcel. If the parcel is larger than the window, then one of the following options may be used to specify where the window should be positioned within its parcel:

frame center
Center the window in its parcel. This is the default if no framing option is specified.

frame n Position the window with its top edge centered on the top edge of the parcel.

frame ne Position the window with its upper-right corner at the upper-right corner of the parcel.

frame e Position the window with its right edge centered on the right edge of the parcel.

frame se Position the window with its lower-right corner at the lower-right corner of the parcel.

frame s Position the window with its bottom edge centered on the bottom edge of the parcel.

frame sw Position the window with its lower-left corner at the lower-left corner of the parcel.

frame w Position the window with its left edge centered on the left edge of the parcel.

frame nw Position the window with its upper-left corner at the upper-left corner of the parcel.

The **pack info** command may be used to retrieve information about the packing order for a parent. It returns a list in the form

window options window options ...

Each *window* is a name of a window packed in *parent*, and the following *options* describes all of the options for that window, just as they would be typed to **pack append**. The order of the list is the same as the packing order for *parent*.

The packer manages the mapped/unmapped state of all the packed children windows. It automatically maps the windows when it packs them, and it unmaps any windows for which there was no space left in the cavity.

The packer makes geometry requests on behalf of the parent windows it manages. For each parent window it requests a size large enough to accommodate all the options specified by all the packed children, such that zero space would be leftover for **expand** options.

Keywords

geometry manager, location, packer, parcel, size

3.14 pack

pack \- Geometry manager that packs around edges of cavity

Synopsis

pack *option arg ?arg ...?*

Description

The **pack** command is used to communicate with the packer, a geometry manager that arranges the children of a parent by packing them in order around the edges of the parent. The **pack** command can have any of several forms, depending on the *option* argument:

pack *slave ?slave ...? ?options?*
If the first argument to **pack** is a window name (any value starting with "."), then the command is processed in the same way as **pack configure**.

pack configure *slave ?slave ...? ?options?*
The arguments consist of the names of one or more slave windows followed by pairs of arguments that specify how to manage the slaves. See "THE PACKER ALGORITHM" below for details on how the options are used by the packer. The following options are supported:

:after *other*
Other must the name of another window. Use its master as the master for the slaves, and insert the slaves just after *other* in the packing order.

:anchor *anchor*
Anchor must be a valid anchor position such as **n** or **sw**; it specifies where to position each slave in its parcel. Defaults to **center**.

:before *other*
Other must the name of another window. Use its master as the master for the slaves, and insert the slaves just before *other* in the packing order.

:expand *boolean*
Specifies whether the slaves should be expanded to consume extra space in their master. *Boolean* may have any proper boolean value, such as **1** or **no**. Defaults to 0.

:fill *style* If a slave's parcel is larger than its requested dimensions, this option may be used to stretch the slave. *Style* must have one of the following values:

none	Give the slave its requested dimensions plus any internal padding requested with **:ipadx** or **:ipady**. This is the default.
x	Stretch the slave horizontally to fill the entire width of its parcel (except leave external padding as specified by **:padx**).
y	Stretch the slave vertically to fill the entire height of its parcel (except leave external padding as specified by **:pady**).
both	Stretch the slave both horizontally and vertically.

:in *other* Insert the slave(s) at the end of the packing order for the master window given by *other*.

:ipadx *amount*
Amount specifies how much horizontal internal padding to leave on each side of the slave(s). *Amount* must be a valid screen distance, such as **2** or **.5c**. It defaults to 0.

:ipady *amount*
Amount specifies how much vertical internal padding to leave on each side of the slave(s). *Amount* defaults to 0.

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

:pady *amount*
Amount specifies how much vertical external padding to leave on each side of the slave(s). *Amount* defaults to 0.

:side *side* Specifies which side of the master the slave(s) will be packed against. Must be **left**, **right**, **top**, or **bottom**. Defaults to **top**.

If no **:in**, **:after** or **:before** option is specified then each of the slaves will be inserted at the end of the packing list for its parent unless it is already managed by the packer (in which case it will be left where it is). If one of these options is specified then all the slaves will be inserted at the specified point. If any of the slaves are already managed by the geometry manager then any unspecified options for them retain their previous values rather than receiving default values. .RE

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 :newinfo *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. Starting with Tk 4.0 this option will be renamed "pack info".

pack :propagate *master ?boolean?*
If *boolean* has a true boolean value such as **1** or **on** then propagation is enabled for *master*, which must be a window name (see "GEOMETRY PROPAGATION" below). If *boolean* has a false boolean value then propagation is disabled for *master*. In either of these cases an empty string is returned. If *boolean* is omitted then the command returns **0** or **1** to indicate whether propagation is currently enabled for *master*. Propagation is enabled by default.

pack :slaves *master*
Returns a list of all of the slaves in the packing order for *master*. The order of the slaves in the list is the same as their order in the packing order. If *master* has no slaves then an empty string is returned.

"The Packer Algorithm"

For each master the packer maintains an ordered list of slaves called the *packing list*. The **:in**, **:after**, and **:before** configuration options are used to specify the master for each slave and the slave's position in the packing list. If none of these options is given for a slave then the slave is added to the end of the packing list for its parent.

The packer arranges the slaves for a master by scanning the packing list in order. At the time it processes each slave, a rectangular area within the master is still unallocated. This area is called the *cavity*; for the first slave it is the entire area of the master.

For each slave the packer carries out the following steps:

[1] The packer allocates a rectangular *parcel* for the slave along the side of the cavity given by the slave's **:side** option. If the side is top or bottom then the width of the parcel is the width of the cavity and its height is the requested height of the slave plus the **:ipady** and **:pady** options. For the left or right side the height of the parcel is the height of the cavity and the width is the requested width of the slave plus the **:ipadx** and **:padx** options. The parcel may be enlarged further because of the **:expand** option (see "EXPANSION" below)

[2] The packer chooses the dimensions of the slave. The width will normally be the slave's requested width plus twice its **:ipadx** option and the height will normally be the slave's requested height plus twice its **:ipady** option. However, if the **:fill** option is **x** or **both** then the width of the slave is expanded to fill the width of the parcel, minus twice the **:padx** option. If the **:fill** option is **y** or **both** then the height of the slave is expanded to fill the width of the parcel, minus twice the **:pady** option.

