Perl Programmers Reference Guide

Tk, 800.022
07–Feb–2001

"There's more than one way to do it."
–– Larry Wall, Author of the Perl Programming Language

Author: ni–s

blank

NAME

Tk – a graphical user interface toolkit for Perl

SYNOPSIS

```
use Tk;
$top = new MainWindow;
MainLoop;
```

DESCRIPTION

The Perl/Tk manual is split up into a number of sections:

Introduction

- *Tk::overview|Tk::overview*
- *Tk::UserGuide|Tk::UserGuide*

Tk Geometry Management

- *Tk::Adjuster|Tk::Adjuster*
- *Tk::form|Tk::form*
- *Tk::grid|Tk::grid*
- *Tk::pack|Tk::pack*
- *Tk::place|Tk::place*
- *Tk::Table|Tk::Table*
- *Tk::Tiler|Tk::Tiler*
- *Tk::Wm|Tk::Wm*

Binding Events and Callbacks

- *Tk::After|Tk::After*
- *Tk::bind|Tk::bind*
- *Tk::bindtags|Tk::bindtags*
- *Tk::callbacks|Tk::callbacks*
- *Tk::Error|Tk::Error*
- *Tk::event|Tk::event*
- *Tk::exit|Tk::exit*
- *Tk::fileevent|Tk::fileevent*
- *Tk::IO|Tk::IO*

Tk Image Classes

- *Tk::Animation|Tk::Animation*
- *Tk::Bitmap|Tk::Bitmap*
- *Tk::Compound|Tk::Compound*
- *Tk::Image|Tk::Image*
- *Tk::Photo|Tk::Photo*

- *Tk::Pixmap|Tk::Pixmap*

Tix Extensions

- *Tk::Balloon|Tk::Balloon*
- *Tk::BrowseEntry|Tk::BrowseEntry*
- *Tk::DialogBox|Tk::DialogBox*
- *Tk::DirTree|Tk::DirTree*
- *Tk::DItem|Tk::DItem*
- *Tk::InputO|Tk::InputO*
- *Tk::LabFrame|Tk::LabFrame*
- *Tk::Mwm|Tk::Mwm*
- *Tk::NoteBook|Tk::NoteBook*
- *Tk::TixGrid|Tk::TixGrid*
- *Tk::tixWm|Tk::tixWm*
- *Tk::TList|Tk::TList*
- *Tk::Tree|Tk::Tree*

Tk Widget Classes

- *Tk::Button|Tk::Button*
- *Tk::Canvas|Tk::Canvas*
- *Tk::Checkbutton|Tk::Checkbutton*
- *Tk::Entry|Tk::Entry*
- *Tk::Frame|Tk::Frame*
- *Tk::HList|Tk::HList*
- *Tk::Label|Tk::Label*
- *Tk::Listbox|Tk::Listbox*
- *Tk::Menu|Tk::Menu*
- *Tk::Menubutton|Tk::Menubutton*
- *Tk::Message|Tk::Message*
- *Tk::Optionmenu|Tk::Optionmenu*
- *Tk::Radiobutton|Tk::Radiobutton*
- *Tk::Scale|Tk::Scale*
- *Tk::Scrollbar|Tk::Scrollbar*
- *Tk::Text|Tk::Text*
- *Tk::Toplevel|Tk::Toplevel*

Tk Generic Methods

- *Tk::Font|Tk::Font*

- *Tk::send|Tk::send*
- *Tk::tkvars|Tk::tkvars*
- *Tk::Widget|Tk::Widget*
- *Tk::X11Font|Tk::X11Font*

User Interaction

- *Tk::DropSite|Tk::DropSite*
- *Tk::Clipboard|Tk::Clipboard*
- *Tk::focus|Tk::focus*
- *Tk::grab|Tk::grab*
- *Tk::selection|Tk::selection*

Creating and Configuring Widgets

- *Tk::CmdLine|Tk::CmdLine*
- *Tk::MainWindow|Tk::MainWindow*
- *Tk::option|Tk::option*
- *Tk::options|Tk::options*
- *Tk::palette|Tk::palette*
- *Tk::Xrm|Tk::Xrm*

Popups and Dialogs

- *Tk::chooseColor|Tk::chooseColor*
- *Tk::ColorEditor|Tk::ColorEditor*
- *Tk::Dialog|Tk::Dialog*
- *Tk::Dialog|Tk::Dialog*
- *Tk::FileSelect|Tk::FileSelect*
- *Tk::getOpenFile|Tk::getOpenFile*
- *Tk::messageBox|Tk::messageBox*

Derived Widgets

- *Tk::composite|Tk::composite*
- *Tk::configspec|Tk::configspec*
- *Tk::Derived|Tk::Derived*
- *Tk::mega|Tk::mega*
- *Tk::ROText|Tk::ROText*
- *Tk::Scrolled|Tk::Scrolled*
- *Tk::TextUndo|Tk::TextUndo*
- *Tk::Reindex|Tk::Reindex*
- *Tk::Pane|Tk::Pane*

- *Tk::ProgressBar\Tk::ProgressBar*

C Programming

- Internals
- pTk
- 3DBorder
- BackgdErr
- BindTable
- CanvPsY
- CanvTkwin
- CanvTxtInfo
- Clipboard
- ClrSelect
- ConfigWidg
- ConfigWind
- CoordToWin
- CrtErrHdlr
- CrtGenHdlr
- CrtImgType
- CrtItemType
- CrtMainWin
- CrtPhImgFmt
- CrtSelHdlr
- CrtWindow
- DeleteImg
- DoOneEvent
- DoWhenIdle
- DrawFocHlt
- EventHndlr
- EventInit
- FileHndlr
- FindPhoto
- FontId
- FreeXId
- GeomReq
- GetAnchor

- GetBitmap
- GetCapStyl
- GetClrmap
- GetColor
- GetCursor
- GetFont
- GetFontStr
- GetGC
- GetImage
- GetJoinStl
- GetJustify
- GetOption
- GetPixels
- GetPixmap
- GetRelief
- GetRootCrd
- GetScroll
- GetSelect
- GetUid
- GetVisual
- GetVRoot
- HandleEvent
- IdToWindow
- ImgChanged
- InternAtom
- MainLoop
- MaintGeom
- MainWin
- ManageGeom
- MapWindow
- MeasureChar
- MoveToplev
- Name
- NameOfImg
- OwnSelect

- ParseArgv
- Preserve
- QWinEvent
- Restack
- RestrictEv
- SetAppName
- SetClass
- SetGrid
- SetVisual
- Sleep
- StrictMotif
- TextLayout
- TimerHndlr
- Tk_Init
- WindowId

Implementation

- *Tk::Eventloop|Tk::Eventloop*
- *Tk::Item|Tk::Item*
- *Tk::Submethods|Tk::Submethods*
- *Tk::WidgetDemo|Tk::WidgetDemo*
- *Tk::widgets|Tk::widgets*

Experimental Modules

- *Tk::Common|Tk::Common*
- *Tk::SunConst|Tk::SunConst*
- *Tk::WinPhoto|Tk::WinPhoto*

Other Modules and Languages

- *Tk::Compile|Tk::Compile*
- *Tk::Tcl−perl|Tk::Tcl−perl*
- *Tk::X|Tk::X*

AUTHOR

Nick Ing−Simmons

SEE ALSO

perl(1)|perl, *wish(1)|wish*.

NAME

Tk::Adjuster − Allow size of packed widgets to be adjusted by user

=for pm Tk/Adjuster.pm

=for category Tk Geometry Management

SYNOPSIS

use Tk::Adjuster;

$adjuster = *$widget*−**Adjuster***(?options?)*;

WIDGET−SPECIFIC OPTIONS

Name: **restore**
Class: **Restore**
Switch: **−restore**

Specifies a boolean value that determines whether the Adjuster should forcibly attempt to make room for itself (by reducing the size of its managed widget) when it is unmapped (for example, due to a size change in a top level window). The default value is 1.

Name: **side**
Class: **Side**
Switch: **−side**

Specifies the side on which the managed widget lies relative to the Adjuster. In conjunction with the pack geometry manager, this relates to the side of the master against which the managed widget and the Adjuster are packed. Must be **left**, **right**, **top**, or **bottom**. Defaults to **top**.

Name: **widget**
Class: **Widget**
Switch: **−widget**

Specifies the widget which is to be managed by the Adjuster.

DESCRIPTION

Tk::Adjuster is a Frame containing a "line" and a "blob".

Dragging with Mouse Button−1 results in a line being dragged to indicate new size. Releasing Button−1 submits GeometryRequests on behalf of the managed widget which will cause the packer to change the widget‘s size.

If Drag is done with Shift button down, then GeometryRequests are made in "real time" so that text−flow effects can be seen, but as a lot more work is done behaviour may be sluggish.

If widget is packed with −side = left or −side = right then width is adjusted. If packed −side = top or −side = bottom then height is adjusted.

packPropagate is turned off for the master window to prevent adjustment changing overall window size. Similarly **packPropagate** is turned off for the managed widget if it has things packed inside it. This is so that the GeometryRequests made by **Tk::Adjuster** are not overridden by pack.

In addition, the managed widget is made non−expandable to prevent the geometry manager reallocating freed space in the master back to the managed widget. Note however that expansion is turned off only after the Adjuster is mapped, which allows the managed widget to expand naturally on window creation.

The Tk::Widget method, **packAdjust**, calls pack on the widget, then creates an instance of **Tk::Adjuster**, and packs that "after" the widget. Its use has two disadvantages however: the Adjuster widget is not made available to the caller, and options cannot be set on the Adjuster. For these reasons, the Tk::Adjuster method, **packAfter** is preferred, but **packAdjust** is retained for backwards compatibility.

WIDGET METHODS

$adjuster–**packAfter**(*managed_widget,* ?pack_options?)

This command configures the Adjuster's **–widget** and **–side** options respectively to *managed_widget* and the **–side** value specified in *pack_options* (**top** if not specified). It then packs the Adjuster after *managed_widget*, with **–fill** set to **x** or **y** as appropriate.

$adjuster–**packForget***?(boolean)?*

This command calls **Tk::Widget::packForget** on the Adjuster. If a parameter is provided and it has a true boolean value, then **packForget** is also called on the managed widget.

$adjuster–**slave**

This command returns the value *$adjuster–cget('–widget')*, ie. the reference to the managed widget.

EXAMPLES

Using an Adjuster to separate two widgets, whereby the left widget is managed, and right widget expands to fill space on a window resize

a) Using packAfter (preferred interface)

```
use Tk;
use Tk::Adjuster;

my $f = MainWindow->new;
my $lst1 = $f->Listbox();
my $adj1 = $f->Adjuster();
my $lst2 = $f->Listbox();

my $side = 'left';
$lst1->pack(-side => $side, -fill => 'both', -expand => 1);
$adj1->packAfter($lst1, -side => $side);
$lst2->pack(-side => $side, -fill => 'both', -expand => 1);
MainLoop;
```

b) Using packAdjust

```
use Tk;
use Tk::Adjuster;

my $f = MainWindow->new;
my $lst1 = $f->Listbox();
my $lst2 = $f->Listbox();

my $side = 'left';
$lst1->packAdjust(-side => $side, -fill => 'both');
$lst2->pack      (-side => $side, -fill => 'both', -expand => 1);
MainLoop;
```

c) Using the standard Tk::Widget::pack

```
use Tk;
use Tk::Adjuster;

my $f = MainWindow->new;
my $side = 'left';
my $lst1 = $f->Listbox();
my $adj  = $f->Adjuster(-widget => $lst1, -side => $side);
my $lst2 = $f->Listbox();

$lst1->pack(-side => $side, -fill => 'both', -expand => 1);
```

```
$adj->pack (-side => $side, -fill => 'y');
$lst2->pack(-side => $side, -fill => 'both', -expand => 1);

MainLoop;
```

Changing the above examples so that `$side` has the value 'right' means the left widget expands to fill space on a window resize.

Changing the above examples so that `$side` has the value 'top' produces a testcase with a horizontal Adjuster. Here the bottom widget expands to fill space on a window resize. Packing to the 'bottom' makes the top widget expand to fill space on window resize.

Using –restore => 0 for multiple columns

In the case of multiple columns (or rows) the "restore" functionality of the Adjuster can be inconvenient. When the user adjusts the width of one column and thereby pushes the Adjuster of another column off the window, this adjuster tries to restore itself by reducing the size of its managed widget. This has the effect that column widths shrink; and the original size is not restored when the user reverses the originating change. The **–restore** option can be used to turn off this functionality. (It makes some sense, however, to leave **–restore** turned on for the first–packed Adjuster, so that at least one Adjuster always remains visible.)

```
use Tk;
use Tk::Adjuster;
my $f = MainWindow->new;
my $lst1 = $f->Listbox();
my $adj1 = $f->Adjuster();
my $lst2 = $f->Listbox();
my $adj2 = $f->Adjuster(-restore => 0);
my $lst3 = $f->Listbox();

my $side = 'left';
$lst1->pack(-side => $side, -fill => 'both', -expand => 1);
$adj1->packAfter($lst1, -side => $side);
$lst2->pack(-side => $side, -fill => 'both', -expand => 1);
$adj2->packAfter($lst2, -side => $side);
$lst3->pack(-side => $side, -fill => 'both', -expand => 1);

MainLoop;
```

BUGS

It is currently not possible to configure the appearance of the Adjuster. It would be nice to be able to set the width and relief of the Adjuster "line" and the presence/absence of the "blob" on the Adjuster.

Tk::Adjuster works theoretically with the grid geometry manager but there are currently some problems which seem to be due to bugs in grid:

a) There's never an Unmap event for the adjuster, so the "restore" functionality has no effect.
b) After adjusting, widgets protrude into the border of the master.
c) grid('Propagate', 0) on MainWindow has no effect – window shrinks/grows when widgets are adjusted.
d) Widgets shuffle to correct position on startup

NAME

Tk::after – Execute a command after a time delay

=for category Binding Events and Callbacks

SYNOPSIS

$widget->**after**(*ms*)

$id = *$widget*->**after**(*ms?,callback?*)

$id = *$widget*->**repeat**(*ms?,callback?*)

$widget*->**afterCancel**(*$id)

$id = *$widget*->**afterIdle**(*callback*)

$widget->**afterInfo**?(*$id*)?

DESCRIPTION

This method is used to delay execution of the program or to execute a callback in background sometime in the future.

In perl/Tk *$widget*->**after** is implemented via the class `Tk::After`, and callbacks are associated with *$widget*, and are automatically cancelled when the widget is destroyed. An almost identical interface, but without automatic cancel, and without repeat is provided via Tk::after method.

The internal Tk::After class has the following synopsis:

```
$id = Tk::After->new($widget,$time,'once',callback);
$id = Tk::After->new($widget,$time,'repeat',callback);
$id->cancel;
```

The **after** method has several forms as follows:

$widget->**after**(*ms*)

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

$widget->**after**(*ms,callback*)

In this form the command returns immediately, but it arranges for *callback* be executed *ms* milliseconds later as an event handler. The callback will be executed exactly once, at the given time. The command will be executed in context of *$widget*. If an error occurs while executing the delayed command then the *Tk::Error*\|*Tk::Error* mechanism is used to report the error. The **after** command returns an identifier (an object in the perl/Tk case) that can be used to cancel the delayed command using **afterCancel**.

$widget->**repeat**(*ms,callback*)

In this form the command returns immediately, but it arranges for *callback* be executed *ms* milliseconds later as an event handler. After *callback* has executed it is re–scheduled, to be executed in a futher *ms*, and so on until it is cancelled.

$widget*->**afterCancel**(*$id)
$id->*cancel*

Cancels the execution of a delayed command that was previously scheduled. *$id* indicates which command should be canceled; it must have been the return value from a previous **after** command. If the command given by *$id* has already been executed (and is not scheduled to be executed again) then **afterCancel** has no effect.

$widget->**afterCancel**(*callback*)

This form is not robust in perl/Tk − its use is deprecated. This command should also cancel the execution of a delayed command. The *callback* argument is compared with pending callbacks, if a match is found, that callback is cancelled and will never be executed; if no such callback is currently pending then the **afterCancel** has no effect.

$widget->**afterIdle**(*callback*)

Arranges for *callback* to be evaluated later as an idle callback. The script will be run exactly once, the next time the event loop is entered and there are no events to process. The command returns an identifier that can be used to cancel the delayed command using **afterCancel**. If an error occurs while executing the script then the *Tk::Error|Tk::Error* mechanism is used to report the error.

$widget->**afterInfo***?($id)?*

This command returns information about existing event handlers. If no *$id* argument is supplied, the command returns a list of the identifiers for all existing event handlers created by the **after** command for this MainWindow. If *$id* is supplied, it specifies an existing handler; *$id* must have been the return value from some previous call to **after** and it must not have triggered yet or been cancelled. In this case the command returns a list with two elements. The first element of the list is the callback associated with *$id*, and the second element is either **idle** or **timer** to indicate what kind of event handler it is.

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

SEE ALSO

Tk::Error|Tk::Error Tk::callbacks|Tk::callbacks

KEYWORDS

cancel, delay, idle callback, sleep, time

NAME

Tk::Animation – Display sequence of Tk::Photo images

=for pm Tk/Animation.pm

=for category Tk Image Classes

SYNOPSIS

```
use Tk::Animation
my $img = $widget->Animation('-format' => 'gif', -file => 'somefile.gif');

$img->start_animation($period);
$img->stop_animation;

$img->add_frames(@images);
```

DESCRIPTION

In the simple case when `Animation` is passed a GIF89 style GIF with multiple 'frames', it will build an internal array of `Photo` images.

`start_animation($period)` then initiates a `repeat` with specified *`$period`* to sequence through these images.

`stop_animation` cancels the `repeat` and resets the image to the first image in the sequence.

The `add_frames` method adds images to the sequence. It is provided to allow animations to be constructed from separate images. All images must be `Photo`s and should all be the same size.

BUGS

The 'period' should probably be a property of the Animation object rather than specified at 'start' time. It may even be embedded in the GIF.

NAME

Tk::Balloon – pop up help balloons.

=for pm Tixish/Balloon.pm

=for category Tix Extensions

SYNOPSIS

```
use Tk::Balloon;
...
$b = $top->Balloon(-statusbar => $status_bar_widget);

# Normal Balloon:
$b->attach($widget,
           -balloonmsg => "Balloon help message",
           -statusmsg => "Status bar message");

# Balloon attached to entries in a menu widget:
$b->attach($menu, -state => 'status',
                  -msg => ['first menu entry',
                           'second menu entry',
                           ...
                          ],
          );

# Balloon attached to individual items in a canvas widget:
$b->attach($canvas, -balloonposition => 'mouse',
                    -msg => {'item1' => 'msg1',
                             'tag2'  => 'msg2',
                              ...
                            },
          );
```

DESCRIPTION

Balloon provides the framework to create and attach help balloons to various widgets so that when the mouse pauses over the widget for more than a specified amount of time, a help balloon is popped up.

Balloons and Menus

If the balloon is attached to a **Menu** widget and the message arguments are array references, then each element in the array will be the message corresponding to a menu entry. The balloon message will then be shown for the entry which the mouse pauses over. Otherwise it is assumed that the balloon is to be attached to the **Menu** as a whole. You can have separate status and balloon messages just like normal balloons.

Balloons and Canvases

If the balloon is attached to a **Canvas** widget and the message arguments are hash references, then each hash key should correspond to a canvas item ID or tag and the associated value will correspond to the message for that canvas item. The balloon message will then be shown for the current item (the one at the position of the mouse). Otherwise it is assumed that the balloon is to be attached to the **Canvas** as a whole. You can have separate status and balloon messages just like normal balloons.

OPTIONS

Balloon accepts all of the options that the **Frame** widget accepts. In addition, the following options are also recognized.

–initwait

Specifies the amount of time to wait without activity before popping up a help balloon. Specified in milliseconds. Defaults to 350 milliseconds. This applies to both the popped up balloon and the status bar message.

–state

Can be one of **'balloon'**, **'status'**, **'both'** or **'none'** indicating that the help balloon, status bar help, both or none respectively should be activated when the mouse pauses over the client widget. Default is **'both'**.

–statusbar

Specifies the widget used to display the status message. This widget should accept the **–text** option and is typically a **Label**. If the widget accepts the **–textvariable** option and that option is defined then it is used instead of the **–text** option.

–balloonposition

Can be one of **'widget'** or **'mouse'**. It controls where the balloon will popup. **'widget'** makes the balloon appear at the lower right corner of the widget it is attached to (default), and **'mouse'** makes the balloon appear below and to the right of the current mouse position.

–postcommand

This option takes a **CODE** reference which will be executed before the balloon and statusbar messages are displayed and should return a true or false value to indicate whether you want the balloon to be displayed or not. This also lets you control where the balloon is positioned by returning a true value that looks like *X,Y* (matches this regular expression: `/^(\d+),(\d+)$/`). If the postcommand returns a value that matches that re then those coordinates will be used as the position to post the balloon. *Warning:* this subroutine should return quickly or the balloon response will appear slow.

–cancelcommand

This option takes a **CODE** reference which will be executed before the balloon and statusbar messages are canceled and should return a true or false value to indicate whether you want the balloon to be canceled or not. *Warning:* this subroutine should return quickly or the balloon response will appear slow.

–motioncommand

This option takes a **CODE** reference which will be executed for any motion event and should return a true or false value to indicate whether the currently displayed balloon should be canceled (deactivated). If it returns true then the balloon will definitely be canceled, if it returns false then it may still be canceled depending the internal rules. *Note:* a new balloon may be posted after the **–initwait** time interval, use the **–postcommand** option to control that behavior. *Warning:* the subroutine should be extremely fast or the balloon response will appear slow and consume a lot of CPU time (it is executed every time the mouse moves over the widgets the balloon is attached to).

METHODS

The **Balloon** widget supports only three non–standard methods:

attach(*widget*, *options*)

Attaches the widget indicated by *widget* to the help system. The allowed options are:

–statusmsg

The argument is the message to be shown on the status bar when the mouse pauses over this client. If this is not specified, but **–msg** is specified then the message displayed on the status bar is the same as the argument for **–msg**. If you give it a scalar reference then it is dereferenced before being displayed. Useful if the postcommand is used to change the message.

–balloonmsg

The argument is the message to be displayed in the balloon that will be popped up when the mouse pauses over this client. As with **–statusmsg** if this is not specified, then it takes its value from the **–msg** specification if any. If neither **–balloonmsg** nor **–msg** are specified, or they are the empty string then no balloon is popped up instead of an empty balloon. If you give it a scalar reference then it is dereferenced before being displayed. Useful if the postcommand is used to change the message.

–msg

The catch–all for **–statusmsg** and **–balloonmsg**. This is a convenient way of specifying the same message to be displayed in both the balloon and the status bar for the client.

–initwait
–state
–statusbar
–balloonposition
–postcommand
–cancelcommand
–motioncommand

These options allow you to override the balloon's default value for those option for some of the widgets it is attached to. It accepts the same values as above and will default to the **Balloon**'s value.

detach(*widget*)

Detaches the specified *widget* from the help system.

destroy

Destroys the specified balloon.

EXAMPLES

See the balloon demo included with the widget demo script that came with the distribution for examples on various ways to use balloons.

NOTES

Because of the overhead associated with each balloon you create (from tracking the mouse movement to know when to activate and deactivate them) you will see the best performance (low CPU consumption) if you create as few balloons as possible and attach them to as many widgets as you can. In other words, don't create a balloon for each widget you want to attach one to.

CAVEATS

Pressing any button will deactivate (cancel) the current balloon, if one exists. You can usually make the balloon reappear by moving the mouse a little. Creative use of the 3 command options can help you out also. If the mouse is over the balloon when a menu is unposted then the balloon will remain until you move off of it.

BUGS

Hopefully none, probably some.

AUTHORS

Rajappa Iyer rsi@earthling.net did the original coding.

Jason A. Smith <smithj4@rpi.edu added support for menus and made some other enhancements.

Slaven Rezic <eserte@cs.tu–berlin.de added support for canvas items.

HISTORY

The code and documentation was derived from Balloon.tcl from the Tix4.0 distribution by Ioi Lam and modified by the above mentioned authors. This code may be redistributed under the same terms as Perl.

NAME

Tk::bind – Arrange for X events to invoke callbacks

=for category Binding Events and Callbacks

SYNOPSIS

Retrieve bindings:

$widget->**bind**

$widget->**bind***(tag)*

$widget->**bind***(sequence)*

$widget->**bind***(tag,sequence)*

Associate and destroy bindings:

$widget->**bind***(sequence,callback)*

$widget->**bind***(tag,sequence,callback)*

DESCRIPTION

The **bind** method associates callbacks with X events. If *callback* is specified, **bind** will arrange for *callback* to be evaluated whenever the event(s) given by *sequence* occur in the window(s) identified by *$widget* or *tag*. If *callback* is an empty string then the current binding for *sequence* is destroyed, leaving *sequence* unbound. In all of the cases where a *callback* argument is provided, **bind** returns an empty string.

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

If no *tag* is specified then the **bind** refers to *$widget*. If *tag* is specified then it is typically a class name and the **bind** refers to all instances of the class on the **MainWindow** associated with *$widget*. (It is possible for *tag* to be another "widget object" but this practice is deprecated.) Perl's **ref**(*$object*) can be used to get the class name of any object. Each window has an associated list of tags, and a binding applies to a particular window if its tag is among those specified for the window. Although the **bindtags** method may be used to assign an arbitrary set of binding tags to a window, the default binding tags provide the following behavior:

If a tag is the name of an internal window the binding applies to that window.

If the tag is the name of a toplevel window the binding applies to the toplevel window and all its internal windows.

If the tag is the name of a class of widgets, such as **Tk::Button**, the binding applies to all widgets in that class;

If *tag* has the value **all**, the binding applies to all windows descended from the MainWindow of the application.

EVENT PATTERNS

The *sequence* argument specifies a sequence of one or more event patterns, with optional white space between the patterns. Each event pattern has the following syntax:

'<modifier–modifier–type–detail'

The entire event pattern is surrounded by angle brackets, and normally needs to be quoted, as angle brackets are special to perl. 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.

A second form of pattern is used to specify a user–defined, named virtual event; see *Tk::event* for details. It has the following syntax:

<<name

The entire virtual event pattern is surrounded by double angle brackets. Inside the angle brackets is the user−defined name of the virtual event. Modifiers, such as **Shift** or **Control**, may not be combined with a virtual event to modify it. Bindings on a virtual event may be created before the virtual event is defined, and if the definition of a virtual event changes dynamically, all windows bound to that virtual event will respond immediately to the new definition.

MODIFIERS

Modifiers consist of any of the following values:

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

Where more than one value is listed, separated by commas, the values are equivalent. Most of the modifiers have the obvious X meanings. For example, **Button1** requires that button 1 be depressed when the event occurs. For a binding to match a given event, the modifiers in the event must include all of those specified in the event pattern. An event may also contain additional modifiers not specified in the binding. For example, if button 1 is pressed while the shift and control keys are down, the pattern **<Control−Button−1>** will match the event, but **<Mod1−Button−1>** will not. If no modifiers are specified, then any combination of modifiers may be present in the event.

Meta and **M** refer to whichever of the **M1** through **M5** modifiers is associated with the meta key(s) on the keyboard (keysyms **Meta_R** and **Meta_L**). If there are no meta keys, or if they are not associated with any modifiers, then **Meta** and **M** will not match any events. Similarly, the **Alt** modifier refers to whichever modifier is associated with the alt key(s) on the keyboard (keysyms **Alt_L** and **Alt_R**).

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

EVENT TYPES

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

```
ButtonPress, Button    Expose  Map
ButtonRelease  FocusIn Motion
Circulate      FocusOut        Property
Colormap       Gravity Reparent
Configure      KeyPress, Key   Unmap
Destroy        KeyRelease      Visibility
Enter  Leave   Activate
Deactivate
```

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

the specifier **<1>** is equivalent to **<ButtonPress−1>**.

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

BINDING CALLBACKS AND SUBSTITUTIONS

The *callback* argument to **bind** is a perl/Tk callback. which will be executed whenever the given event sequence occurs. (See *Tk::callbacks* for description of the possible forms.) *Callback* will be associated with the same **MainWindow** that is associated with the `$widget` that was used to invoke the **bind** method, and it will run as though called from **MainLoop**. If *callback* contains any **Ev**(%) calls, then each "nested" **Ev**(%) "callback" will be evaluated when the event occurs to form arguments to be passed to the main *callback*. The replacement depends on the character %, as defined in the list below. Unless otherwise indicated, the replacement string is the numeric (decimal) value of the given field from the current event. Perl/Tk has enhanced this mechanism slightly compared to the comparable Tcl/Tk mechanism. The enhancements are not yet all reflected in the list below. 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 (not the same as **undef**!).

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

‘a’ The *above* field from the event, formatted as a hexadecimal number. Valid only for **Configure** events.

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

‘c’ The *count* field from the event. Valid only for **Expose** events.

‘d’ The *detail* field from the event. The **‘d’** is replaced by a string identifying the detail. For **Enter**, **Leave**, **FocusIn**, and **FocusOut** events, the string will be one of the following:

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

For events other than these, the substituted string is undefined. (Note that this is *not* the same as Detail part of sequence use to specify the event.)

‘f’ The *focus* field from the event (or **1**). Valid only for **Enter** and **Leave** events.

‘h’ The *height* field from the event. Valid only for **Configure**, **Expose**, and **GraphicsExpose** 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 **Map**, **Reparent**, and **Configure** events.

‘p’ The *place* field from the event, substituted as one of the strings **PlaceOnTop** or **PlaceOnBottom**. Valid only for **Circulate** events.

's' The *state* field from the event. For **ButtonPress**, **ButtonRelease**, **Enter**, **KeyPress**, **KeyRelease**, **Leave**, and **Motion** events, a decimal string is substituted. For **Visibility**, one of the strings **VisibilityUnobscured**, **VisibilityPartiallyObscured**, and **VisibilityFullyObscured** is substituted.

't' The *time* field from the event. Valid only for events that contain a *time* field.

'w' The *width* field from the event. Valid only for **Configure**, **Expose**, and **GraphicsExpose** 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.

'@' The string "@*x,y*" where *x* and *y* are as above. Valid only for events containing *x* and *y* fields. This format is used my methods of **Tk::Text** and similar widgets.

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

'B' The *border_width* field from the event. Valid only for **Configure** events.

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

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

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

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

'S' The *subwindow* window identifier from the event, as an object if it is one otherwise as a hexadecimal number. Valid only for events containing a *subwindow* field.

'T' The *type* field from the event. Valid for all event types.

'W' The window to which the event was reported (the `$widget` field from the event) – as an perl/Tk object. Valid for all event types.

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

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

MULTIPLE MATCHES

It is possible for several bindings to match a given X event. If the bindings are associated with different *tag*'s, then each of the bindings will be executed, in order. By default, a class binding will be executed first, followed by a binding for the widget, a binding for its toplevel, and an **all** binding. The **bindtags** method may be used to change this order for a particular window or to associate additional binding tags with the window.

return and **Tk–>break** may be used inside a callback to control the processing of matching callbacks. If **return** is invoked, then the current callback is terminated but Tk will continue processing callbacks associated with other *tag*'s. If **Tk–>break** is invoked within a callback, then that callback terminates and no other callbacks will be invoked for the event. (**Tk–>break** is implemented via perl's **die** with a special value which is "caught" by the perl/Tk "glue" code.)

If more than one binding matches a particular event and they have the same *tag*, then the most specific binding is chosen and its callback is evaluated. The following tests are applied, in order, to determine which of several matching sequences is more specific: (a) an event pattern that specifies a specific button or key is

more specific than one that doesn't; (b) a longer sequence (in terms of number of events matched) is more specific than a shorter sequence; (c) if the modifiers specified in one pattern are a subset of the modifiers in another pattern, then the pattern with more modifiers is more specific. (d) a virtual event whose physical pattern matches the sequence is less specific than the same physical pattern that is not associated with a virtual event. (e) given a sequence that matches two or more virtual events, one of the virtual events will be chosen, but the order is undefined.

If the matching sequences contain more than one event, then tests (c)–(e) are applied in order from the most recent event to the least recent event in the sequences. If these tests fail to determine a winner, then the most recently registered sequence is the winner.

If there are two (or more) virtual events that are both triggered by the same sequence, and both of those virtual events are bound to the same window tag, then only one of the virtual events will be triggered, and it will be picked at random:

```
$widget->eventAdd('<<Paste>>' => '<Control-y>');
$widget->eventAdd('<<Paste>>' => '<Button-2>');
$widget->eventAdd <<Scroll>>' => '<Button-2>');
$widget->bind('Tk::Entry','<<Paste>>',sub { print 'Paste'});
$widget->bind('Tk::Entry','<<Scroll>>', sub {print 'Scroll'});
```

If the user types Control–y, the **<<Paste>>** binding will be invoked, but if the user presses button 2 then one of either the **<<Paste>>** or the **<<Scroll>>** bindings will be invoked, but exactly which one gets invoked is undefined.

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

MULTI–EVENT SEQUENCES AND IGNORED EVENTS

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

ERRORS

If an error occurs in executing the callback for a binding then the **Tk::Error** mechanism is used to report the error. The **Tk::Error** mechanism will be executed at same call level, and associated with the same **MainWindow** as as the callback was invoked.

CAVEATS

Note that for the **Canvas** widget, the call to **bind** has to be fully qualified. This is because there is already a bind method for the **Canvas** widget, which binds individual canvas tags.