[3] The packer positions the slave over its parcel. If the slave is smaller than the parcel then the **:anchor** option determines where in the parcel the slave will be placed. If **:padx** or **:pady** is non-zero, then the given amount of external padding will always be left between the slave and the edges of the parcel.

Once a given slave has been packed, the area of its parcel is subtracted from the cavity, leaving a smaller rectangular cavity for the next slave. If a slave doesn't use all of its parcel, the unused space in the parcel will not be used by subsequent slaves. If the cavity should become too small to meet the needs of a slave then the slave will be given whatever space is left in the cavity. If the cavity shrinks to zero size, then all remaining slaves on the packing list will be unmapped from the screen until the master window becomes large enough to hold them again.

"Expansion"

If a master window is so large that there will be extra space left over after all of its slaves have been packed, then the extra space is distributed uniformly among all of the slaves for which the **:expand** option is set. Extra horizontal space is distributed among the expandable slaves whose **:side** is **left** or **right**, and extra vertical space is distributed among the expandable slaves whose **:side** is **top** or **bottom**.

"Geometry Propagation"

The packer normally computes how large a master must be to just exactly meet the needs of its slaves, and it sets the requested width and height of the master to these dimensions.

This causes geometry information to propagate up through a window hierarchy to a top-level window so that the entire sub-tree sizes itself to fit the needs of the leaf windows. However, the **pack propagate** command may be used to turn off propagation for one or more masters. If propagation is disabled then the packer will not set the requested width and height of the packer. This may be useful if, for example, you wish for a master window to have a fixed size that you specify.

"Restrictions On Master Windows"

The master for each slave must either be the slave's parent (the default) or a descendant of the slave's parent. This restriction is necessary to guarantee that the slave can be placed over any part of its master that is visible without danger of the slave being clipped by its parent.

"Packing Order"

If the master for a slave is not its parent then you must make sure that the slave is higher in the stacking order than the master. Otherwise the master will obscure the slave and it will appear as if the slave hasn't been packed correctly. The easiest way to make sure the slave is higher than the master is to create the master window first: the most recently created window will be highest in the stacking order. Or, you can use the **raise** and **lower** commands to change the stacking order of either the master or the slave.

Keywords

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

3.15 place

place \- Geometry manager for fixed or rubber-sheet placement

Synopsis

place *window option value ?option value ...?*

place configure *window option value ?option value ...?*

place forget *window*

place info *window*

place slaves *window*

Description

The placer is a geometry manager for Tk. It provides simple fixed placement of windows, where you specify the exact size and location of one window, called the *slave*, within another window, called the *master*. The placer also provides rubber-sheet placement, where you specify the size and location of the slave in terms of the dimensions of the master, so that the slave changes size and location in response to changes in the size of the master. Lastly, the placer allows you to mix these styles of placement so that, for example, the slave has a fixed width and height but is centered inside the master.

If the first argument to the **place** command is a window path name or **configure** then the command arranges for the placer to manage the geometry of a slave whose path name is *window*. The remaining arguments consist of one or more *option:value* pairs that specify the way in which *window*'s geometry is managed. If the placer is already managing *window*, then the *option:value* pairs modify the configuration for *window*. In this form the **place** command returns an empty string as result. The following *option:value* pairs are supported:

:in *master* *Master* specifes the path name of the window relative to which *window* is to be placed. *Master* must either be *window*'s parent or a descendant of *window*'s parent. In addition, *master* and *window* must both be descendants of the same top-level window. These restrictions are necessary to guarantee that *window* is visible whenever *master* is visible. If this option isn't specified then the master defaults to *window*'s parent.

:x *location* *Location* specifies the x-coordinate within the master window of the anchor point for *window*. The location is specified in screen units (i.e. any of the forms accepted by **Tk_GetPixels**) and need not lie within the bounds of the master window.

:relx *location*
Location specifies the x-coordinate within the master window of the anchor point for *window*. In this case the location is specified in a relative fashion as a floating-point number: 0.0 corresponds to the left edge of the master and 1.0 corresponds to the right edge of the master. *Location* need not be in the range 0.0\-1.0.

:y *location* *Location* specifies the y-coordinate within the master window of the anchor point for *window*. The location is specified in screen units (i.e. any of the forms accepted by **Tk_GetPixels**) and need not lie within the bounds of the master window.

:rely *location*
Location specifies the y-coordinate within the master window of the anchor point for *window*. In this case the value is specified in a relative fashion as a floating-point number: 0.0 corresponds to the top edge of the master and 1.0 corresponds to the bottom edge of the master. *Location* need not be in the range 0.0\-1.0.

:anchor *where*
Where specifies which point of *window* is to be positioned at the (x,y) location selected by the **:x**, **:y**, **:relx**, and **:rely** options. The anchor point is in terms

of the outer area of *window* including its border, if any. Thus if *where* is **se** then the lower-right corner of *window*'s border will appear at the given (x,y) location in the master. The anchor position defaults to **nw**.

:width *size*
: *Size* specifies the width for *window* in screen units (i.e. any of the forms accepted by **Tk_GetPixels**). The width will be the outer width of *window* including its border, if any. If *size* is an empty string, or if no **:width** or **:relwidth** option is specified, then the width requested internally by the window will be used.

:relwidth *size*
: *Size* specifies the width for *window*. In this case the width is specified as a floating-point number relative to the width of the master: 0.5 means *window* will be half as wide as the master, 1.0 means *window* will have the same width as the master, and so on.

:height *size*
: *Size* specifies the height for *window* in screen units (i.e. any of the forms accepted by **Tk_GetPixels**). The height will be the outer dimension of *window* including its border, if any. If *size* is an empty string, or if no **:height** or **:relheight** option is specified, then the height requested internally by the window will be used.

:relheight *size*
: *Size* specifies the height for *window*. In this case the height is specified as a floating-point number relative to the height of the master: 0.5 means *window* will be half as high as the master, 1.0 means *window* will have the same height as the master, and so on.