$canvas–>**Tk::bind**

SEE ALSO

Tk::Error\Tk::Error Tk::callbacks\Tk::callbacks Tk::bindtags\Tk::bindtags

KEYWORDS

Event, binding

NAME

Tk::bindtags – Determine which bindings apply to a window, and order of evaluation

=for category Binding Events and Callbacks

SYNOPSIS

$widget->**bindtags***([tagList]); @tags* = *$widget*->**bindtags***;*

DESCRIPTION

When a binding is created with the **bind** command, it is associated either with a particular window such as *$widget,* a class name such as **Tk::Button**, the keyword **all**, or any other string. All of these forms are called *binding tags*. Each window has a list of binding tags that determine how events are processed for the window. When an event occurs in a window, it is applied to each of the window's tags in order: for each tag, the most specific binding that matches the given tag and event is executed. See the *Tk::bind* documentation for more information on the matching process.

By default, each window has four binding tags consisting of the the window's class name, name of the window, the name of the window's nearest toplevel ancestor, and **all**, in that order. Toplevel windows have only three tags by default, since the toplevel name is the same as that of the window.

Note that this order is *different* from order used by Tcl/Tk. Tcl/Tk has the window ahead of the class name in the binding order. This is because Tcl is procedural rather than object oriented and the normal way for Tcl/Tk applications to override class bindings is with an instance binding. However, with perl/Tk the normal way to override a class binding is to derive a class. The perl/Tk order causes instance bindings to execute after the class binding, and so instance bind callbacks can make use of state changes (e.g. changes to the selection) than the class bindings have made.

The **bindtags** command allows the binding tags for a window to be read and modified.

If *$widget*->**bindtags** is invoked without an argument, then the current set of binding tags for `$widget` is returned as a list. If the *tagList* argument is specified to **bindtags**, then it must be a reference to and array; the tags for `$widget` are changed to the elements of the array. (A reference to an anonymous array can be created by enclosin the elements in **[ ]**.) The elements of *tagList* may be arbitrary strings or widget objects, if no window exists for an object at the time an event is processed, then the tag is ignored for that event. The order of the elements in *tagList* determines the order in which binding callbacks are executed in response to events. For example, the command

```
$b->bindtags([$b,ref($b),$b->toplevel,'all'])
```

applies the Tcl/Tk binding order which binding callbacks will be evaluated for a button (say) **$b** so that **$b's** instance bindings are invoked first, following by bindings for **$b's** class, followed by bindings for **$b's** toplevel, followed by '**all**' bindings.

If *tagList* is an empty list i.e. **[]**, then the binding tags for `$widget` are returned to the perl/Tk default state described above.

The **bindtags** command may be used to introduce arbitrary additional binding tags for a window, or to remove standard tags. For example, the command

```
$b->bindtags(['TrickyButton',$b->toplevel,'all'])
```

replaces the (say) **Tk::Button** tag for **$b** with **TrickyButton**. This means that the default widget bindings for buttons, which are associated with the **Tk::Button** tag, will no longer apply to **$b,** but any bindings associated with **TrickyButton** (perhaps some new button behavior) will apply.

BUGS

The current mapping of the 'native' Tk behaviour of this method i.e. returning a list but only accepting a reference to an array is counter intuitive. The perl/Tk interface may be tidied up, returning a list is sensible so, most likely fix will be to allow a list to be passed to /fIset/fR the bindtags.

SEE ALSO

Tk::bind\Tk::bind Tk::callbacks\Tk::callbacks

KEYWORDS

binding, event, tag

NAME

Tk::Bitmap – Images that display two colors

=for category Tk Image Classes

SYNOPSIS

$image = *$widget*->**Bitmap***?(name??,options?)*

DESCRIPTION

A bitmap is an image whose pixels can display either of two colors or be transparent. A bitmap image is defined by four things: a background color, a foreground color, and two bitmaps, called the *source* and the *mask*. Each of the bitmaps specifies 0/1 values for a rectangular array of pixels, and the two bitmaps must have the same dimensions. For pixels where the mask is zero, the image displays nothing, producing a transparent effect. For other pixels, the image displays the foreground color if the source data is one and the background color if the source data is zero.

CREATING BITMAPS

Bitmaps are created using *$widget*->**Bitmap**. Bitmaps support the following *options*:

–background => *color*

Specifies a background color for the image in any of the standard ways accepted by Tk. If this option is set to an empty string then the background pixels will be transparent. This effect is achieved by using the source bitmap as the mask bitmap, ignoring any **–maskdata** or **–maskfile** options.

–data => *string*

Specifies the contents of the source bitmap as a string. The string must adhere to X11 bitmap format (e.g., as generated by the **bitmap** program). If both the **–data** and **–file** options are specified, the **–data** option takes precedence.

–file => *name*

name gives the name of a file whose contents define the source bitmap. The file must adhere to X11 bitmap format (e.g., as generated by the **bitmap** program).

–foreground => *color*

Specifies a foreground color for the image in any of the standard ways accepted by Tk.

–maskdata => *string*

Specifies the contents of the mask as a string. The string must adhere to X11 bitmap format (e.g., as generated by the **bitmap** program). If both the **–maskdata** and **–maskfile** options are specified, the **–maskdata** option takes precedence.

–maskfile => *name*

name gives the name of a file whose contents define the mask. The file must adhere to X11 bitmap format (e.g., as generated by the **bitmap** program).

IMAGE METHODS

When a bitmap image is created, Tk also creates a new object. This object supports the **configure** and **cget** methods described in *Tk::options* which can be used to enquire and modify the options described above.

SEE ALSO

Tk::Image|Tk::Image Tk::Pixmap|Tk::Pixmap Tk::Photo|Tk::Photo

KEYWORDS

bitmap, image

NAME

Tk::BrowseEntry – entry widget with popup choices.

=for pm Tixish/BrowseEntry.pm

=for category Tix Extensions

SYNOPSIS

```
use Tk::BrowseEntry;

$b = $frame->BrowseEntry(-label => "Label", -variable => \$var);
$b->insert("end", "opt1");
$b->insert("end", "opt2");
$b->insert("end", "opt3");
...
$b->pack;
```

DESCRIPTION

BrowseEntry is a poor man's ComboBox. It may be considered an enhanced version of LabEntry which provides a button to popup the choices of the possible values that the Entry may take. BrowseEntry supports all the options LabEntry supports except **–textvariable**. This is replaced by **–variable**. Other options that BrowseEntry supports.

–listwidth

Specifies the width of the popup listbox.

–variable

Specifies the variable in which the entered value is to be stored.

–browsecmd

Specifies a function to call when a selection is made in the popped up listbox. It is passed the widget and the text of the entry selected. This function is called after the entry variable has been assigned the value.

–listcmd

Specifies the function to call when the button next to the entry is pressed to popup the choices in the listbox. This is called before popping up the listbox, so can be used to populate the entries in the listbox.

–arrowimage

Specifies the image to be used in the arrow button beside the entry widget. The default is an downward arrow image in the file cbxarrow.xbm

–choices

Specifies the list of choices to pop up. This is a reference to an array of strings specifying the choices.

–state

Specifies one of three states for the widget: normal, readonly, or disabled. If the widget is disabled then the value may not be changed and the arrow button won't activate. If the widget is readonly, the entry may not be edited, but it may be changed by choosing a value from the popup listbox. normal is the default.

METHODS

insert(*index*, *string*)

Inserts the text of *string* at the specified *index*. This string then becomes available as one of the choices.

delete(*index1***, *index2*)**

Deletes items from *index1* to *index2*.

BUGS

BrowseEntry should really provide more of the ComboBox options.

AUTHOR

Rajappa Iyer rsi@earthling.net

Chris Dean ctdean@cogit.com made additions.

NAME

Tk::Button – Create and manipulate Button widgets

=for category Tk Widget Classes

SYNOPSIS

$button = *$parent*->**Button**(*?options?*);

STANDARD OPTIONS

–activebackground –cursor –highlightthickness –takefocus –activeforeground –disabledforeground –image –text –anchor –font –justify –textvariable –background –foreground –padx –underline –bitmap –highlightbackground–pady –wraplength –borderwidth –highlightcolor –relief

See *Tk::options* for details of the standard options.

WIDGET–SPECIFIC OPTIONS

Name: **command**
Class: **Command**
Switch: **–command**

Specifies a *Tk callback\Tk::callbacks* to associate with the button. This command is typically invoked when mouse button 1 is released over the button window.

Name: **default**
Class: **Default**
Switch: **–default**

Specifies one of three states for the default ring: **normal**, **active**, or **disabled**. In active state, the button is drawn with the platform specific appearance for a default button. In normal state, the button is drawn with the platform specific appearance for a non–default button, leaving enough space to draw the default button appearance. The normal and active states will result in buttons of the same size. In disabled state, the button is drawn with the non–default button appearance without leaving space for the default appearance. The disabled state may result in a smaller button than the active state. ring.

Name: **height**
Class: **Height**
Switch: **–height**

Specifies a desired height for the button. If an image or bitmap is being displayed in the button then the value is in screen units (i.e. any of the forms acceptable to **Tk_GetPixels**); for text it is in lines of text. If this option isn‘t specified, the button‘s desired height is computed from the size of the image or bitmap or text being displayed in it.

Name: **state**
Class: **State**
Switch: **–state**

Specifies one of three states for the button: **normal**, **active**, or **disabled**. In normal state the button is displayed using the **foreground** and **background** options. The active state is typically used when the pointer is over the button. In active state the button is displayed using the **activeForeground** and **activeBackground** options. Disabled state means that the button should be insensitive: the default bindings will refuse to activate the widget and will ignore mouse button presses. In this state the **disabledForeground** and **background** options determine how the button is displayed.

Name: **width**
Class: **Width**
Switch: **–width**

Specifies a desired width for the button. If an image or bitmap is being displayed in the button then the

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

DESCRIPTION

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

A button is a widget that displays a textual string, bitmap or image. If text is displayed, it must all be in a single font, but it can occupy multiple lines on the screen (if it contains newlines or if wrapping occurs because of the **–wraplength** option) and one of the characters may optionally be underlined using the **–underline** option. It can display itself in either of three different ways, according to the **–state** option; it can be made to appear raised, sunken, or flat; and it can be made to flash. When a user invokes the button (by pressing mouse button 1 with the cursor over the button), then the *Tk callback|Tk::callbacks* specified in the **–command** option is invoked.

WIDGET METHODS

The **Button** method creates a widget object. This object supports the **configure** and **cget** methods described in *Tk::options* which can be used to enquire and modify the options described above. The widget also inherits all the methods provided by the generic *Tk::Widget|Tk::Widget* class.

The following additional methods are available for button widgets:

$button–>**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**.

$button–>**invoke**

Invoke the *callback|Tk::callbacks* associated with the buttons **–command** option, if there is one. The return value is the return value from the callback, or the undefined value if there is no callback associated with the button. This command is ignored if the button's state is **disabled**.

DEFAULT BINDINGS

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

[1] A button activates whenever the mouse passes over it and deactivates whenever the mouse leaves the button. Under Windows, this binding is only active when mouse button 1 has been pressed over the button.

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

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

[4] When a button has the input focus, the space key causes the button to be invoked.

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

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

KEYWORDS

button, widget

NAME

Tk::callbacks – Specifying code for Tk to call.

=for category Binding Events and Callbacks

SYNOPSIS

One can specify a callback in one of the following ways:

Without arguments:

```
... => \&subname, ...
... => sub { ... }, ...
... => 'methodname', ...
```

or with arguments:

```
... => [ \&subname ?, args ...? ], ...
... => [ sub { ... } ?, args...? ], ...
... => [ 'methodname' ?, args...?], ...
```

DESCRIPTION

Perl/Tk has a callback, where Tcl/Tk has a command string (i.e. a fragment of Tcl to be executed). A perl/Tk callback can take one of the following basic forms:

- Reference to a subroutine `\&subname`
- Anonymous subroutine (closure) `sub { ... }`
- A method name `'methodname'`

Any of these can be provided with arguments by enclosing them and the arguments in **[]**. Here are some examples:

$mw*–**bind**(*$class, **"<Delete>" => 'Delete'**);

This will call *$widget*–**Delete**, the *$widget* being provided (by bind) as the one where the Delete key was pressed.

While having bind provide a widget object for you is ideal in many cases it can be irritating in others. Using the list form this behaviour can be modified:

$a*–>**bind**(**"<Delete>"**,*[$b => 'Delete']);

because the first element *$b* is an object bind will call *$b*–>**Delete**.

Note that method/object ordering only matters for `bind` callbacks, the auto–quoting in perl5.001 makes the first of these a little more readable:

`$w->configure(-yscrollcommand` => [set => `$ysb]);`

`$w->configure(-yscrollcommand` => [`$ysb` => 'set']);

but both will call `$ysb->set(args` provided by Tk)

Another use of arguments allows you to write generalized methods which are easier to re–use:

```
$a->bind("<Next>",['Next','Page']);
```

```
$a->bind("<Down>",['Next','Line']);
```

This will call `$a->`*Next*`('Page')` or `$a->`*Next*`('Line')` respectively.

Note that the contents of the `[]` are evaluated by perl when the callback is created. It is often desirable for the arguments provided to the callback to depend on the details of the event which caused it to be executed. To allow for this callbacks can be nested using the `Ev(...)` "constructor". `Ev(...)` inserts callback objects into the argument list. When perl/Tk glue code is preparing the argument list for the callback it is about to call it spots these special objects and recursively applies the callback process to them.

EXAMPLES

```
$entry->bind('<Return>' => [$w , 'validate', Ev(['get'])]);

$toplevel->bind('all', '<Visibility>', [\&unobscure, Ev('s')]);

$mw->bind($class, '<Down>', ['SetCursor', Ev('UpDownLine',1)]);
```

SEE ALSO

Tk::bind\|Tk::bind Tk::after\|Tk::after Tk::options\|Tk::options Tk::fileevent\|Tk::fileevent

KEYWORDS

callback, closure, anonymous subroutine, bind

NAME

Tk::Canvas – Create and manipulate Canvas widgets

=for category Tk Widget Classes

SYNOPSIS

$canvas = *$parent*->**Canvas***(?options?);*

STANDARD OPTIONS

–background –highlightthickness –insertwidth –state –borderwidth
–insertbackground –relief –tile –cursor –insertborderwidth
–selectbackground –takefocus –highlightbackground –insertofftime
–selectborderwidth –xscrollcommand –highlightcolor –insertontime
–selectforeground –yscrollcommand

WIDGET–SPECIFIC OPTIONS

Name: **closeEnough**
Class: **CloseEnough**
Switch: **–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.

Name: **confine**
Class: **Confine**
Switch: **–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.

Name: **height**
Class: **Height**
Switch: **–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.

Name: **scrollRegion**
Class: **ScrollRegion**
Switch: **–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.

Name: **state**
Class: **State**
Switch: **–state**

Modifies the default state of the canvas where *state* may be set to one of: normal, disabled, or hidden. Individual canvas objects all have their own state option, which overrides the default state. Many options can take separate specifications such that the appearance of the item can be different in different situations. The options that start with "active" control the appearence when the mouse pointer is over it, while the option starting with "disabled" controls the appearence when the state is disabled.

Name: **width**
Class: **width**
Switch: **–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.

Name: **xScrollIncrement**
Class: **ScrollIncrement**
Switch: **–xscrollincrement**

Specifies an increment for horizontal scrolling, in any of the usual forms permitted for screen distances. If the value of this option is greater than zero, the horizontal view in the window will be constrained so that the canvas x coordinate at the left edge of the window is always an even multiple of **xScrollIncrement**; furthermore, the units for scrolling (e.g., the change in view when the left and right arrows of a scrollbar are selected) will also be **xScrollIncrement**. If the value of this option is less than or equal to zero, then horizontal scrolling is unconstrained.

Name: **yScrollIncrement**
Class: **ScrollIncrement**
Switch: **–yscrollincrement**

Specifies an increment for vertical scrolling, in any of the usual forms permitted for screen distances. If the value of this option is greater than zero, the vertical view in the window will be constrained so that the canvas y coordinate at the top edge of the window is always an even multiple of **yScrollIncrement**; furthermore, the units for scrolling (e.g., the change in view when the top and bottom arrows of a scrollbar are selected) will also be **yScrollIncrement**. If the value of this option is less than or equal to zero, then vertical scrolling is unconstrained.

DESCRIPTION

The **Canvas** method creates a new window (given by the `$canvas` argument) and makes it into a canvas widget. Additional options, described above, may be specified on the command line or in the option database to configure aspects of the canvas such as its colors and 3–D relief. The **canvas** command returns its `$canvas` argument. At the time this command is invoked, there must not exist a window named `$canvas`, but `$canvas'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 *callbacks\Tk::callbacks* may be associated with items in much the same way that the *bind\Tk::bind* method allows callbacks to be bound to widgets. For example, a particular callback may be associated with the **<Button–1>** event so that the callback 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 Callbacks 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 methods may be used to re–arrange the order of the display list.

Window items are an exception to the above rules. The underlying window systems require them always to be drawn on top of other items. In addition, the stacking order of window items is not affected by any of the canvas methods; you must use the *raise\Tk::Widget* and *lower\Tk::Widget* Tk widget methods instead.

ITEM IDS AND TAGS

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

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

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

the items in the canvas.

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

When specifying items in canvas methods, 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.

tagOrId may contain a logical expressions of tags by using operators: '&&', '||', '^' '!', and parenthezised subexpressions. For example:

```
$c->find('withtag', '(a&&!b)|(!a&&b)');
```

or equivalently:

```
$c->find('withtag', 'a^b');
```

will find only those items with either "a" or "b" tags, but not both.

Some methods 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 methods is to use the first (lowest) of these items in the display list that is suitable for the method. Exceptions are noted in the method 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** methods; this is typically used for scrolling. Canvases do not support scaling or rotation of the canvas coordinate system relative to the window coordinate system.

Individual items may be moved or scaled using methods 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 methods 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.

In a similar fashion, line and polygon items support *index* for identifying, inserting and deleting subsets of their coordinates. Indices are used for commands such as inserting or deleting a range of characters or coordinates, 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. Lines and Polygons don't support the insertion cursor and the selection. Their indices are supposed to be even always, because coordinates always appear in pairs.

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. If indexes are odd for lines and polygons, they will be automatically decremented by one. 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. For polygons, numbers less than 0 or greater then the length of the coordinate list will be adjusted by adding or substracting the length until the result is between zero and the length, inclusive.

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

insert

Refers to the character just before which the insertion cursor is drawn in this item. Not valid for lines and polygons.

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 or coordinate 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. The Tcl string form "@x,y" is also allowed.

DASH PATTERNS

Many items support the notion of an dash pattern for outlines.

The first possible syntax is a list of integers. Each element represents the number of pixels of a line segment. Only the odd segments are drawn using the "outline" color. The other segments are drawn transparant.

The second possible syntax is a character list containing only 5 possible characters **[.,–_]**. The space can be used to enlarge the space between other line elements, and can not occur as the first position in the string. Some examples:

```
-dash .      = -dash [2,4]
-dash -      = -dash [6,4]
-dash -.     = -dash [6,4,2,4]
-dash -..    = -dash [6,4,2,4,2,4]
-dash '. '   = -dash [2,8]
-dash ',     = -dash [4,4]
```

The main difference of this syntax with the previous is that it it shape–conserving. This means that all values in the dash list will be multiplied by the line width before display. This assures that "." will always be displayed as a dot and "–" always as a dash regardless of the line width.

On systems where only a limited set of dash patterns, the dash pattern will be displayed as the most close dash pattern that is available. For example, on Windows only the first 4 of the above examples are available. The last 2 examples will be displayed identically as the first one.

WIDGET METHODS

The **Canvas** method creates a widget object. This object supports the **configure** and **cget** methods described in *Tk::options* which can be used to enquire and modify the options described above. The widget also inherits all the methods provided by the generic *Tk::Widget\Tk::Widget* class.

The following additional methods are available for canvas widgets:

$canvas->**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 method 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*.

$canvas->**bbox**(*tagOrId,* ?*tagOrId, tagOrId, ...*?)

Returns a list with four elements giving an approximate bounding box for all the items named by the *tagOrId* arguments. The list has the form ''*x1 y1 x2 y2*'' such that the drawn areas of all the named elements are within the region bounded by *x1* on the left, *x2* on the right, *y1* on the top, and *y2* on the bottom. The return value may overestimate the actual bounding box by a few pixels. If no items match any of the *tagOrId* arguments or if the matching items have empty bounding boxes (i.e. they have nothing to display) then an empty string is returned.

$canvas->**bind**(*tagOrId?, sequence*? ?,*callback*?)

This method associates *callback* with all the items given by *tagOrId* such that whenever the event sequence given by *sequence* occurs for one of the items the callback will be invoked. This method is similar to the **bind** method except that it operates on items in a canvas rather than entire widgets. See *Tk::bind* for complete details on the syntax of *sequence* and the substitutions performed on *callback* 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 *callback* is omitted then the method returns the *callback* associated with *tagOrId* and *sequence* (an

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

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

It is possible for multiple bindings to match a particular event. This could occur, for example, if one binding is associated with the item's id and another is associated with one of the item's tags. When this occurs, all of the matching bindings are invoked. A binding associated with the **all** tag is invoked first, followed by one binding for each of the item's tags (in order), followed by a binding associated with the item's id. If there are multiple matching bindings for a single tag, then only the most specific binding is invoked. A **continue** in a callback terminates that subroutine, and a **break** method terminates that subroutine and skips any remaining callbacks for the event, just as for the **bind** method.

If bindings have been created for a canvas window using the **CanvasBind** method, then they are invoked in addition to bindings created for the canvas's items using the **bind** method. The bindings for items will be invoked before any of the bindings for the window as a whole.

`$canvas`−>**canvasx**(*screenx?, gridspacing*?)

Given a window x−coordinate in the canvas *screenx*, this method 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.

`$canvas`−>**canvasy**(*screeny, ?gridspacing*?)

Given a window y−coordinate in the canvas *screeny* this method 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.

`$canvas`−>**coords**(*tagOrId ?x0,y0 ...*?)

Query or modify the coordinates that define an item. If no coordinates are specified, this method 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.

`$canvas`−>**create**(*type, x, y, ?x, y, ...*?, ?*option, value, ...*?)

Create a new item in `$canvas` 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 method. This method returns the id for the new item.

`$canvas`−>**dchars**(*tagOrId, first, ?last*?)

For each item given by *tagOrId*, delete the characters, or coordinates, in the range given by *first* and *last*, inclusive. If some of the items given by *tagOrId* don't support Text items interpret *first* and *last* as indices to a character, line and polygon items interpret them indices to a coordinate (an x,y pair). within the item(s) as described in *"INDICES"* above. If *last* is omitted, it defaults to *first*. This method returns an empty string.

$canvas->**delete***(?tagOrId, tagOrId, ...?)*

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

$canvas->**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 method. If *tagToDelete* is omitted then it defaults to *tagOrId*. This method returns an empty string.

$canvas->**find***(searchCommand, ?arg, arg, ...?)*

This method 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** method. The items are returned in stacking order, with the lowest item first.

focus

$canvas->**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 method 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** method) 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 method with the **CanvasFocus** method to set the focus window to the canvas (if it wasn't there already).

$canvas->**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.

$canvas->**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 method 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 method **focus**, below), but the cursor position may be set even when the item doesn't have the focus. This method returns an empty string.

$canvas->**index***(tagOrId, index)*

This method 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. Text items interpret *index* as an index to a character, line and polygon items interpret it as an index to a coordinate (an x,y pair). The return value is guaranteed to lie between 0 and the number of characters, or coordinates, 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).

$canvas->**insert***(tagOrId, beforeThis, string)*

For each of the items given by *tagOrId*, if the item supports text or coordinate, insertion then *string* is inserted into the item's text just before the character, or coordinate, whose index is *beforeThis*. Text items interpret *beforethis* as an index to a character, line and polygon items interpret it as an index to a coordinate (an x,y pair). For lines and polygons the *string* must be a valid coordinate sequence. See

"INDICES" above for information about the forms allowed for *beforeThis*. This method returns an empty string.

`$canvas->`**itemcget**(*tagOrId, option*)

Returns the current value of the configuration option for the item given by *tagOrId* whose name is *option*. This method is similar to the *cget\Tk::option* method except that it applies to a particular item rather than the widget as a whole. *Option* may have any of the values accepted by the **create** method when the item was created. If *tagOrId* is a tag that refers to more than one item, the first (lowest) such item is used.

`$canvas->`**itemconfigure**(*tagOrId,* ?*option*?, ?*value*?, ?*option, value, ...*?)

This method is similar to the *configure\Tk::option* method 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::options* for information on the format of this list). If *option* is specified with no *value*, then the method 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 method modifies the given widget option(s) to have the given value(s) in each of the items given by *tagOrId*; in this case the method returns an empty string. The *option*s and *value*s are the same as those permissible in the **create** method when the item(s) were created; see the sections describing individual item types below for details on the legal options.

`$canvas->`**lower**(*tagOrId,* ?*belowThis*?)

Move all of the items given by *tagOrId* to a new position in the display list just before the item given by *belowThis*. If *tagOrId* refers to more than one item then all are moved but the relative order of the moved items will not be changed. *BelowThis* is a tag or id; if it refers to more than one item then the first (lowest) of these items in the display list is used as the destination location for the moved items. Note: this method has no effect on window items. Window items always obscure other item types, and the stacking order of window items is determined by the **raise** and **lower** methods of the widget, not the **raise** and **lower** methods for canvases. This method returns an empty string.

`$canvas->`**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 method returns an empty string.

`$canvas->`**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 method. If the interpreter that owns the canvas is marked as safe, the operation will fail because safe interpreters are not allowed to write files. If the **-channel** option is specified, the argument denotes the name of a channel already opened for writing. The Postscript is written to that channel, and the channel is left open for further writing at the end of the operation. The Postscript is created in Encapsulated Postscript form using version 3.0 of the Document Structuring Conventions. Note: by default Postscript is only generated for information that appears in the canvas's window on the screen. If the canvas is freshly created it may still have its initial size of 1x1 pixel so nothing will appear in the Postscript. To get around this problem either invoke the **update** method to wait for the canvas window to reach its final size, or else use the **-width** and **-height** options to specify the area of the canvas to print. The *option-value* argument pairs provide additional information to control the generation of Postscript. The following options are supported:

-colormap => *hashRef*

HashRef must be a reference to a hash variable or an anonymous hash that specifies a color mapping to use in the Postscript. Each value of the hash 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 a key in the hash with the same

name as the color. If so, Tk uses the value of the element as the Postscript method to set the color. If this option hasn't been specified, or if there isn't a key in *hashRef* 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 method instead of being written to a file.

–fontmap => *hashRef*

HashRef must be a reference to a hash variable or an anonymous hash that specifies a font mapping to use in the Postscript. Each value of the hash must consist of an array reference 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 *hashRef* contains a value with the same name as the font. If there is such an element, then the font information contained in that element is used in the Postscript. Otherwise Tk attempts to guess what Postscript font to use. Tk's guesses generally only work for well–known fonts such as Times and Helvetica and Courier, and only if the X font name does not omit any dashes up through the point size. For example, **–*–Courier–Bold–R–Normal—*–120–*** will work but ***Courier–Bold–R–Normal*120**** will not; Tk needs the dashes to parse the font name).

–height => *size*

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

–pageanchor => *anchor*

Specifies which point of the printed area of the canvas should appear over the positioning point on the page (which is given by the **–pagex** and **–pagey** options). For example, **–pageanchor=n** means that the top center of the area of the canvas being printed (as it appears in the canvas window) should be over the positioning point. Defaults to **center**.

–pageheight => *size*

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

–pagewidth => *size*

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

–pagex => *position*

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

–pagey => *position*

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

–rotate => *boolean*

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

–width => *size*

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

–x = *position*

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

–y = *position*

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

`$canvas`–>**raise**(*tagOrId, ?aboveThis?*)

Move all of the items given by *tagOrId* to a new position in the display list just after the item given by *aboveThis*. If *tagOrId* refers to more than one item then all are moved but the relative order of the moved items will not be changed. *AboveThis* is a tag or id; if it refers to more than one item then the last (topmost) of these items in the display list is used as the destination location for the moved items. Note: this method has no effect on window items. Window items always obscure other item types, and the stacking order of window items is determined by the **raise** and **lower** widget commands, not the **raise** and **lower** methods for canvases. This method returns an empty string.

`$canvas`–>**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 method returns an empty string.

`$canvas`–>**scan**(*option, args*)

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

`$canvas`–>**scanMark**(*x, y*)

Records *x* and *y* and the canvas's current view; used in conjunction with later **scanDragto** method. Typically this method 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.

`$canvas`–>**scanDragto**(*x, y, ?gain?.*)

This method computes the difference between its *x* and *y* arguments (which are typically mouse coordinates) and the *x* and *y* arguments to the last **scanMark** method for the widget. It then adjusts the view by 10 times the difference in coordinates. This method is typically associated It then adjusts the view by *gain* times the difference in coordinates, where *gain* defaults to 10. 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.

$canvas->**select***(option, ?tagOrId, arg?)*

Manipulates the selection in one of several ways, depending on *option*. The method 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.

$canvas->**selectAdjust***(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 **selectTo** method calls. If the selection isn't currently in *tagOrId* then this method behaves the same as the **selectTo** widget method. Returns an empty string.

$canvas->**selectClear**

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

$canvas->**selectFrom***(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 method doesn't change the selection; it just sets the fixed end of the selection for future **selectTo** method calls. Returns an empty string.

$canvas->**selectItem**

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.

$canvas->**selectTo***(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 **selectAdjust** or **selectFrom** method calls 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.

$canvas->**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.

$canvas->**xview***(?args?)*

This method is used to query and change the horizontal position of the information displayed in the canvas's window. It can take any of the following forms:

$canvas->**xview**

Returns a list containing two elements. Each element is a real fraction between 0 and 1; together they describe the horizontal span that is visible in the window. For example, if the first element is .2 and the second element is .6, 20% of the canvas's area (as defined by the **-scrollregion** option) is off-screen to the left, the middle 40% is visible in the window, and 40% of the canvas is off-screen to the right. These are the same values passed to scrollbars via the **-xscrollcommand** option.

$canvas->**xviewMoveto***(fraction)*

Adjusts the view in the window so that *fraction* of the total width of the canvas is off-screen to the left. *Fraction* must be a fraction between 0 and 1.

$canvas–>**xviewScroll***(number, what***)**

This method shifts the view in the window left or right according to *number* and *what*. *Number* must be an integer. *What* must be either **units** or **pages** or an abbreviation of one of these. If *what* is **units**, the view adjusts left or right in units of the **xScrollIncrement** option, if it is greater than zero, or in units of one–tenth the window's width otherwise. If *what is* **pages** then the view adjusts in units of nine–tenths the window's width. If *number* is negative then information farther to the left becomes visible; if it is positive then information farther to the right becomes visible.

$canvas–>**yview***(?args?)*

This method is used to query and change the vertical position of the information displayed in the canvas's window. It can take any of the following forms:

$canvas–>**yview**

Returns a list containing two elements. Each element is a real fraction between 0 and 1; together they describe the vertical span that is visible in the window. For example, if the first element is .6 and the second element is 1.0, the lowest 40% of the canvas's area (as defined by the **–scrollregion** option) is visible in the window. These are the same values passed to scrollbars via the **–yscrollcommand** option.

$canvas–>**yviewMoveto***(fraction)*

Adjusts the view in the window so that *fraction* of the canvas's area is off–screen to the top. *Fraction* is a fraction between 0 and 1.

$canvas–>**yviewScroll***(number, what*)

This method adjusts the view in the window up or down according to *number* and *what*. *Number* must be an integer. *What* must be either **units** or **pages**. If *what* is **units**, the view adjusts up or down in units of the **yScrollIncrement** option, if it is greater than zero, or in units of one–tenth the window's height otherwise. If *what* is **pages** then the view adjusts in units of nine–tenths the window's height. If *number* is negative then higher information becomes visible; if it is positive then lower information becomes visible.

OVERVIEW OF ITEM TYPES

The sections below describe the various types of items supported by canvas widgets. Each item type is characterized by two things: first, the form of the **create** method 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** methods. Most items don't support indexing or selection or the methods related to them, such as **index** and **insert**. Where items do support these facilities, it is noted explicitly in the descriptions below. At present, text, line and polygon items provide this support. For lines and polygons the indexing facility is used to manipulate the coordinates of the item.