:bordermode *mode*
: *Mode* determines the degree to which borders within the master are used in determining the placement of the slave. The default and most common value is **inside**. In this case the placer considers the area of the master to be the innermost area of the master, inside any border: an option of **:x 0** corresponds to an x-coordinate just inside the border and an option of **:relwidth 1.0** means *window* will fill the area inside the master's border. If *mode* is **outside** then the placer considers the area of the master to include its border; this mode is typically used when placing *window* outside its master, as with the options **:x 0 :y 0 :anchor ne**. Lastly, *mode* may be specified as **ignore**, in which case borders are ignored: the area of the master is considered to be its official X area, which includes any internal border but no external border. A bordermode of **ignore** is probably not very useful.

  If the same value is specified separately with two different options, such as **:x** and **:relx**, then the most recent option is used and the older one is ignored.

  The **place slaves** command returns a list of all the slave windows for which *window* is the master. If there are no slaves for *window* then an empty string is returned.

  The **place forget** command causes the placer to stop managing the geometry of *window*. As a side effect of this command *window* will be unmapped so that it

doesn't appear on the screen. If *window* isn't currently managed by the placer then the command has no effect. **Place forget** returns an empty string as result.

The **place info** command returns a list giving the current configuration of *window*. The list consists of *option:value* pairs in exactly the same form as might be specified to the **place configure** command. If the configuration of a window has been retrieved with **place info**, that configuration can be restored later by first using **place forget** to erase any existing information for the window and then invoking **place configure** with the saved information.

"Fine Points"

It is not necessary for the master window to be the parent of the slave window. This feature is useful in at least two situations. First, for complex window layouts it means you can create a hierarchy of subwindows whose only purpose is to assist in the layout of the parent. The "real children" of the parent (i.e. the windows that are significant for the application's user interface) can be children of the parent yet be placed inside the windows of the geometry-management hierarchy. This means that the path names of the "real children" don't reflect the geometry-management hierarchy and users can specify options for the real children without being aware of the structure of the geometry-management hierarchy.

A second reason for having a master different than the slave's parent is to tie two siblings together. For example, the placer can be used to force a window always to be positioned centered just below one of its siblings by specifying the configuration

:in *sibling* **:relx 0.5 :rely 1.0 :anchor n :bordermode outside**

Whenever the sibling is repositioned in the future, the slave will be repositioned as well.

Unlike many other geometry managers (such as the packer) the placer does not make any attempt to manipulate the geometry of the master windows or the parents of slave windows (i.e. it doesn't set their requested sizes). To control the sizes of these windows, make them windows like frames and canvases that provide configuration options for this purpose.

Keywords

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

3.16 raise

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

3.17 selection

selection \- Manipulate the X selection

Synopsis

selection *option ?arg arg ...?*

Description

This command provides a Tcl interface to the X selection mechanism and implements the full selection functionality described in the X Inter-Client Communication Conventions Manual (ICCCM), except that it supports only the primary selection.

The first argument to **selection** determines the format of the rest of the arguments and the behavior of the command. The following forms are currently supported:

selection :clear *window*
: If there is a selection anywhere on *window*'s display, clear it so that no window owns the selection anymore. Returns an empty string.

selection :get *?type?*
: Retrieves the value of the primary selection and returns it as a result. **Type** specifies the form in which the selection is to be returned (the desired "target" for conversion, in ICCCM terminology), and should be an atom name such as STRING or FILE_NAME; see the Inter-Client Communication Conventions Manual for complete details. **Type** defaults to STRING. The selection :owner may choose to return the selection in any of several different representation formats, such as STRING, ATOM, INTEGER, etc. (this format is different than the selection type; see the ICCCM for all the confusing details). If the selection is returned in a non-string format, such as INTEGER or ATOM, the **selection** command converts it to string format as a collection of fields separated by spaces: atoms are converted to their textual names, and anything else is converted to hexadecimal integers.

selection :handle *window command ?type? ?format?*
: Creates a handler for selection requests, such that *command* will be executed whenever the primary selection is owned by *window* and someone attempts to retrieve it in the form given by *type* (e.g. *type* is specified in the **selection :get** command). *Type* defaults to STRING. If *command* is an empty string then any existing handler for *window* and *type* is removed.

When the selection is requested and *window* is the selection :owner and *type* is the requested type, *command* will be executed as a Tcl command with two additional numbers appended to it (with space separators). The two additional numbers are *offset* and *maxBytes*: *offset* specifies a starting character position in the selection and *maxBytes* gives the maximum number of bytes to retrieve. The command should return a value consisting of at most *maxBytes* of the selection, starting at position *offset*. For very large selections (larger than *maxBytes*) the selection will be retrieved using several invocations of *command* with increasing *offset* values. If *command* returns a string whose length is less than *maxBytes*, the return value is assumed to include all of the remainder of the selection; if the length of *command*'s result is equal to *maxBytes* then *command* will be invoked again, until it eventually returns a result shorter than *maxBytes*. The value of *maxBytes* will always be relatively large (thousands of bytes).

If *command* returns an error then the selection retrieval is rejected just as if the selection didn't exist at all.

The *format* argument specifies the representation that should be used to transmit the selection to the requester (the second column of Table 2 of the ICCCM), and defaults to STRING. If *format* is STRING, the selection is transmitted as 8-bit ASCII characters (i.e. just in the form returned by *command*). If *format* is ATOM, then the return value from *command* is divided into fields separated by white space; each field is converted to its atom value, and the 32-bit atom value is transmitted instead of the atom name. For any other *format*, the return value from *command* is divided into fields separated by white space and each field is converted to a 32-bit integer; an array of integers is transmitted to the selection requester.

The *format* argument is needed only for compatibility with selection requesters that don't use Tk. If the Tk toolkit is being used to retrieve the selection then the value is converted back to a string at the requesting end, so *format* is irrelevant. .RE

selection :own ?*window*? ?*command*?

If *window* is specified, then it becomes the new selection :owner and the command returns an empty string as result. The existing owner, if any, is notified that it has lost the selection. If *command* is specified, it is a Tcl script to execute when some other window claims ownership of the selection away from *window*. If neither *window* nor *command* is specified then the command returns the path name of the window in this application that owns the selection, or an empty string if no window in this application owns the selection.