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 methods of the following form:

```
$canvas->createArc(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** methods to change the item's configuration. The following options are supported for arcs:

–dash => *pattern*

–activedash => *pattern*
–disableddash => *pattern*

This option specifies dash patterns for the normal state, the active state, and the disabled state of an arc item. *pattern* may have any of the forms accepted by **Tk_GetDash**. If the dash options are omitted then the default is a solid outline.

–dashoffset => *offset*

The starting *offset* into the pattern provided by the **–dash** option. **–dashoffset** is ignored if there is no **–dash** pattern.

–extent => *degrees*

Specifies the size of the angular range occupied by the arc. The arc's range extends for *degrees* degrees counter–clockwise from the starting angle given by the **–start** option. *Degrees* may be negative. If it is greater than 360 or less than –360, then *degrees* modulo 360 is used as the extent.

–fill => *color*
–activefill => *color*
–disabledfill => *color*

Specifies the color to be used to fill the arc region in its normal, active, and disabled states, *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*
–activeoutline => *color*
–disabledoutline => *color*

This option specifies the color that should be used to draw the outline of the arc in its normal, active and disabled states. *Color* may have any of the forms accepted by **Tk_GetColor**. This option defaults to **black**. If *color* is specified as undef then no outline is drawn for the arc.

–outlinestipple => *bitmap*
–activeoutlinestipple => *bitmap*
–disabledoutlinestipple => *bitmap*

This option specifies stipple patterns that should be used to draw the outline of the arc in its normal, active and disabled states. Indicates that the outline for the arc should be drawn with a stipple pattern; *bitmap* specifies the stipple pattern to use, in any of the forms accepted by **Tk_GetBitmap**. If the **–outline** option hasn't been specified then this option has no effect. If *bitmap* is an empty string (the default), then the outline is drawn in a solid fashion.

–start => *degrees*

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

–state => *state*

Modifies the state of the arc item where *state* may be set to one of: normal, disabled, hidden or "". If set to empty, the state of the canvas itself is used. An arc item may also be in the "active" state if the mouse is currently over it. Many options can take separate specifications in normal, active and disabled states such that the appearance of the item can be different in each state.

–stipple => *bitmap*
–activestipple => *bitmap*
–disabledstipple => *bitmap*

This option specifies stipple patterns that should be used to fill the the arc in its normal, active and disabled states. *bitmap* specifies the stipple pattern to use, in any of the forms accepted by **Tk_GetBitmap**. If the **–fill** option hasn't been specified then this option has no effect. If *bitmap* is an empty string (the default), then filling is done in a solid fashion.

–style => *type*

Specifies how to draw the arc. If *type* is **pieslice** (the default) then the arc's region is defined by a section of the oval's perimeter plus two line segments, one between the center of the oval and each end of the perimeter section. If *type* is **chord** then the arc's region is defined by a section of the oval's perimeter plus a single line segment connecting the two end points of the perimeter section. If *type* is **arc** then the arc's region consists of a section of the perimeter alone. In this last case the **–fill** option is ignored.

–tags => *tagList*

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

–updatecommand => *command*

Specifies a callback that is to be executed every time the arc item is updated on the screen.

–width => *outlineWidth*
–activewidth => *outlineWidth*
–disabledwidth => *outlineWidth*

Specifies the width of the outline to be drawn around the arc's region, in its normal, active and disabled states. *outlineWidth* may be in any of the forms described in the *"COORDINATES"* section above. If the **–outline** option has been specified as undef 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 methods of the following form:

```
$canvas->createBitmap(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** methods 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*
–activebackground => *color*
–disabledbackground => *color*

Specifies the color to use for each of the bitmap's '0' valued pixels in its normal, active and disabled states. *Color* may have any of the forms accepted by **Tk_GetColor**. If this option isn't specified, or if it is specified as undef, then nothing is displayed where the bitmap pixels are 0; this produces a transparent effect.

–bitmap => *bitmap*
–activebitmap => *bitmap*
–disabledbitmap => *bitmap*

Specifies the bitmaps to display in the item in its normal, active and disabled states. *Bitmap* may have any of the forms accepted by **Tk_GetBitmap**.

–foreground => *color*
–activeforeground => *bitmap*
–disabledforeground => *bitmap*

Specifies the color to use for each of the bitmap's '1' valued pixels in its normal, active and disabled states. *Color* may have any of the forms accepted by **Tk_GetColor** and defaults to **black**.

–state => *state*

Modifies the state of the bitmap item where *state* may be set to one of: normal, disabled, or hidden. An bitmap item may also be in the "active" state if the mouse is currently over it. Many options can take separate specifications in normal, active and disabled states such that the appearance of the item can be different in each state.

–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.

–updatecommand => *command*

Specifies a callback that is to be executed every time the bitmap item is updated on the screen.

GRID ITEMS

Items of type **grid** are intended for producing a visual reference for interpreting other items. They can be drawn as either lines (with dash style) or as rectangular "dots" at each grid point.

Items of type **grid** are unlike other items they always cover the whole of the canvas, but are never enclosed by nor overlap any area and are not near any point. That is they are intended to be always visible but not "pickable", as such they do support the "active" state. They are like other items in that: multiple grids are permitted, they can be raised and lowered relative to other items, they can be moved and scaled. As yet grids do not apear in PostScript output.

Grids have outline like configure options. Grids are created with methods of the following form:

```
$canvas->createGrid(x1, y1, x2, y2, ?option, value, option, value, ...?)
```

The arguments *x1*, *y1* give the origin of the grid. *x2*, and *y2* give the coordinates of the next grid point in their respective directions. 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** methods to change the item's configuration. The following options are supported for grids:

–lines => *boolean*

If **–lines** is set to a true value then lines are drawn for both X and Y grids in the style determined by **–dash**. Otherwise retangular "dots" are drawn at each grid point.

–dash => *pattern*
–disableddash => *pattern*

This option specifies dash patterns for the normal state, and the disabled state of a grid item. *pattern* may have any of the forms accepted by **Tk_GetDash**. If the dash options are omitted then the default is a solid outline.

–dashoffset => *offset*

The starting *offset* into the pattern provided by the **–dash** option. **–dashoffset** is ignored if there is no **–dash** pattern.

–color => *color*
–disabledcolor => *color*

This option specifies the color that should be used to draw the outline of the grid in its normal and disabled states. *Color* may have any of the forms accepted by **Tk_GetColor**. This option defaults to **black**. If *color* is undef then no grid will be drawn.

–stipple => *bitmap*
–disabledstipple => *bitmap*

This option specifies stipple patterns that should be used to draw the outline of the rectangle in its normal and disabled states. *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 outline is drawn in a solid fashion.

–state => *state*

Modifies the state of the rectangle item where *state* may be set to one of: normal, disabled, or hidden. Many options can take separate specifications in normal and disabled states such that the appearance of the item can be different in each state.

–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.

–updatecommand => *command*

Specifies a callback that is to be executed every time the grid item is updated on the screen.

–width => *outlineWidth*
–disabledwidth => *outlineWidth*

Specifies the width of the lines drawn by the grid or the size (in both X and Y) of the dots, in its normal and disabled states. This option defaults to 1.0.

IMAGE ITEMS

Items of type **image** are used to display images on a canvas. Images are created with methods of the following form:

```
$canvas->createImage(x, y, ?option, value, option, value, ...?)
```

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

–anchor => *anchorPos*

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

–image => *name*
–activeimage => *name*
–disabledimage => *name*

Specifies the name of the images to display in the item in is normal, active and disabled states. This image must have been created previously with the **imageCreate** method.

–state => *state*

Modifies the state of the image item where *state* may be set to one of: normal, disabled, or hidden. An image item may also be in the "active" state if the mouse is currently over it. Many options can take separate specifications in normal, active and disabled states such that the appearance of the item can be different in each state.

–tags => *tagList*

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

–updatecommand => *command*

Specifies a callback that is to be executed every time the image item is updated on the screen.

LINE ITEMS

Items of type **line** appear on the display as one or more connected line segments or curves. Line items support coordinate indexing operations using the canvas methods: **dchars, index, insert.** Lines are created with methods of the following form:

```
$canvas->createLine(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** methods 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.

–dash => *pattern*
–activedash => *pattern*
–disableddash => *pattern*

This option specifies dash patterns for the normal state, the active state, and the disabled state of a line item. *pattern* may have any of the forms accepted by **Tk_GetDash**. If the dash options are omitted then the default is a solid outline.

–dashoffset => *offset*

The starting *offset* into the pattern provided by the **–dash** option. **–dashoffset** is ignored if there is no **–dash** pattern.

–fill => *color*
–activefill => *color*
–disabledfill => *color*

Specifies the color to be used to fill the line in its normal, active, and disabled states. *Color* may have any of the forms acceptable to **Tk_GetColor**. It may also be undef, in which case the line will be transparent. This option defaults to **black**.

–joinstyle => *style*

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

–smooth => *boolean*

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

–splinesteps => *number*

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

–state => *state*

Modifies the state of the line item where *state* may be set to one of: normal, disabled, or hidden. A line item may also be in the "active" state if the mouse is currently over it. Many options can take separate specifications in normal, active and disabled states such that the appearance of the item can be different in each state.

–stipple => *bitmap*
–activestipple => *bitmap*
–disabledstipple => *bitmap*

This option specifies stipple patterns that should be used to fill the the line in its normal, active and disabled states. *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.

–updatecommand => *command*

Specifies a callback that is to be executed every time the line item is updated on the screen.

–width => *lineWidth*
–activewidth => *lineWidth*
–disabledwidth => *lineWidth*

Specifies the width of the line in its normal, active and disabled states. *lineWidth* may be 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 methods of the following form:

```
$canvas->createOval(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** methods to change the item's configuration. The following options are supported for ovals:

–dash => *pattern*
–activedash => *pattern*
–disableddash => *pattern*

This option specifies dash patterns for the normal state, the active state, and the disabled state of an oval item. *pattern* may have any of the forms accepted by **Tk_GetDash**. If the dash options are omitted then the default is a solid outline.

–dashoffset => *offset*

The starting *offset* into the pattern provided by the **–dash** option. **–dashoffset** is ignored if there is no **–dash** pattern.

–fill => *color*
–activefill => *color*
–disabledfill => *color*

Specifies the color to be used to fill the oval in its normal, active, and disabled states. *Color* may have any of the forms accepted by **Tk_GetColor**. If *color* is undef (the default), then then the oval will not be filled.

–outline => *color*
–activeoutline => *color*
–disabledoutline => *color*

This option specifies the color that should be used to draw the outline of the oval in its normal, active and disabled states. *Color* may have any of the forms accepted by **Tk_GetColor**. This option defaults to **black**. If *color* is undef then no outline will be drawn for the oval.

–outlinestipple => *bitmap*
–activeoutlinestipple => *bitmap*
–disabledoutlinestipple => *bitmap*

This option specifies stipple patterns that should be used to draw the outline of the oval in its normal, active and disabled states. *bitmap* specifies the stipple pattern to use, in any of the forms accepted by **Tk_GetBitmap**. If the **–outline** option hasn't been specified then this option has no effect. If *bitmap* is an empty string (the default), then the outline is drawn in a solid fashion.

–state => *state*

Modifies the state of the oval item where *state* may be set to one of: normal, disabled, or hidden. An oval item may also be in the "active" state if the mouse is currently over it. Many options can take separate specifications in normal, active and disabled states such that the appearance of the item can be different in each state.

–stipple => *bitmap*
–activestipple => *bitmap*
–disabledstipple => *bitmap*

This option specifies stipple patterns that should be used to fill the the oval in its normal, active and disabled states. *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.

–updatecommand => *command*

Specifies a callback that is to be executed every time the oval item is updated on the screen.

–width => *outlineWidth*
–activewidth => *outlineWidth*
–disabledwidth => *outlineWidth*

Specifies the width of the outline to be drawn around the oval, in its normal, active and disabled states. *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. Polygon items support coordinate indexing operations using the canvas methods: **dchars, index, insert.** Polygons are created with methods of the following form:

```
$canvas->createPolygon(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** methods to change the item's configuration. The following options are supported for polygons:

–dash => *pattern*
–activedash => *pattern*
–disableddash => *pattern*

This option specifies dash patterns for the normal state, the active state, and the disabled state of an polygon item. *pattern* may have any of the forms accepted by **Tk_GetDash**. If the dash options are omitted then the default is a solid outline.

–dashoffset => *offset*

The starting *offset* into the pattern provided by the **–dash** option. **–dashoffset** is ignored if there is no **–dash** pattern.

–fill => *color*
–activefill => *color*
–disabledfill => *color*

Specifies the color to be used to fill the polygon in its normal, active, and disabled states. *Color* may have any of the forms acceptable to **Tk_GetColor**. If *color* is undef then the polygon 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 outline. *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**.

–outline => *color*
–activeoutline => *color*
–disabledoutline => *color*

This option specifies the color that should be used to draw the outline of the polygon in its normal, active and disabled states. *Color* may have any of the forms accepted by **Tk_GetColor**. If *color* is undef then no outline will be drawn for the polygon. This option defaults to undef (no outline).

–outlinestipple => *bitmap*
–activeoutlinestipple => *bitmap*
–disabledoutlinestipple => *bitmap*

This option specifies stipple patterns that should be used to draw the outline of the polygon in its normal, active and disabled states. *bitmap* specifies the stipple pattern to use, in any of the forms accepted by **Tk_GetBitmap**. If the **–outline** option hasn't been specified then this option has no effect. If *bitmap* is an empty string (the default), then the outline is drawn in a solid fashion.

–smooth => *boolean*

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

–state => *state*

Modifies the state of the polygon item where *state* may be set to one of: normal, disabled, or hidden. A polygon item may also be in the "active" state if the mouse is currently over it. Many options can take separate specifications in normal, active and disabled states such that the appearance of the item can be different in each state.

–stipple => *bitmap*
–activestipple => *bitmap*
–disabledstipple => *bitmap*

This option specifies stipple patterns that should be used to fill the the polygon in its normal, active and disabled states. *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.

–updatecommand => *command*

Specifies a callback that is to be executed every time the polygon item is updated on the screen.

–width => *outlineWidth*
–activewidth => *outlineWidth*
–disabledwidth => *outlineWidth*
Specifies the width of the outline to be drawn around

the polygon, in its normal, active and disabled states. *outlineWidth* may be in any of the forms described in the COORDINATES section above. *OutlineWidth* specifies the width of the outline to be drawn around the polygon, in any of the forms described in the *"COORDINATES"* section above. If the **–outline** option hasn't been specified then this option has no effect. This option defaults to 1.0.

Polygon items are different from other items such as rectangles, ovals and arcs in that interior points are considered to be ''inside'' a polygon (e.g. for purposes of the **find closest** and **find overlapping** methods) even if it is not filled. For most other item types, an interior point is considered to be inside the item only if the item is filled or if it has neither a fill nor an outline. If you would like an unfilled polygon whose interior points are not considered to be inside the polygon, use a line item instead.

RECTANGLE ITEMS

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

```
$canvas->createRectangle(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** methods to change the item's configuration. The following options are supported for rectangles:

–dash => *pattern*
–activedash => *pattern*
–disableddash => *pattern*

This option specifies dash patterns for the normal state, the active state, and the disabled state of a rectangle item. *pattern* may have any of the forms accepted by **Tk_GetDash**. If the dash options are omitted then the default is a solid outline.

–dashoffset => *offset*

The starting *offset* into the pattern provided by the **–dash** option. **–dashoffset** is ignored if there is no **–dash** pattern.

–fill => *color*
–activefill => *color*
–disabledfill => *color*

Specifies the color to be used to fill the rectangle in its normal, active, and disabled states. *Color* may be specified in any of the forms accepted by **Tk_GetColor**. If *color* is undef (the default), then the rectangle will not be filled.

–outline => *color*
–activeoutline => *color*
–disabledoutline => *color*

This option specifies the color that should be used to draw the outline of the rectangle in its normal, active and disabled states. *Color* may have any of the forms accepted by **Tk_GetColor**. This option defaults to **black**. If *color* is undef then no outline will be drawn for the rectangle.

–outlinestipple => *bitmap*
–activeoutlinestipple => *bitmap*
–disabledoutlinestipple => *bitmap*

This option specifies stipple patterns that should be used to draw the outline of the rectangle in its normal, active and disabled states. *bitmap* specifies the stipple pattern to use, in any of the forms accepted by **Tk_GetBitmap**. If the **–outline** option hasn't been specified then this option has no effect. If *bitmap* is an empty string (the default), then the outline is drawn in a solid fashion.

–state => *state*

Modifies the state of the rectangle item where *state* may be set to one of: normal, disabled, or hidden. A rectangle item may also be in the "active" state if the mouse is currently over it. Many options can take separate specifications in normal, active and disabled states such that the appearance of the item can be different in each state.

–stipple => *bitmap*
–activestipple => *bitmap*
–disabledstipple => *bitmap*

This option specifies stipple patterns that should be used to fill the the rectangle in its normal, active and disabled states. *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.

–updatecommand => *command*

Specifies a callback that is to be executed every time the rectangle item is updated on the screen.

–width => *outlineWidth*
–activewidth => *outlineWidth*
–disabledwidth => *outlineWidth*

Specifies the width of the outline to be drawn around the rectangle, in its normal, active and disabled states. *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 methods: **dchars**, **focus**, **icursor**, **index**, **insert**, **select**. Text items are created with methods of the following form:

```
$canvas->createText(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** methods 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*
−activefill => *color*
−disabledfill => *color*

Specifies the color to be used to fill the text in its normal, active, and disabled states. *Color* may have any of the forms accepted by **Tk_GetColor**. If *color* is undef then the text will be transparent. 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**.

−state => *state*

Modifies the state of the text item where *state* may be set to one of: normal, disabled, or hidden. A text item may also be in the "active" state if the mouse is currently over it. Many options can take separate specifications in normal, active and disabled states such that the appearance of the item can be different in each state.

−stipple => *bitmap*
−activestipple => *bitmap*
−disabledstipple => *bitmap*

This option specifies stipple patterns that should be used to fill the the text in its normal, active and disabled states. *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** methods. This option defaults to an empty string.

–updatecommand => *command*

Specifies a callback that is to be executed every time the text item is updated on the screen.

–width => *lineLength*

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

WINDOW ITEMS

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

$canvas–**createWindow**(*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** method 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.

–state => *state*

Modifies the state of the window item where *state* may be set to one of: normal, disabled, or hidden.

–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.

–updatecommand => *command*

Specifies a callback that is to be executed every time the window item is updated on the screen.

–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 => *$widget*

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

Note: due to restrictions in the ways that windows are managed, it is not possible to draw other graphical items (such as lines and images) on top of window items. A window item always obscures any graphics that overlap it, regardless of their order in the display list.

APPLICATION–DEFINED ITEM TYPES

It is possible for individual applications to define new item types for canvas widgets using C code. See the documentation for **Tk_CreateItemType**.

BINDINGS

Canvas has default bindings to allow scrolling if necessary: <Up, <Down, <Left and <Right (and their <Control–* counter parts). Further <Proir, <Next, <Home and <End. These bindings allow you to navigate the same way as in other widgets that can scroll.

CREDITS

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

KEYWORDS

canvas, widget

NAME

Tk::Checkbutton – Create and manipulate Checkbutton widgets

=for category Tk Widget Classes

SYNOPSIS

$checkbutton = *$parent*->**Checkbutton***(?options?);*

STANDARD OPTIONS

–activebackground –cursor –highlightthickness –takefocus –activeforeground –disabledforeground –image –text –anchor –font –justify –textvariable –background –foreground –padx –underline –bitmap –highlightbackground–pady –wraplength –borderwidth –highlightcolor –relief

See *Tk::options* for details of the standard options.

WIDGET–SPECIFIC OPTIONS

Name: **command**
Class: **Command**
Switch: **–command**

Specifies a *Tk callback\Tk::callbacks* 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.

Name: **height**
Class: **Height**
Switch: **–height**

Specifies a desired height for the button. If an image or bitmap is being displayed in the button then the value is in screen units (i.e. any of the forms acceptable to **Tk_GetPixels**); for text it is in lines of text. If this option isn‘t specified, the button‘s desired height is computed from the size of the image or bitmap or text being displayed in it.

Name: **indicatorOn**
Class: **IndicatorOn**
Switch: **–indicatoron**

Specifies whether or not the indicator should be drawn. Must be a proper boolean value. If false, the **relief** option is ignored and the widget‘s relief is always sunken if the widget is selected and raised otherwise.

Name: **offValue**
Class: **Value**
Switch: **–offvalue**

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

Name: **onValue**
Class: **Value**
Switch: **–onvalue**

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

Name: **selectColor**
Class: **Background**
Switch: **–selectcolor**

Specifies a background color to use when the button is selected. If **indicatorOn** is true then the color applies to the indicator. Under Windows, this color is used as the background for the indicator

regardless of the select state. If **indicatorOn** is false, this color is used as the background for the entire widget, in place of **background** or **activeBackground**, whenever the widget is selected. If specified as an empty string then no special color is used for displaying when the widget is selected.

Name: **selectImage**
Class: **SelectImage**
Switch: **–selectimage**

Specifies an image to display (in place of the **image** option) when the checkbutton is selected. This option is ignored unless the **image** option has been specified.

Name: **state**
Class: **State**
Switch: **–state**

Specifies one of three states for the checkbutton: **normal**, **active**, or **disabled**. In normal state the checkbutton is displayed using the **foreground** and **background** options. The active state is typically used when the pointer is over the checkbutton. In active state the checkbutton is displayed using the **activeForeground** and **activeBackground** options. Disabled state means that the checkbutton should be insensitive: the default bindings will refuse to activate the widget and will ignore mouse button presses. In this state the **disabledForeground** and **background** options determine how the checkbutton is displayed.

Name: **variable**
Class: **Variable**
Switch: **–variable**

Specifies reference to a variable to set to indicate whether or not this button is selected. Defaults to `\$widget->{'Value'}` member of the widget's hash. In general perl variables are `undef` unless specifically initialized which will not match either default **–onvalue** or default **–offvalue**.

Name: **width**
Class: **Width**
Switch: **–width**

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

DESCRIPTION

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

A checkbutton is a widget that displays a textual string, bitmap or image and a square called an *indicator*. If text is displayed, it must all be in a single font, but it can occupy multiple lines on the screen (if it contains newlines or if wrapping occurs because of the **wrapLength** option) and one of the characters may optionally be underlined using the **underline** option. A checkbutton has all of the behavior of a simple button, including the following: it can display itself in either of three different ways, according to the **state** option; it can be made to appear raised, sunken, or flat; it can be made to flash; and it invokes a *Tk callback|Tk::callbacks* whenever mouse button 1 is clicked over the checkbutton.

In addition, checkbuttons can be *selected*. If a checkbutton is selected then the indicator is normally drawn with a selected appearance, and a Tcl variable associated with the checkbutton is set to a particular value (normally 1). Under Unix, the indicator is drawn with a sunken relief and a special color. Under Windows, the indicator is drawn with a check mark inside. If the checkbutton is not selected, then the indicator is drawn with a deselected appearance, and the associated variable is set to a different value (typically 0). Under Unix, the indicator is drawn with a raised relief and no special color. Under Windows, the indicator is drawn

without a check mark inside. By default, the name of the variable associated with a checkbutton is the same as the *name* used to create the checkbutton. The variable name, and the ''on'' and ''off'' values stored in it, may be modified with options on the command line or in the option database. Configuration options may also be used to modify the way the indicator is displayed (or whether it is displayed at all). By default a checkbutton is configured to select and deselect itself on alternate button clicks. In addition, each checkbutton monitors its associated variable and automatically selects and deselects itself when the variables value changes to and from the button's ''on'' value.

WIDGET METHODS

The **Checkbutton** method creates a widget object. This object supports the **configure** and **cget** methods described in *Tk::options* which can be used to enquire and modify the options described above. The widget also inherits all the methods provided by the generic *Tk::Widget|Tk::Widget* class.

The following additional methods are available for checkbutton widgets:

$checkbutton->**deselect**

Deselects the checkbutton and sets the associated variable to its ''off'' value.

$checkbutton->**flash**

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

$checkbutton->**invoke**

Does just what would have happened if the user invoked the checkbutton with the mouse: toggle the selection state of the button and invoke the *Tk callback|Tk::callbacks* associated with the checkbutton, if there is one. The return value is the return value from the *Tk callback|Tk::callbacks*, or an empty string if there is no command associated with the checkbutton. This command is ignored if the checkbutton's state is **disabled**.

$checkbutton->**select**

Selects the checkbutton and sets the associated variable to its ''on'' value.

$checkbutton->**toggle**

Toggles the selection state of the button, redisplaying it and modifying its associated variable to reflect the new state.

BINDINGS

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

[1] On Unix systems, a checkbutton activates whenever the mouse passes over it and deactivates whenever the mouse leaves the checkbutton. On Mac and Windows systems, when mouse button 1 is pressed over a checkbutton, the button activates whenever the mouse pointer is inside the button, and deactivates whenever the mouse pointer leaves the button.

[2] When mouse button 1 is pressed over a checkbutton, it is invoked (its selection state toggles and the command associated with the button is invoked, if there is one).

[3] When a checkbutton has the input focus, the space key causes the checkbutton to be invoked. Under Windows, there are additional key bindings; plus (+) and equal (=) select the button, and minus (–) deselects the button.

If the checkbutton's state is **disabled** then none of the above actions occur: the checkbutton is completely non–responsive.

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

KEYWORDS

checkbutton, widget

NAME

chooseColor – pops up a dialog box for the user to select a color.

=for category Popups and Dialogs

SYNOPSIS

$color = *$widget*->**chooseColor***?(-option=>value, ...)?;*

DESCRIPTION

The method **chooseColor** is implemented as a perl wrapper on the core tk "command" **tk_chooseColor**. The *$widget* is passed as the argument to **-parent** described below. The implementation of internal **tk_chooseColor** is platform specific, on Win32 it is a native dialog, and on UNIX/X it is implemented in terms of *Tk::ColorEditor\Tk::ColorEditor*.

The core tk command **tk_chooseColor** pops up a dialog box for the user to select a color. The following *option-value* pairs are possible as command line arguments:

-initialcolor=>*color*

Specifies the color to display in the color dialog when it pops up. *color* must be in a form acceptable to the **Tk_GetColor** function.

-parent=>`$widget`

Makes `$widget` the logical parent of the color dialog. The color dialog is displayed on top of its parent window.

-title=>*titleString*

Specifies a string to display as the title of the dialog box. If this option is not specified, then a default title will be displayed.

If the user selects a color, **tk_chooseColor** will return the name of the color in a form acceptable to **Tk_GetColor**. If the user cancels the operation, the command will return **undef**.

EXAMPLE

```
$widget->configure(-fg => $parent->chooseColor(-initialcolor => 'gray',
                -title => "Choose color"));
```

KEYWORDS

color selection dialog

NAME

Tk::clipboard – Manipulate Tk clipboard

=for category User Interaction

SYNOPSIS

$widget->**clipboard***Option?(args)?*

DESCRIPTION

This command provides an interface to the Tk clipboard, which stores data for later retrieval using the selection mechanism. In order to copy data into the clipboard, **clipboardClear** must be called, followed by a sequence of one or more calls to **clipboardAppend**. To ensure that the clipboard is updated atomically, all appends should be completed before returning to the event loop.

The following methods are currently supported:

$widget->**clipboardClear**

Claims ownership of the clipboard on *$widget's* display and removes any previous contents. Returns an empty string.

$widget->**clipboardAppend***(?***–format***=>format?,?***–type***=>type?,?***—***?,data)*

Appends *data* to the clipboard on `$widget's` display in the form given by *type* with the representation given by *format* and claims ownership of the clipboard on `$widget's` display.

Type specifies the form in which the selection is to be returned (the desired ''target'' for conversion, in ICCCM terminology), and should be an atom name such as STRING or FILE_NAME; see the Inter–Client Communication Conventions Manual for complete details. *Type* defaults to STRING.

The *format* argument specifies the representation that should be used to transmit the selection to the requester (the second column of Table 2 of the ICCCM), and defaults to STRING. If *format* is STRING, the selection is transmitted as 8–bit ASCII characters. See the *Tk::Selection* documentation for explanation of what happens if *format* is not STRING. Note that arguments passed to **clipboardAppend** are concatenated before conversion, so the caller must take care to ensure appropriate spacing across string boundaries. All items appended to the clipboard with the same *type* must have the same *format*.

A **—** argument may be specified to mark the end of options: the next argument will always be used as *data*. This feature may be convenient if, for example, *data* starts with a **–**.

KEYWORDS

clear, format, clipboard, append, selection, type

NAME

Tk::CmdLine – Process standard X11 command line options and set initial resources

=for pm Tk/CmdLine.pm

=for category Creating and Configuring Widgets

SYNOPSIS

```
Tk::CmdLine::SetArguments([@argument]);

my $value = Tk::CmdLine::cget([$option]);

Tk::CmdLine::SetResources((\@resource | $resource) [, $priority]);

Tk::CmdLine::LoadResources(
    [ -symbol   => $symbol     ]
    [ -file     => $fileSpec   ]
    [ -priority => $priority   ]
    [ -echo     => $fileHandle ] );
```

DESCRIPTION

Process standard X11 command line options and set initial resources.

The X11R5 man page for X11 says: "Most X programs attempt to use the same names for command line options and arguments. All applications written with the X Toolkit Intrinsics automatically accept the following options: ...". This module processes these command line options for perl/Tk applications using the `SetArguments` function.

This module can optionally be used to load initial resources explicitly via function `SetResources`, or from specified files (default: the standard X11 application–specific resource files) via function `LoadResources`.

Command Line Options

–background *Color* | **–bg** *Color*

Specifies the color to be used for the window background.

–class *Class*

Specifies the class under which resources for the application should be found. This option is useful in shell aliases to distinguish between invocations of an application, without resorting to creating links to alter the executable file name.

–display *Display* | **–screen** *Display*

Specifies the name of the X server to be used.

–font *Font* | **–fn** *Font*

Specifies the font to be used for displaying text.

–foreground *Color* | **–fg** *Color*

Specifies the color to be used for text or graphics.

–geometry *Geometry*

Specifies the initial size and location of the *first MainWindow|Tk::MainWindow*.

–iconic

Indicates that the user would prefer that the application‘s windows initially not be visible as if the windows had been immediately iconified by the user. Window managers may choose not to honor the application‘s request.

–motif

Specifies that the application should 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.

–name *Name*

Specifies the name under which resources for the application should be found. This option is useful in shell aliases to distinguish between invocations of an application, without resorting to creating links to alter the executable file name.

–synchronous

Indicates that requests to the X server should be sent synchronously, instead of asynchronously. Since Xlib normally buffers requests to the server, errors do do not necessarily get reported immediately after they occur. This option turns off the buffering so that the application can be debugged. It should never be used with a working program.

–title *TitleString*

This option specifies the title to be used for this window. This information is sometimes used by a window manager to provide some sort of header identifying the window.

–xrm *ResourceString*

Specifies a resource pattern and value to override any defaults. It is also very useful for setting resources that do not have explicit command line arguments.

The *ResourceString* is of the form <*pattern*>:<*value*>, that is (the first) ':' is used to determine which part is pattern and which part is value. The (<*pattern*>, <*value*>) pair is entered into the options database with **optionAdd** (for each *MainWindow\|Tk::MainWindow* configured), with *interactive* priority.

Initial Resources

There are several mechanism for initializing the resource database to be used by an X11 application. Resources may be defined in a `$HOME/.Xdefaults` file, a system application defaults file (e.g. /usr/lib/X11/app–defaults/<**CLASS**>), or a user application defaults file (e.g. `$HOME/<CLASS>`). The Tk::CmdLine functionality for setting initial resources concerns itself with the latter two.

Resource files contain data lines of the form <*pattern*>:<*value*>. They may also contain blank lines and comment lines (denoted by a ! character as the first non–blank character). Refer to *option\|Tk::option* for a description of <*pattern*>:<*value*>.

System Application Defaults Files

System application defaults files may be specified via environment variable `$XFILESEARCHPATH` which, if set, contains a colon–separated list of file patterns.

User Application Defaults Files

User application defaults files may be specified via environment variables `$XUSERFILESEARCHPATH, $XAPPLRESDIR` or `$HOME`.

METHODS

SetArguments

Extract the X11 options contained in a specified array (@ARGV by default).

```
Tk::CmdLine::SetArguments([@argument])
```

The X11 options may be specified using a single dash – as per the X11 convention, or using two dashes — as per the POSIX standard (e.g. **–geometry** *100x100*, **–geometry** *100x100* or **–geometry=***100x100*). The options may be interspersed with other options or arguments. A — by itself terminates option processing.

By default, command line options are extracted from @ARGV the first time a MainWindow is created. The Tk::MainWindow constructor indirectly invokes `SetArguments` to do this.

cget

Get the value of a configuration option specified via `SetArguments`. (`cget` first invokes `SetArguments` if it has not already been invoked.)

```
Tk::CmdLine::cget([$option])
```

The valid options are: **–class**, **–name**, **–screen** and **–title**. If no option is specified, **–class** is implied.

A typical use of `cget` might be to obtain the application class in order to define the name of a resource file to be loaded in via `LoadResources`.

```
my $class = Tk::CmdLine::cget(); # process command line and return class
```

SetResources

Set the initial resources.

```
Tk::CmdLine::SetResources((\@resource | $resource) [, $priority])
```

A single resource may be specified using a string of the form '<*pattern*>:<*value*>'. Multiple resources may be specified by passing an array reference whose elements are either strings of the above form, and/or anonymous arrays of the form [<*pattern*>, <*value*>]. The optional second argument specifies the priority, as defined in *option\|Tk::option*, to be associated with the resources (default: *userDefault*).

Note that `SetResources` first invokes `SetArguments` if it has not already been invoked.

LoadResources

Load initial resources from one or more files.