Keywords

clear, format, handler, ICCCM, own, selection, target, type

3.18 send

send \- Execute a command in a different interpreter

Synopsis

send *interp cmd ?arg arg ...?*

Description

This command arranges for *cmd* (and *arg*s) to be executed in the interpreter named by *interp*. It returns the result or error from that command execution. *Interp* must be the name of an interpreter registered on the display associated with the interpreter in which the command is invoked; it need not be within the same process or application. If no *arg* arguments are present, then the command to be executed is contained entirely within the *cmd* argument. If one or more *arg*s are present, they are concatenated to form the command to be executed, just as for the **eval** Tcl command.

Security

The **send** command is potentially a serious security loophole, since any application that can connect to your X server can send scripts to your applications. These incoming scripts can use Tcl to read and write your files and invoke subprocesses under your name. Host-based access control such as that provided by **xhost** is particularly insecure, since it allows anyone with an account on particular hosts to connect to your server, and if disabled it allows anyone anywhere to connect to your server. In order to provide at least a small amount of security, Tk checks the access control being used by the server and rejects incoming sends unless (a) **xhost**-style access control is enabled (i.e. only certain hosts can establish connections) and (b) the list of enabled hosts is empty. This means that applications cannot connect to your server unless they use some other form of authorization such as that provide by **xauth**.

Keywords

interpreter, remote execution, security, send

3.19 tk

tk \- Manipulate Tk internal state

Synopsis

tk *option ?arg arg ...?*

Description

The **tk** command provides access to miscellaneous elements of Tk's internal state. Most of the information manipulated by this command pertains to the application as a whole, or to a screen or display, rather than to a particular window. The command can take any of a number of different forms depending on the *option* argument. The legal forms are:

tk :colormodel *window* ?*newValue*?

If *newValue* isn't specified, this command returns the current color model in use for *window*'s screen, which will be either **color** or **monochrome**. If *newValue* is specified, then it must be either **color** or **monochrome** or an abbreviation of one of them; the color model for *window*'s screen is set to this value.

The color model is used by Tk and its widgets to determine whether it should display in black and white only or use colors. A single color model is shared by all of the windows managed by one process on a given screen. The color model for a screen is set initially by Tk to **monochrome** if the display has four or fewer bit planes and to **color** otherwise. The color model will automatically be changed from **color** to **monochrome** if Tk fails to allocate a color because all entries in the colormap were in use. An application can change its own color model at any time (e.g. it might change the model to **monochrome** in order to conserve colormap entries, or it might set the model to **color** to use color on a four-bit display in special circumstances), but an application is not allowed to change the color model to **color** unless the screen has at least two bit planes. .RE

Keywords

color model, internal state

3.20 tkerror

tkerror \- Command invoked to process background errors

Synopsis

tkerror *message*

Description

The **tkerror** command doesn't exist as built-in part of Tk. 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** of **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 Tk library and there is no obvious way for Tk to report the error.

When Tk detects a background error, it invokes the **tkerror** command, passing it the error message as its only argument. Tk 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. Tk will ignore any result returned by the **tkerror** command.

If another Tcl error occurs within the **tkerror** command then Tk reports the error itself by writing a message to stderr.

The Tk 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 that shows where the error occurred.

Keywords

background error, reporting

3.21 tkvars

tkvars \- Variables used or set by Tk

Description

The following Tcl variables are either set or used by Tk at various times in its execution:

tk_library Tk sets this variable hold the name of a directory containing a library of Tcl scripts related to Tk. These scripts include an initialization file that is normally processed whenever a Tk application starts up, plus other files containing procedures that implement default behaviors for widgets. The value of this variable is taken from the TK_LIBRARY environment variable, if one exists, or else from a default value compiled into Tk.

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

tk_priv This variable is an array containing several pieces of information that are private to Tk. The elements of **tk_priv** are used by Tk library procedures and default bindings. They should not be accessed by any code outside Tk.

tk_strictMotif
This variable is set to zero by default. If an application sets it to one, then Tk attempts to adhere as closely as possible to Motif look-and-feel standards. For example, active elements such as buttons and scrollbar sliders will not change color when the pointer passes over them.

tk_version Tk sets this variable in the interpreter for each application. The variable holds the current version number of the Tk library in the form *major.minor*. *Major* and *minor* are integers. The major version number increases in any Tk release that includes changes that are not backward compatible (i.e. whenever existing Tk applications and scripts may have to change to work with the new release). The minor version number increases with each new release of Tk, except that it resets to zero whenever the major version number changes.

tkVersion Has the same value as **tk_version**. This variable is obsolete and will be deleted soon.

Keywords

variables, version

3.22 tkwait

tkwait \- Wait for variable to change or window to be destroyed

Synopsis

tkwait :variable *name*

tkwait :visibility *name*
tkwait :window *name*

Description

The **tkwait** command waits for one of several things to happen, then it returns without taking any other actions. The return value is always an empty string. If the first argument is **:variable** (or any abbreviation of it) then the second argument is the name of a global variable and the command waits for that variable to be modified. If the first argument is **:visibility** (or any abbreviation of it) then the second argument is the name of a window and the **tkwait** command waits for a change in its visibility state (as indicated by the arrival of a VisibilityNotify event). This form is typically used to wait for a newly-created window to appear on the screen before taking some action. If the first argument is **:window** (or any abbreviation of it) then the second argument is the name of a window and the **tkwait** command waits for that window to be destroyed. This form is typically used to wait for a user to finish interacting with a dialog box before using the result of that interaction.

While the **tkwait** command is waiting it processes events in the normal fashion, so the application will continue to respond to user interactions.

Keywords

variable, visibility, wait, window

3.23 update

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

Synopsis

update ?**:idletasks**?

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. 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

3.24 winfo

winfo \- Return window-related information

Synopsis

winfo *option ?arg arg ...?*

Description

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

winfo :atom *name*
: Returns a decimal string giving the integer identifier for the atom whose name is *name*. If no atom exists with the name *name* then a new one is created.

winfo :atomname *id*
: Returns the textual name for the atom whose integer identifier is *id*. This command is the inverse of the **winfo :atom** command. Generates an error if no such atom exists.

winfo :cells *window*
: Returns a decimal string giving the number of cells in the color map for *window*.

winfo :children *window*
: Returns a list containing the path names of all the children of *window*. 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 in screen units (i.e. any form acceptable

to **Tk_GetPixels**) in the coordinate system of the root window (if a virtual-root window manager is in use then the coordinate system of the virtual root window is used). If no window in this application contains the point then an empty string is returned. In selecting the containing window, children are given higher priority than parents and among siblings the highest one in the stacking order is chosen.

winfo :depth *window*
Returns a decimal string giving the depth of *window* (number of bits per pixel).

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

winfo :fpixels *window number*
Returns a floating-point value giving the number of pixels in *window* corresponding to the distance given by *number*. *Number* may be specified in any of the forms acceptable to **Tk_GetScreenMM**, such as "2.0c" or "1i". The return value may be fractional; for an integer value, use **winfo :pixels**.

winfo :geometry *window*
Returns the geometry for *window*, in the form *width***x***height*+*x*+*y*. All dimensions are in pixels.

winfo :height *window*
Returns a decimal string giving *window*'s height in pixels. When a window is first created its height will be 1 pixel; the height will eventually be changed by a geometry manager to fulfill the window's needs. If you need the true height immediately after creating a widget, invoke **update** to force the geometry manager to arrange it, or use **winfo :reqheight** to get the window's requested height instead of its actual height.

winfo :id *window*
Returns a hexadecimal string indicating the X identifier for *window*.

winfo :interps
Returns a list whose members are the names of all Tcl interpreters (e.g. all Tk-based applications) currently registered for the display of the invoking application.

winfo :ismapped *window*
Returns **1** if *window* is currently mapped, **0** otherwise.

winfo :name *window*
Returns *window*'s name (i.e. its name within its parent, as opposed to its full path name). The command **winfo :name .** will return the name of the application.

winfo :parent *window*
Returns the path name of *window*'s parent, or an empty string if *window* is the main window of the application.

winfo :pathname *id*
: Returns the path name of the window whose X identifier is *id*. *Id* must be a decimal, hexadecimal, or octal integer and must correspond to a window in the invoking application.

winfo :pixels *window number*
: Returns the number of pixels in *window* corresponding to the distance given by *number*. *Number* may be specified in any of the forms acceptable to **Tk_GetPixels**, such as "2.0c" or "1i". The result is rounded to the nearest integer value; for a fractional result, use **winfo :fpixels**.

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

winfo :reqwidth *window*
: Returns a decimal string giving *window*'s requested width, in pixels. This is the value used by *window*'s geometry manager to compute its geometry.

winfo :rgb *window color*
: Returns a list containing three decimal values, which are the red, green, and blue intensities that correspond to *color* in the window given by *window*. *Color* may be specified in any of the forms acceptable for a color option.

winfo :rootx *window*
: Returns a decimal string giving the x-coordinate, in the root window of the screen, of the upper-left corner of *window*'s border (or *window* if it has no border).

winfo :rooty *window*
: Returns a decimal string giving the y-coordinate, in the root window of the screen, of the upper-left corner of *window*'s border (or *window* if it has no border).

winfo :screen *window*
: Returns the name of the screen associated with *window*, in the form *displayName.screenIndex*.

winfo :screencells *window*
: Returns a decimal string giving the number of cells in the default color map for *window*'s screen.

winfo :screendepth *window*
: Returns a decimal string giving the depth of the root window of *window*'s screen (number of bits per pixel).

winfo :screenheight *window*
: Returns a decimal string giving the height of *window*'s screen, in pixels.

winfo :screenmmheight *window*
: Returns a decimal string giving the height of *window*'s screen, in millimeters.

winfo :screenmmwidth *window*
: Returns a decimal string giving the width of *window*'s screen, in millimeters.

winfo :screenvisual *window*
Returns one of the following strings to indicate the default visual type for *window*'s screen: **directcolor**, **grayscale**, **pseudocolor**, **staticcolor**, **staticgray**, or **truecolor**.

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

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

winfo :visual *window*
Returns one of the following strings to indicate the visual type for *window*: **directcolor**, **grayscale**, **pseudocolor**, **staticcolor**, **staticgray**, or **truecolor**.

winfo :vrootheight *window*
Returns the height of the virtual root window associated with *window* if there is one; otherwise returns the height of *window*'s screen.

winfo :vrootwidth *window*
Returns the width of the virtual root window associated with *window* if there is one; otherwise returns the width of *window*'s screen.

winfo :vrootx *window*
Returns the x-offset of the virtual root window associated with *window*, relative to the root window of its screen. This is normally either zero or negative. Returns 0 if there is no virtual root window for *window*.

winfo :vrooty *window*
Returns the y-offset of the virtual root window associated with *window*, relative to the root window of its screen. This is normally either zero or negative. Returns 0 if there is no virtual root window for *window*.

winfo :width *window*
Returns a decimal string giving *window*'s width in pixels. When a window is first created its width will be 1 pixel; the width will eventually be changed by a geometry manager to fulfill the window's needs. If you need the true width immediately after creating a widget, invoke **update** to force the geometry manager to arrange it, or use **winfo :reqwidth** to get the window's requested width instead of its actual width.

winfo :x *window*
Returns a decimal string giving the x-coordinate, in *window*'s parent, of the upper-left corner of *window*'s border (or *window* if it has no border).

winfo :y *window*
Returns a decimal string giving the y-coordinate, in *window*'s parent, of the upper-left corner of *window*'s border (or *window* if it has no border).

Keywords

atom, children, class, geometry, height, identifier, information, interpreters, mapped, parent, path name, screen, virtual root, width, window

3.25 wm

wm \- Communicate with window manager

Synopsis

wm *option window ?args?*

Description

The **wm** command is used to interact with window managers in order to control such things as the title for a window, its geometry, or the increments in terms of which it may be resized. The **wm** command can take any of a number of different forms, depending on the *option* argument. All of the forms expect at least one additional argument, *window*, which must be the path name of a top-level window.