```
Tk::CmdLine::LoadResources(
     [ -symbol   => $symbol     ]
     [ -file     => $fileSpec   ]
     [ -priority => $priority   ]
     [ -echo     => $fileHandle ] );
```

[**–symbol** => `$symbol`] specifies the name of an environment variable that, if set, defines a colon–separated list of one or more directories and/or file patterns. `$XUSERFILESEARCHPATH` is a special case. If `$XUSERFILESEARCHPATH` is not set, `$XAPPLRESDIR` is checked instead. If `$XAPPLRESDIR` is not set, `$HOME` is checked instead.

An item is identified as a file pattern if it contains one or more /%[A–Za–z]/ patterns. Only patterns **%L**, **%T** and **%N** are currently recognized. All others are replaced with the null string. Pattern **%L** is translated into `$LANG`. Pattern **%T** is translated into *app–defaults*. Pattern **%N** is translated into the application class name.

Each file pattern, after substitutions are applied, is assumed to define a FileSpec to be examined.

When a directory is specified, FileSpecs **<DIRECTORY>/<LANG>/<CLASS>** and **<DIRECTORY>/<CLASS>** are defined, in that order.

[**–file** => `$fileSpec`] specifies a resource file to be loaded in. The file is silently skipped if if does not exist, or if it is not readable.

[**–priority** => `$priority`] specifies the priority, as defined in *option\|Tk::option*, to be associated with the resources (default: *userDefault*).

[**–echo** => `$fileHandle`] may be used to specify that a line should be printed to the corresponding FileHandle (default: *STDOUT) everytime a file is examined / loaded.

If no **–symbol** or **–file** options are specified, `LoadResources` processes symbol `$XFILESEARCHPATH` with priority *startupFile* and `$XUSERFILESEARCHPATH` with priority

userDefault. (Note that `$XFILESEARCHPATH` and `$XUSERFILESEARCHPATH` are supposed to contain only patterns. `$XAPPLRESDIR` and `$HOME` are supposed to be a single directory. `LoadResources` does not check/care whether this is the case.)

For each set of FileSpecs, `LoadResources` examines each FileSpec to determine if the file exists and is readable. The first file that meets this criteria is read in and `SetResources` is invoked.

Note that `LoadResources` first invokes `SetArguments` if it has not already been invoked.

NOTES

This module is an object−oriented module whose methods can be invoked as object methods, class methods or regular functions. This is accomplished via an internally−maintained object reference which is created as necessary, and which always points to the last object used. `SetArguments`, `SetResources` and `LoadResources` return the object reference.

EXAMPLES

1 @ARGV is processed by Tk::CmdLine at MainWindow creation.

```
use Tk;

# <Process @ARGV - ignoring all X11-specific options>

my $mw = MainWindow->new();

MainLoop();
```

2 @ARGV is processed by Tk::CmdLine before MainWindow creation. An @ARGV of (—geometry=100x100 −opt1 a b c −bg red) is equal to (−opt1 a b c) after `SetArguments` is invoked.

```
use Tk;

Tk::CmdLine::SetArguments(); # Tk::CmdLine->SetArguments() works too

# <Process @ARGV - not worrying about X11-specific options>

my $mw = MainWindow->new();

MainLoop();
```

3 Just like 2) except that default arguments are loaded first.

```
use Tk;

Tk::CmdLine::SetArguments(qw(-name test -iconic));
Tk::CmdLine::SetArguments();

# <Process @ARGV - not worrying about X11-specific options>

my $mw = MainWindow->new();

MainLoop();
```

4 @ARGV is processed by Tk::CmdLine before MainWindow creation. Standard resource files are loaded in before MainWindow creation.

```
use Tk;

Tk::CmdLine::SetArguments();

# <Process @ARGV - not worrying about X11-specific options>

Tk::CmdLine::LoadResources();

my $mw = MainWindow->new();

MainLoop();
```

5 @ARGV is processed by Tk::CmdLine before MainWindow creation. Standard resource files are loaded in before MainWindow creation using non−default priorities.

```
use Tk;

Tk::CmdLine::SetArguments();

# <Process @ARGV - not worrying about X11-specific options>

Tk::CmdLine::LoadResources(-echo => \*STDOUT,
    -priority => 65, -symbol => 'XFILESEARCHPATH' );
Tk::CmdLine::LoadResources(-echo => \*STDOUT,
    -priority => 75, -symbol => 'XUSERFILESEARCHPATH' );

my $mw = MainWindow->new();

MainLoop();
```

6 @ARGV is processed by Tk::CmdLine before MainWindow creation. Standard resource files are loaded in before MainWindow creation. Individual resources are also loaded in before MainWindow creation.

```
use Tk;

Tk::CmdLine::SetArguments();

# <Process @ARGV - not worrying about X11-specific options>

Tk::CmdLine::LoadResources();

Tk::CmdLine::SetResources( # set a single resource
    '*Button*background: red',
    'widgetDefault' );

Tk::CmdLine::SetResources( # set multiple resources
    [ '*Button*background: red', '*Button*foreground: blue' ],
    'widgetDefault' );

my $mw = MainWindow->new();

MainLoop();
```

ENVIRONMENT

HOME (optional)

Home directory which may contain user application defaults files as $HOME/$LANG/<**CLASS**> or $HOME/<**CLASS**>.

LANG (optional)

The current language (default: *C*).

XFILESEARCHPATH (optional)

Colon−separated list of FileSpec patterns used in defining system application defaults files.

XUSERFILESEARCHPATH (optional)

Colon−separated list of FileSpec patterns used in defining user application defaults files.

XAPPLRESDIR (optional)

Directory containing user application defaults files as $XAPPLRESDIR/$LANG/<**CLASS**> or $XAPPLRESDIR/<**CLASS**>.

SEE ALSO

MainWindow\Tk::MainWindow option\Tk::option

HISTORY

- 1999.03.04 Ben Pavon <ben.pavon@hsc.hac.com>

  Rewritten as an object−oriented module.

  Allow one to process command line options in a specified array (@ARGV by default). Eliminate restrictions on the format and location of the options within the array (previously the X11 options could not be specified in POSIX format and had to be at the beginning of the array).

  Added the `SetResources` and `LoadResources` functions to allow the definition of resources prior to MainWindow creation.

NAME

Tk::ColorEditor – a general purpose Tk widget Color Editor

=for pm Tk/ColorEditor.pm

=for category Popups and Dialogs

SYNOPSIS

```
use Tk::ColorEditor;

$cref = $mw->ColorEditor(-title => $title, -cursor => @cursor);

$cref->Show;
```

DESCRIPTION

ColorEditor is implemented as an object with various methods, described below. First, create your ColorEditor object during program initialization (one should be sufficient), and then configure it by specifying a list of Tk widgets to colorize. When it's time to use the editor, invoke the `Show()` method.

ColorEditor allows some customization: you may alter the color attribute menu by adding and/or deleting menu items and/or separators, turn the status window on or off, alter the configurator's list of color widgets, or even supply your own custom color configurator callback.

1. Call the constructor to create the editor object, which in turn returns a blessed reference to the new object:

```
use Tk::ColorEditor;

$cref = $mw->ColorEditor(
    -title  => $title,
    -cursor => @cursor,
);
```

```
   mw     - a window reference, usually the result of a MainWindow->new
            call.  As the default root of a widget tree, $mw and all
            descendant widgets at object-creation-time are configured
            by the default color configurator procedure.  (You probably
            want to change this though or you might end up colorizing
            ColorEditor!)
   title  - Toplevel title, default = ' '.
   cursor - a valid Tk '-cursor' specification (default is
            'top_left_arrow').  This cursor is used over all ColorEditor
            "hot spots".
```

2. Invoke the `configure()` method to change editor characteristics:

```
$cref->configure(-option => value, ..., -option-n => value-n);
```

```
   options:
     -command             : a callback to a  'set_colors' replacement.
     -widgets             : a reference to a list of widget references
                            for the color configurator.
     -display_status      : TRUE IFF display the ColorEditor status
                            window when applying colors.
     -add_menu_item       : 'SEP', or a color attribute menu item.
     -delete_menu_item    : 'SEP', a color attribute menu item, or color
                            attribute menu ordinal.
```

For example:

```
   $cref->configure(-delete_menu_item   => 3,
```

```
            -delete_menu_item   => 'disabledforeground',
            -add_menu_item      => 'SEP',
            -add_menu_item      => 'New color attribute',
            -widgets            => [$ce, $qu, $f2b2],
            -widgets            => [$f2->Descendants],
            -command            => [\&my_special_configurator, some, args ]
        );
```

3. Invoke the `Show()` method on the editor object, say, by a button or menu press:

```
$cref->Show;
```

4. The cget(−widgets) method returns a reference to a list of widgets that are colorized by the configurator. Typically, you add new widgets to this list and then use it in a subsequent `configure()` call to expand your color list.

```
$cref->configure(
    -widgets => [
        @{$Filesystem_ref->cget(-widgets)}, @{$cref->cget(-widgets)},
    ]
);
```

5. The `delete_widgets()` method expects a reference to a list of widgets which are then removed from the current color list.

```
$cref->delete_widgets($OBJTABLE{$objname}->{'-widgets'})
```

AUTHORS

Stephen O. Lidie, Lehigh University Computing Center. 95/03/05 lusol@Lehigh.EDU

Many thanks to Guy Decoux (decoux@moulon.inra.fr) for doing the initial translation of tcolor.tcl to TkPerl, from which this code has been derived.

NAME

Tk::DragDrop::Common − private class used by Drag&Drop

=for pm DragDrop/DragDrop/Common.pm

=for category Experimental Modules

DESCRIPTION

This class provides methods to automate the the loading and declaring of Drop and Site ‘types’.

NAME

Tk::composite – Defining a new composite widget class

=for category Derived Widgets

SYNOPSIS

```
package Tk::Whatever;

require Tk::Derived;
require Tk::Frame;                      # or Tk::Toplevel
@ISA = qw(Tk::Derived Tk::Frame)';      # or Tk::Toplevel

Construct Tk::Widget 'Whatever';

sub ClassInit
{
 my ($class,$mw) = @_;

 #... e.g., class bindings here ...
 $class->SUPER::ClassInit($mw);
}

sub Populate
{
 my ($cw,$args) = @_;

 my $flag = delete $args->{-flag};
 if (defined $flag)
  {
   # handle -flag => xxx which can only be done at create
   # time the delete above ensures that new() does not try
   # and do  $cw->configure(-flag => xxx);
  }

 $cw->SUPER::Populate($args);

 $w = $cw->Component(...);

 $cw->Delegates(...);

 $cw->ConfigSpecs(
            '-cursor'    => [SELF,'cursor','Cursor',undef],
            '-something' => [METHOD,dbName,dbClass,'default'],
            '-text'      => [$label,dbName,dbClass,'default'],
            '-heading'   => [{-text=>$head},
                             heading,Heading,'My Heading'],
            );
}

sub something
{
 my ($cw,$value) = @_;
 if (@_ > 1)
  {
   # set it
  }
 return # current value
}

1;
```

```
__END__

# Anything not documented is *private* - your POD is god, so to speak.

=head1 NAME

Tk::Whatever - a whatever widget

=head1 SYNOPSIS

  use Tk::Whatever;

  $widget = $parent->Whatever(...);

=head1 DESCRIPTION

You forgot to document your widget, didn't you? :-)

...
```

DESCRIPTION

The intention behind a composite is to create a higher−level widget, sometimes called a "super−widget" or "meta−widget". Most often, a composite will be built upon other widgets by **using** them, as opposed to specializing on them. For example, the supplied composite widget **LabEntry** is *made of* an **Entry** and a **Label**; it is neither a *kind−of* **Label** nor is it a *kind−of* **Entry**.

Most of the work of a composite widget consist in creating subwidgets, arrange to dispatch configure options to the proper subwidgets and manage composite−specific configure options.

GLORY DETAILS

Depending on your perl/Tk knowledget this section may be enlighting or confusing.

Composite Widget

Since perl/Tk is heavilly using an object−oriented approach, it is no suprise that creating a composite goes through a **`new()`** method. However, the composite does not normally define a **`new()`** method itself: it is usually sufficient to simply inherit it from **Tk::Widget**.

This is what happens when the composite use

```
@ISA = qw(Tk::Frame);  # or Tk::Toplevel
```

to specify its inheritance chain. To complete the initialisation of the widget, it must call the **Construct** method from class **Widget**. That method accepts the name of the new class to create, i.e. the package name of your composite widget:

```
Construct Tk::Widget 'Whatever';
```

Here, **Whatever** is the package name (aka the widget's **class**). This will define a constructor method for **Whatever**, normally named after the widget's class. Instanciating that composite in client code would the look like:

```
$mw = MainWindow->new();    # Creates a top-level main window

$cw = $mw->Whatever();      # Creates an instance of the
                            # composite widget Whatever
```

Whenever a composite is instanciated in client code, `Tk::Widget::new()` will be invoked via the widget's class constructor. That **new** method will call

```
$cw->InitObject(\%args);
```

where *%args* is the arguments passed to the widget's constructor. Note that **InitObject** receives a **reference** to the hash array containing all arguments.

For composite widgets that needs an underlying frame, **InitObject** will typically be inherited from **Tk::Frame**, that is, no method of this name will appear in the composite package. For composites that don't

need a frame, **InitObject** will typically be defined in the composite class (package). Compare the **LabEntry** composite with **Optionmenu**: the former is **Frame** based while the latter is **Widget** based.

In **Frame** based composites, **`Tk::Frame::InitObject()`** will call **`Populate()`**, which should be defined to create the characteristic subwidgets of the class.

Widget based composites don't need an extra **Populate** layer; they typically have their own **InitObject** method that will create subwidgets.

Creating Subwidgets

Subwidget creation happens usually in **`Populate()`** (**Frame** based) or **`InitObject()`** (**Widget** based). The composite usually calls the subwidget's constructor method either directly, for "private" subwidgets, or indirectly through the **Component** method for subwidgets that should be advertised to clients.

Populate may call **Delegates** to direct calls to methods of chosen subwidgets. For simple composites, typically most if not all methods are directed to a single subwidget – e.g. **ScrListbox** directs all methods to the core **Listbox** so that *`$composite`*`->`**get***(...)* calls *`$listbox`*`->`**get***(...)*.

Further steps for Frame based composites

Populate should also call **`ConfigSpecs()`** to specify the way that configure–like options should be handled in the composite. Once **Populate** returns, method **Tk::Frame::ConfigDefault** walks through the **ConfigSpecs** entries and populates `%$args` hash with defaults for options from X resources (***.Xdefaults***, etc).

When **`InitObject()`** returns to **`Tk::Widget::new()`**, a call to **`$cw->`***configure*(**`%$args`**) is made which sets *all* the options.

SEE ALSO

Tk::ConfigSpecs|Tk::ConfigSpecs Tk::Derived|Tk::Derived

NAME

Tk::Compound − Create multi−line compound images.

=for category Tk Image Classes

use Tk::Compound; *$image* = *$widget*−>**Compound***?(name??,options?)*
$image−>**Line***?(options?)* *$image*−>**Text***?(options?)* *$image*−>**Bitmap***?(options?)*
$image−>**Image***?(options?)* *$image*−>**Space***?(options?)*

DESCRIPTION

Compound image types can be used to create images that consists of multiple horizontal lines; each line is composed of a series of items (texts, bitmaps, images or spaces) arranged from left to right. Compound images are mainly used to embed complex drawings into widgets that support the **−image** option. As shown in the EXAMPLE section below, a compound image can be used to display a bitmap and a text string simutaneously in a Tk **Button** widget.

Compound images can only be used on windows on the same display as, and with the same pixel depth and visual as the *$widget* used to create them.

CREATING COMPOUND IMAGES

Compounds are created using *$widget*−>**Compound**. Compounds support the following *options*:

−background => *color*

Specifies the background color of the compound image. This color is also used as the default background color for the bitmap items in the compound image.

−borderwidth => *pixels*

Specifies a non−negative value indicating the width of the 3−D border drawn around the compound image.

−font => *font*

Specifies the default font for the text items in the compound image.

−foreground => *color*

Specifies the default foreground color for the bitmap and text items in the compound image.

−padx => *value*

Specifies a non−negative value indicating how much extra space to request for the compound image in the X−direction. The *value* may have any of the forms acceptable to **Tk_GetPixels(3)**.

−pady => *value*

Specifies a non−negative value indicating how much extra space to request for the compound image in the Y−direction.

−relief => *value*

Specifies the 3−D effect desired for the background of the compound image. Acceptable values are **raised**, **sunken**, **flat**, **ridge**, and **groove**.

−showbackground => *value*

Specifies whether the background and the 3D borders should be drawn. Must be a valid boolean value. By default the background is not drawn and the compound image appears to have a transparent background.

IMAGE COMMAND

When a compound image is created, Tk also creates a new object. This object supports the **configure** and **cget** methods described in *Tk::options* which can be used to enquire and modify the options described above.

The object also supports the following methods:

$compound–>**Line***?(option =* **value ...)?**

Creates a new line at the bottom of the compound image. Lines support the following *options*:

–anchor value

Specifies how the line should be aligned along the horizontal axis. When the values are **w**, **sw** or **nw**, the line is aligned to the left. When the values are **c**, **s** or **n**, the line is aligned to the middle. When the values are **e**, **se** or **ne**, the line is aligned to the right.

–padx => *value*

Specifies a non–negative value indicating how much extra space to request for this line in the X–direction.

$compound–>*Itemtype?(option =* value ...)?

Creates a new item of the type *Itemtype* at the end of the last line of the compound image. All types of items support these following common *options*:

–anchor value

Specifies how the item should be aligned along the vertical axis. When the values are **n**, **nw** or **ne**, the item is aligned to the top of the line. When the values are **c**, **w** or **e**, the item is aligned to the middle of the line. When the values are **s**, **se** or **sw**, the item is aligned to the bottom of the line.

–padx => *value*

Specifies a non–negative value indicating how much extra space to request for this item in the X–direction.

–pady => *value*

Specifies a non–negative value indicating how much extra space to request for this item in the Y–direction.

item–type can be any of the following:

$compound–>**Bitmap***?(option =* value ...)?

Creates a new bitmap item of at the end of the last line of the compound image. Additional *options* accepted by the bitmap type are:

–background => *color*

Specifies the background color of the bitmap item.

–bitmap => *name*

Specifies a bitmap to display in this item, in any of the forms acceptable to **Tk_GetBitmap(3)**.

–foreground => *color*

Specifies the foreground color of the bitmap item.

$compound–>**Image***?(option =* value ...)?

Creates a new image item of at the end of the last line of the compound image. Additional *options* accepted by the image type are:

–image => *name*

Specifies an image to display in this item. *name* must have been created with the **image create** command.

$compound–>**Space***?(option =* value ...)?

Creates a new space item of at the end of the last line of the compound image. Space items do not display anything. They just acts as space holders that add additional spaces between items inside a compound image. Additional *options* accepted by the image type are:

–width => *value*

Specifies the width of this space. The *value* may have any of the forms acceptable to **Tk_GetPixels(3)**.

–height => *value*

Specifies the height of this space. The *value* may have any of the forms acceptable to **Tk_GetPixels(3)**.

$compound–>**Text***?(option* = value ...)?

Creates a new text item of at the end of the last line of the compound image. Additional *options* accepted by the text type are:

–background => *color*

Specifies the background color of the text item.

–font => *name*

Specifies the font to be used for this text item.

–foreground => *color*

Specifies the foreground color of the text item.

–justify *value*

When there are multiple lines of text displayed in a text item, this option determines how the lines line up with each other. *value* must be one of **left**, **center**, or **right**. **Left** means that the lines' left edges all line up, **center** means that the lines' centers are aligned, and **right** means that the lines' right edges line up.

–text => *string*

Specifies a text string to display in this text item.

–underline *value*

Specifies the integer index of a character to underline in the text item. 0 corresponds to the first character of the text displayed in the text item, 1 to the next character, and so on.

–wraplength *value*

This option specifies the maximum line length of the label string on this text item. If the line length of the label string exceeds this length, it is wrapped onto the next line, so that no line is longer than the specified length. The value may be specified in any of the standard forms for screen distances. If this value is less than or equal to 0 then no wrapping is done: lines will break only at newline characters in the text.

EXAMPLE

The following example creates a compound image with a bitmap and a text string and places this image into a **Button(n)** widget. Notice that the image must be created using the widget that it resides in.