The legal forms for the **wm** command are:

wm :aspect *window ?minNumer minDenom maxNumer maxDenom?*

If *minNumer*, *minDenom*, *maxNumer*, and *maxDenom* are all specified, then they will be passed to the window manager and the window manager should use them to enforce a range of acceptable aspect ratios for *window*. The aspect ratio of *window* (width/length) will be constrained to lie between *minNumer/minDenom* and *maxNumer/maxDenom*. If *minNumer* etc. are all specified as empty strings, then any existing aspect ratio restrictions are removed. If *minNumer* etc. are specified, then the command returns an empty string. Otherwise, it returns a Tcl list containing four elements, which are the current values of *minNumer*, *minDenom*, *maxNumer*, and *maxDenom* (if no aspect restrictions are in effect, then an empty string is returned).

wm :client *window ?name?*

If *name* is specified, this command stores *name* (which should be the name of the host on which the application is executing) in *window*'s **WM_CLIENT_MACHINE** property for use by the window manager or session manager. The command returns an empty string in this case. If *name* isn't specified, the command returns the last name set in a **wm :client** command for *window*. If *name* is specified as an empty string, the command deletes the **WM_CLIENT_MACHINE** property from *window*.

wm :command *window ?value?*

If *value* is specified, this command stores *value* in *window*'s **WM_COMMAND** property for use by the window manager or session manager and returns an empty string. *Value* must have proper list structure; the elements should contain the words of the command used to invoke the application. If *value* isn't specified then the command returns the last value set in a **wm :command** command for *window*. If *value* is specified as an empty string, the command deletes the **WM_COMMAND** property from *window*.

wm :deiconify *window*

Arrange for *window* to be displayed in normal (non-iconified) form. This is done by mapping the window. If the window has never been mapped then this

command will not map the window, but it will ensure that when the window is first mapped it will be displayed in de-iconified form. Returns an empty string.

wm :focusmodel *window* ?**active**|**passive**?

If **active** or **passive** is supplied as an optional argument to the command, then it specifies the focus model for *window*. In this case the command returns an empty string. If no additional argument is supplied, then the command returns the current focus model for *window*. An **active** focus model means that *window* will claim the input focus for itself or its descendants, even at times when the focus is currently in some other application. **Passive** means that *window* will never claim the focus for itself: the window manager should give the focus to *window* at appropriate times. However, once the focus has been given to *window* or one of its descendants, the application may re-assign the focus among *window*'s descendants. The focus model defaults to **passive**, and Tk's **focus** command assumes a passive model of focussing.

wm :frame *window*

If *window* has been reparented by the window manager into a decorative frame, the command returns the X window identifier for the outermost frame that contains *window* (the window whose parent is the root or virtual root). If *window* hasn't been reparented by the window manager then the command returns the X window identifier for *window*.

wm :geometry *window* ?*newGeometry*?

If *newGeometry* is specified, then the geometry of *window* is changed and an empty string is returned. Otherwise the current geometry for *window* is returned (this is the most recent geometry specified either by manual resizing or in a **wm :geometry** command). *NewGeometry* has the form =*width***x***height*\(+-*x*\(+-*y*, where any of =, *width***x***height*, or \(+-*x*\(+-*y* may be omitted. *Width* and *height* are positive integers specifying the desired dimensions of *window*. If *window* is gridded (see GRIDDED GEOMETRY MANAGEMENT below) then the dimensions are specified in grid units; otherwise they are specified in pixel units. *X* and *y* specify the desired location of *window* on the screen, in pixels. If *x* is preceded by +, it specifies the number of pixels between the left edge of the screen and the left edge of *window*'s border; if preceded by - then *x* specifies the number of pixels between the right edge of the screen and the right edge of *window*'s border. If *y* is preceded by + then it specifies the number of pixels between the top of the screen and the top of *window*'s border; if *y* is preceded by - then it specifies the number of pixels between the bottom of *window*'s border and the bottom of the screen. If *newGeometry* is specified as an empty string then any existing user-specified geometry for *window* is cancelled, and the window will revert to the size requested internally by its widgets.

wm :grid *window* ?*baseWidth baseHeight widthInc heightInc*?

This command indicates that *window* is to be managed as a gridded window. It also specifies the relationship between grid units and pixel units. *BaseWidth* and *baseHeight* specify the number of grid units corresponding to the pixel dimensions requested internally by *window* using **Tk_GeometryRequest**. *WidthInc* and *heightInc* specify the number of pixels in each horizontal and

vertical grid unit. These four values determine a range of acceptable sizes for *window*, corresponding to grid-based widths and heights that are non-negative integers. Tk will pass this information to the window manager; during manual resizing, the window manager will restrict the window's size to one of these acceptable sizes. Furthermore, during manual resizing the window manager will display the window's current size in terms of grid units rather than pixels. If *baseWidth* etc. are all specified as empty strings, then *window* will no longer be managed as a gridded window. If *baseWidth* etc. are specified then the return value is an empty string. Otherwise the return value is a Tcl list containing four elements corresponding to the current *baseWidth*, *baseHeight*, *widthInc*, and *heightInc*; if *window* is not currently gridded, then an empty string is returned. Note: this command should not be needed very often, since the **Tk_SetGrid** library procedure and the **setGrid** option provide easier access to the same functionality.

wm :group *window* ?*pathName*?
If *pathName* is specified, it gives the path name for the leader of a group of related windows. The window manager may use this information, for example, to unmap all of the windows in a group when the group's leader is iconified. *PathName* may be specified as an empty string to remove *window* from any group association. If *pathName* is specified then the command returns an empty string; otherwise it returns the path name of *window*'s current group leader, or an empty string if *window* isn't part of any group.

wm :iconbitmap *window* ?*bitmap*?
If *bitmap* is specified, then it names a bitmap in the standard forms accepted by Tk (see the **Tk_GetBitmap** manual entry for details). This bitmap is passed to the window manager to be displayed in *window*'s icon, and the command returns an empty string. If an empty string is specified for *bitmap*, then any current icon bitmap is cancelled for *window*. If *bitmap* is specified then the command returns an empty string. Otherwise it returns the name of the current icon bitmap associated with *window*, or an empty string if *window* has no icon bitmap.

wm :iconify *window*
Arrange for *window* to be iconified. It *window* hasn't yet been mapped for the first time, this command will arrange for it to appear in the iconified state when it is eventually mapped.

wm :iconmask *window* ?*bitmap*?
If *bitmap* is specified, then it names a bitmap in the standard forms accepted by Tk (see the **Tk_GetBitmap** manual entry for details). This bitmap is passed to the window manager to be used as a mask in conjunction with the **iconbitmap** option: where the mask has zeroes no icon will be displayed; where it has ones, the bits from the icon bitmap will be displayed. If an empty string is specified for *bitmap* then any current icon mask is cancelled for *window* (this is equivalent to specifying a bitmap of all ones). If *bitmap* is specified then the command returns an empty string. Otherwise it returns the name of the current icon mask associated with *window*, or an empty string if no mask is in effect.

wm :iconname *window ?newName?*

If *newName* is specified, then it is passed to the window manager; the window manager should display *newName* inside the icon associated with *window*. In this case an empty string is returned as result. If *newName* isn't specified then the command returns the current icon name for *window*, or an empty string if no icon name has been specified (in this case the window manager will normally display the window's title, as specified with the **wm :title** command).

wm :iconposition *window ?x y?*

If *x* and *y* are specified, they are passed to the window manager as a hint about where to position the icon for *window*. In this case an empty string is returned. If *x* and *y* are specified as empty strings then any existing icon position hint is cancelled. If neither *x* nor *y* is specified, then the command returns a Tcl list containing two values, which are the current icon position hints (if no hints are in effect then an empty string is returned).

wm :iconwindow *window ?pathName?*

If *pathName* is specified, it is the path name for a window to use as icon for *window*: when *window* is iconified then *pathName* should be mapped to serve as icon, and when *window* is de-iconified then *pathName* will be unmapped again. If *pathName* is specified as an empty string then any existing icon window association for *window* will be cancelled. If the *pathName* argument is specified then an empty string is returned. Otherwise the command returns the path name of the current icon window for *window*, or an empty string if there is no icon window currently specified for *window*. Note: not all window managers support the notion of an icon window.

wm :maxsize *window ?width height?*

If *width* and *height* are specified, then *window* becomes resizable and *width* and *height* give its maximum permissible dimensions. For gridded windows the dimensions are specified in grid units; otherwise they are specified in pixel units. During manual sizing, the window manager should restrict the window's dimensions to be less than or equal to *width* and *height*. If *width* and *height* are specified as empty strings, then the maximum size option is cancelled for *window*. If *width* and *height* are specified, then the command returns an empty string. Otherwise it returns a Tcl list with two elements, which are the maximum width and height currently in effect; if no maximum dimensions are in effect for *window* then an empty string is returned. See the sections on geometry management below for more information.

wm :minsize *window ?width height?*

If *width* and *height* are specified, then *window* becomes resizable and *width* and *height* give its minimum permissible dimensions. For gridded windows the dimensions are specified in grid units; otherwise they are specified in pixel units. During manual sizing, the window manager should restrict the window's dimensions to be greater than or equal to *width* and *height*. If *width* and *height* are specified as empty strings, then the minimum size option is cancelled for *window*. If *width* and *height* are specified, then the command returns an empty string. Otherwise it returns a Tcl list with two elements, which are

the minimum width and height currently in effect; if no minimum dimensions are in effect for *window* then an empty string is returned. See the sections on geometry management below for more information.

wm :overrideredirect *window* ?*boolean*?
If *boolean* is specified, it must have a proper boolean form and the override-redirect flag for *window* is set to that value. If *boolean* is not specified then **1** or **0** is returned to indicate whether or not the override-redirect flag is currently set for *window*. Setting the override-redirect flag for a window causes it to be ignored by the window manager; among other things, this means that the window will not be reparented from the root window into a decorative frame and the user will not be able to manipulate the window using the normal window manager mechanisms.

wm :positionfrom *window* ?*who*?
If *who* is specified, it must be either **program** or **user**, or an abbreviation of one of these two. It indicates whether *window*'s current position was requested by the program or by the user. Many window managers ignore program-requested initial positions and ask the user to manually position the window; if **user** is specified then the window manager should position the window at the given place without asking the user for assistance. If *who* is specified as an empty string, then the current position source is cancelled. If *who* is specified, then the command returns an empty string. Otherwise it returns **user** or **window** to indicate the source of the window's current position, or an empty string if no source has been specified yet. Most window managers interpret "no source" as equivalent to **program**. Tk will automatically set the position source to **user** when a **wm :geometry** command is invoked, unless the source has been set explicitly to **program**.

wm :protocol *window* ?*name*? ?*command*?
This command is used to manage window manager protocols such as **WM_DELETE_WINDOW**. *Name* is the name of an atom corresponding to a window manager protocol, such as **WM_DELETE_WINDOW** or **WM_SAVE_YOURSELF** or **WM_TAKE_FOCUS**. If both *name* and *command* are specified, then *command* is associated with the protocol specified by *name*. *Name* will be added to *window*'s **WM_PROTOCOLS** property to tell the window manager that the application has a protocol handler for *name*, and *command* will be invoked in the future whenever the window manager sends a message to the client for that protocol. In this case the command returns an empty string. If *name* is specified but *command* isn't, then the current command for *name* is returned, or an empty string if there is no handler defined for *name*. If *command* is specified as an empty string then the current handler for *name* is deleted and it is removed from the **WM_PROTOCOLS** property on *window*; an empty string is returned. Lastly, if neither *name* nor *command* is specified, the command returns a list of all the protocols for which handlers are currently defined for *window*.

Tk always defines a protocol handler for **WM_DELETE_WINDOW**, even if you haven't asked for one with **wm :protocol**. If a **WM_DELETE_WINDOW** message arrives when you

haven't defined a handler, then Tk handles the message by destroying the window for which it was received. .RE

wm :sizefrom *window ?who?*

If *who* is specified, it must be either **program** or **user**, or an abbreviation of one of these two. It indicates whether *window*'s current size was requested by the program or by the user. Some window managers ignore program-requested sizes and ask the user to manually size the window; if **user** is specified then the window manager should give the window its specified size without asking the user for assistance. If *who* is specified as an empty string, then the current size source is cancelled. If *who* is specified, then the command returns an empty string. Otherwise it returns **user** or **window** to indicate the source of the window's current size, or an empty string if no source has been specified yet. Most window managers interpret "no source" as equivalent to **program**.

wm :state *window*

Returns the current state of *window*: either **normal**, **iconic**, or **withdrawn**.

wm :title *window ?string?*

If *string* is specified, then it will be passed to the window manager for use as the title for *window* (the window manager should display this string in *window*'s title bar). In this case the command returns an empty string. If *string* isn't specified then the command returns the current title for the *window*. The title for a window defaults to its name.

wm :transient *window ?master?*

If *master* is specified, then the window manager is informed that *window* is a transient window (e.g. pull-down menu) working on behalf of *master* (where *master* is the path name for a top-level window). Some window managers will use this information to manage *window* specially. If *master* is specified as an empty string then *window* is marked as not being a transient window any more. If *master* is specified, then the command returns an empty string. Otherwise the command returns the path name of *window*'s current master, or an empty string if *window* isn't currently a transient window.

wm :withdraw *window*

Arranges for *window* to be withdrawn from the screen. This causes the window to be unmapped and forgotten about by the window manager. If the window has never been mapped, then this command causes the window to be mapped in the withdrawn state. Not all window managers appear to know how to handle windows that are mapped in the withdrawn state. Note: it sometimes seems to be necessary to withdraw a window and then re-map it (e.g. with **wm :deiconify**) to get some window managers to pay attention to changes in window attributes such as group.

"Sources Of Geometry Information"

Size-related information for top-level windows can come from three sources. First, geometry requests come from the widgets that are descendants of a top-level window. Each

widget requests a particular size for itself by calling **Tk_GeometryRequest**. This information is passed to geometry managers, which then request large enough sizes for parent windows so that they can layout the children properly. Geometry information passes upwards through the window hierarchy until eventually a particular size is requested for each top-level window. These requests are called *internal requests* in the discussion below. The second source of width and height information is through the **wm :geometry** command. Third, the user can request a particular size for a window using the interactive facilities of the window manager. The second and third types of geometry requests are called *external requests* in the discussion below; Tk treats these two kinds of requests identically.

"Ungridded Geometry Management"

Tk allows the geometry of a top-level window to be managed in either of two general ways: ungridded or gridded. The ungridded form occurs if no **wm :grid** command has been issued for a top-level window. Ungridded management has several variants. In the simplest variant of ungridded windows, no **wm :geometry**, **wm :minsize**, or **wm :maxsize** commands have been invoked either. In this case, the window's size is determined totally by the internal requests emanating from the widgets inside the window: Tk will ask the window manager not to permit the user to resize the window interactively.

If a **wm :geometry** command is invoked on an ungridded window, then the size in that command overrides any size requested by the window's widgets; from now on, the window's size will be determined entirely by the most recent information from **wm :geometry** commands. To go back to using the size requested by the window's widgets, issue a **wm :geometry** command with an empty *geometry* string.

To enable interactive resizing of an ungridded window, one or both of the **wm :maxsize** and **wm :minsize** commands must be issued. The information from these commands will be passed to the window manager, and size changes within the specified range will be permitted. For ungridded windows the limits refer to the top-level window's dimensions in pixels. If only a **wm :maxsize** command is issued then the minimum dimensions default to 1; if only a **wm :minsize** command is issued then the maximum dimensions default to the size of the display. If the size of a window is changed interactively, it has the same effect as if **wm :geometry** had been invoked: from now on, internal geometry requests will be ignored. To return to internal control over the window's size, issue a **wm :geometry** command with an empty *geometry* argument. If a window has been manually resized or moved, the **wm :geometry** command will return the geometry that was requested interactively.

"Gridded Geometry Management"

The second style of geometry management is called *gridded*. This approach occurs when one of the widgets of an application supports a range of useful sizes. This occurs, for example, in a text editor where the scrollbars, menus, and other adornments are fixed in size but the edit widget can support any number of lines of text or characters per line. In this case, it is usually desirable to let the user specify the number of lines or characters-per-line, either with the **wm :geometry** command or by interactively resizing the window. In the case of text, and in other interesting cases also, only discrete sizes of the window make sense, such as integral numbers of lines and characters-per-line; arbitrary pixel sizes are not useful.

Gridded geometry management provides support for this kind of application. Tk (and the window manager) assume that there is a grid of some sort within the application and that the application should be resized in terms of *grid units* rather than pixels. Gridded geometry management is typically invoked by turning on the **setGrid** option for a widget; it can also be invoked with the **wm :grid** command or by calling **Tk_SetGrid**. In each of these approaches the particular widget (or sometimes code in the application as a whole) specifies the relationship between integral grid sizes for the window and pixel sizes. To return to non-gridded geometry management, invoke **wm :grid** with empty argument strings.

When gridded geometry management is enabled then all the dimensions specified in **wm :minsize**, **wm :maxsize**, and **wm :geometry** commands are treated as grid units rather than pixel units. Interactive resizing is automatically enabled, and it will be carried out in even numbers of grid units rather than pixels. By default there are no limits on the minimum or maximum dimensions of a gridded window. As with ungridded windows, interactive resizing has exactly the same effect as invoking the **wm :geometry** command. For gridded windows, internally- and externally-requested dimensions work together: the externally-specified width and height determine the size of the window in grid units, and the information from the last **wm :grid** command maps from grid units to pixel units.

Bugs

The window manager interactions seem too complicated, especially for managing geometry. Suggestions on how to simplify this would be greatly appreciated.

Most existing window managers appear to have bugs that affect the operation of the **wm** command. For example, some changes won't take effect if the window is already active: the window will have to be withdrawn and de-iconified in order to make the change happen.

Keywords

aspect ratio, deiconify, focus model, geometry, grid, group, icon, iconify, increments, position, size, title, top-level window, units, window manager

Short Contents

Table of Contents