```
my $b = $parent->Button;
my $c = $b->Compound;
$b->configure(-image => $c);
$c->Line;
$c->Bitmap(-bitmap => 'warning');
$c->Space(-width => 8);
$c->Text(-text => "Warning", -underline => 0);
$b->pack;
```

KEYWORDS

image(n), Tix(n)

NAME

Tk::ConfigSpecs – Defining behaviour of 'configure' for composite widgets.

=for category Derived Widgets

SYNOPSIS

```
sub Populate
{
 my ($composite,$args) = @_;
 ...
 $composite->ConfigSpecs('-attribute' => [ where,dbName,dbClass,default ]);
 $composite->ConfigSpecs('-alias' => '-otherattribute');
 $composite->ConfigSpecs('DEFAULT' => [ where ]);
 ...
}

$composite->configure(-attribute => value);
```

DESCRIPTION

The aim is to make the composite widget configure method look as much like a regular Tk widget's configure as possible. (See *Tk::options* for a description of this behaviour.) To enable this the attributes that the composite as a whole accepts needs to be defined.

Defining the ConfigSpecs for a class.

Typically a widget will have one or more calls like the following

```
$composite->ConfigSpecs(-attribute => [where,dbName,dbClass,default]);
```

in its **Populate** method. When **ConfigSpecs** is called this way (with arguments) the arguments are used to construct or augment/replace a hash table for the widget. (More than one *−option=>value* pair can be specified to a single call.)

dbName, **dbClass** and default are only used by **ConfigDefault** described below, or to respond to 'inquiry' configure commands.

It may be either one of the values below, or a list of such values enclosed in **[]**.

The currently permitted values of **where** are:

'ADVERTISED'

apply **configure** to *advertised* subwidgets.

'DESCENDANTS'

apply **configure** recursively to all descendants.

'CALLBACK'

Setting the attribute does `Tk::Callback->new($value)` before storing in `$composite->{Configure}{-attribute}`. This is appropriate for `-command => ...` attributes that are handled by the composite and not forwarded to a subwidget. (E.g. **Tk::Tiler** has `-yscrollcommand` to allow it to have scrollbar attached.)

This may be the first of several 'validating' keywords (e.g. font, cursor, anchor etc.) that core Tk makes special for C code.

'CHILDREN'

apply **configure** to all children. (Children are the immediate descendants of a widget.)

'METHOD'

Call `$cw->attribute(value)`

This is the most general case. Simply have a method of the composite class with the same name as the attribute. The method may do any validation and have whatever side–effects you like. (It is probably worth 'queueing' using **afterIdle** for more complex side–effects.)

'PASSIVE'

Simply store value in `$composite->{Configure}{-attribute}`.

This form is also a useful placeholder for attributes which you currently only handle at create time.

'SELF'

Apply **configure** to the core widget (e.g. **Frame**) that is the basis of the composite. (This is the default behaviour for most attributes which makes a simple Frame behave the way you would expect.) Note that once you have specified **ConfigSpecs** for an attribute you must explicitly include `'SELF'` in the list if you want the attribute to apply to the composite itself (this avoids nasty infinite recursion problems).

`$reference` (blessed)

Call **`$reference`**–configure(–attribute = value)

A common case is where **`$reference`** is a subwidget.

`$reference` may also be result of

```
Tk::Config->new(setmethod,getmethod,args,...);
```

Tk::Config class is used to implement all the above keyword types. The class has `configure` and `cget` methods so allows higher level code to *always* just call one of those methods on an *object* of some kind.

hash reference

Defining:

```
$cw->ConfigSpecs(
        ...
        -option => [ { -optionX=>$w1, -optionY=>[$w2, $w3] },
                       dbname dbclass default ],
        ...
        );
```

So `$cw->configure(-option => value)` actually does

```
$w1->configure(-optionX => value);
$w2->configure(-optionY => value);
$w3->configure(-optionY => value);
```

'otherstring'

Call

```
$composite->Subwidget('otherstring')->configure( -attribute => value );
```

While this is here for backward compatibility with Tk–b5, it is probably better just to use the subwidget reference directly. The only case for retaining this form is to allow an additional layer of abstraction – perhaps having a 'current' subwidget – this is unproven.

Aliases

`ConfigSpecs( -alias => '-otherattribute' )` is used to make `-alias` equivalent to `-otherattribute`. For example the aliases

```
-fg => '-foreground',
-bg => '-background'
```

are provided automatically (if not already specified).

Default Values

When the **Populate** method returns **ConfigDefault** is called. This calls

```
$composite->ConfigSpecs;
```

(with no arguments) to return a reference to a hash. Entries in the hash take the form:

```
'-attribute' => [ where, dbName, dbClass, default ]
```

ConfigDefault ignores 'where' completely (and also the DEFAULT entry) and checks the 'options' database on the widget's behalf, and if an entry is present matching dbName/dbClass

```
-attribute => value
```

is added to the list of options that **new** will eventually apply to the widget. Likewise if there is not a match and default is defined this default value will be added.

Alias entries in the hash are used to convert user-specified values for the alias into values for the real attribute.

`New()`-time Configure

Once control returns to **new**, the list of user-supplied options augmented by those from **ConfigDefault** are applied to the widget using the **configure** method below.

Widgets are most flexible and most Tk-like if they handle the majority of their attributes this way.

Configuring composites

Once the above have occurred calls of the form:

```
$composite->configure( -attribute => value );
```

should behave like any other widget as far as end-user code is concerned. **configure** will be handled by **Tk::Derived::configure** as follows:

```
$composite->ConfigSpecs;
```

is called (with no arguments) to return a reference to a hash **-attribute** is looked up in this hash, if **-attribute** is not present in the hash then **'DEFAULT'** is looked for instead. (Aliases are tried as well and cause redirection to the aliased attribute). The result should be a reference to a list like:

```
[ where, dbName, dbClass, default ]
```

at this stage only *where* is of interest, it maps to a list of object references (maybe only one) foreach one

```
$object->configure( -attribute => value );
```

is **eval**ed.

Inquiring attributes of composites

```
$composite->cget( '-attribute' );
```

This is handled by **Tk::Derived::cget** in a similar manner to configure. At present if *where* is a list of more than one object it is ignored completely and the "cached" value in

```
$composite->{Configure}{-attribute}.
```

is returned.

CAVEATS

It is the author's intention to port as many of the "Tix" composite widgets as make sense. The mechanism described above may have to evolve in order to make this possible, although now aliases are handled I think the above is sufficient.

SEE ALSO

Tk::composite|Tk::composite, *Tk::options|Tk::options*

NAME

Tk::Derived – Base class for widgets derived from others

=for pm Tk/Derived.pm

=for category Derived Widgets

SYNOPSIS

```
package Tk::Whatever;
require Tk::Something;
require Tk::Derived;

@ISA = qw(Tk::Derived Tk::Something);

sub Populate
{
 my ($cw,$args) = @_;
 ...
 $cw->SUPER::Populate($args);
 $cw->ConfigSpecs(...);
 ...
}
```

DESCRIPTION

Tk::Derived is used with perl5's multiple inheritance to override some methods normally inherited from Tk::Widget.

Tk::Derived should precede any Tk widgets in the class's @ISA.

Tk::Derived's main purpose is to apply wrappers to `configure` and `cget` methods of widgets to allow the derived widget to add to or modify behaviour of the configure options supported by the base widget.

The derived class should normally override the `Populate` method provided by Tk::Derived and call `ConfigSpecs` to declare configure options.

The public methods provided by Tk::Derived are as follows:

–>ConfigSpecs(–*key* => [*kind*, *name*, *Class*, *default*], ...)

SEE ALSO

Tk::ConfigSpecs|Tk::ConfigSpecs Tk::mega|Tk::mega

NAME

Tk::Dialog − Create modal dialog and wait for a response.

=for pm Tk/Dialog.pm

=for category Popups and Dialogs

SYNOPSIS

$dialog = *$parent*−>**Dialog**(−*option* => *value, ...);*

DESCRIPTION

This procedure is part of the Tk script library − its arguments describe a dialog box. After creating a dialog box, **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 text string of the selected button.

While waiting for the user to respond, **Dialog** sets a local grab. This prevents the user from interacting with the application in any way except to invoke the dialog box. See **Show()** method.

The following option/value pairs are supported:

−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_button

Text label string of the button that displays the default ring.

−buttons

A reference to a list of button label strings. Each *string* specifies text to display in a button, in order from left to right.

METHODS

```
$answer = $dialog->Show(?-global?);
```

This method displays the dialog, waits for the user‘s response, and stores the text string of the selected button in `$answer`. If −*global* is specified a global (rather than local) grab is performed.

EXAMPLE

$dialog = *$mw*−>**Dialog**(−*text* => ‘Save File?‘, −bitmap => ‘question‘, −title => ‘Save File Dialog‘, −default_button => ‘Yes‘, −buttons => [qw/Yes No Cancel/);

KEYWORDS

bitmap, dialog, modal, messageBox

NAME

Tk::DialogBox – create and manipulate a dialog screen.

=for pm Tixish/DialogBox.pm

=for category Tix Extensions

SYNOPSIS

```
use Tk::DialogBox
...
$d = $top->DialogBox(-title => "Title", -buttons => ["OK", "Cancel"]);
$w = $d->add(Widget, args);
...
$button = $d->Show;
```

DESCRIPTION

DialogBox is very similar to **Dialog** except that it allows any widget in the top frame. **DialogBox** creates two frames—–"top" and "bottom". The bottom frame shows all the specified buttons, lined up from left to right. The top frame acts as a container for all other widgets that can be added with the **`add()`** method. The non–standard options recognized by **DialogBox** are as follows:

–title

Specify the title of the dialog box. If this is not set, then the name of the program is used.

–buttons

The buttons to display in the bottom frame. This is a reference to an array of strings containing the text to put on each button. There is no default value for this. If you do not specify any buttons, no buttons will be displayed.

–default_button

Specifies the default button that is considered invoked when user presses <Return on the dialog box. This button is highlighted. If no default button is specified, then the first element of the array whose reference is passed to the **–buttons** option is used as the default.

METHODS

DialogBox supports only two methods as of now:

add(*widget*, *options*)

Add the widget indicated by *widget*. *Widget* can be the name of any Tk widget (standard or contributed). *Options* are the options that the widget accepts. The widget is advertized as a subwidget of **DialogBox**.

Show(*grab*)

Display the dialog box, until user invokes one of the buttons in the bottom frame. If the grab type is specified in *grab*, then **Show** uses that grab; otherwise it uses a local grab. Returns the name of the button invoked.

BUGS

There is no way of removing a widget once it has been added to the top frame.

There is no control over the appearance of the buttons in the bottom frame nor is there any way to control the placement of the two frames with respect to each other e.g. widgets to the left, buttons to the right instead of widgets on the top and buttons on the bottom always.

AUTHOR

Rajappa Iyer rsi@earthling.net

NAME

Tk::DirTree – Create and manipulate DirTree widgets

=for pm Tixish/DirTree.pm

=for category Tix Extensions

SYNOPSIS

use Tk::DirTree;

$dirtree = *$parent*->**DirTree**(*?options?*);

SUPER–CLASS

The **DirTree** class is derived from the *Tree|Tk::Tree* class and inherits all the methods, options and subwidgets of its super–class.

STANDARD OPTIONS

Tree supports all the standard options of a Tree widget. See *Tk::options* for details on the standard options.

WIDGET–SPECIFIC OPTIONS

Name: **browseCmd**
Class: **BrowseCmd**
Switch: **–browsecmd**

Specifies a *callback|Tk::callbacks* to call whenever the user browses on a directory (usually by single–clicking on the name of the directory). The callback is called with one argument, the complete pathname of the directory.

Name: **command**
Class: **Command**
Switch: **–command**

Specifies the *callback|Tk::callbacks* to be called when the user activates on a directory (usually by double–clicking on the name of the directory). The callback is called with one argument, the complete pathname of the directory.

Name: **dircmd**
Class: **DirCmd**
Switch: **–dircmd**

Specifies the *callback|Tk::callbacks* to be called when a directory listing is needed for a particular directory. If this option is not specified, by default the DirTree widget will attempt to read the directory as a Unix directory. On special occasions, the application programmer may want to supply a special method for reading directories: for example, when he needs to list remote directories. In this case, the **–dircmd** option can be used. The specified callback accepts two arguments: the first is the name of the directory to be listed; the second is a Boolean value indicating whether hidden sub–directories should be listed. This callback returns a list of names of the sub–directories of this directory. For example:

```
sub read_dir {
    my( $dir, $showhidden ) = @_;
    return( qw/DOS NORTON WINDOWS/ ) if $dir eq "C:\\";
    return();
}
```

Name: **showHidden**
Class: **ShowHidden**
Switch: **–showhidden**

Specifies whether hidden directories should be shown. By default, a directory name starting with a period "." is considered as a hidden directory. This rule can be overridden by supplying an alternative **–dircmd** option.

Name: **directory**
Class: **Directory**
Switch: **–directory**
Alias: **–value**

Specifies the name of the current directory to be displayed in the DirTree widget.

DESCRIPTION

The **DirTree** constructor method creates a new window (given by the `$dirtree` argument) and makes it into a DirTree widget. Additional options, described above, may be specified on the command line or in the option database to configure aspects of the DirTree such as its cursor and relief. The DirTree widget displays a list view of a directory, its previous directories and its sub–directories. The user can choose one of the directories displayed in the list or change to another directory.

WIDGET METHODS

The **DirTree** method creates a widget object. This object supports the **configure** and **cget** methods described in *Tk::options* which can be used to enquire and modify the options described above. The widget also inherits all the methods provided by the generic *Tk::Widget\Tk::Widget* class.

The following additional methods are available for DirTree widgets:

*`$dirtree`–>***chdir***(dir)***

Change the current directory to *dir*.

BINDINGS

The mouse and keyboard bindings of the DirTree widget are the same as the bindings of the *Tree\Tk::Tree* widget.

KEYWORDS

directory, tree, tix

SEE ALSO

Tk::Tree\Tk::Tree Tk::HList\Tk::HList

AUTHOR

Perl/TK version by Chris Dean <ctdean@cogit.com. Original Tcl/Tix version by Ioi Kim Lam.

NAME

Tk::DItem – Tix Display Items

=for category Tix Extensions

SYNOPSIS

DESCRIPTION

The Tix **Display Items** and **Display Types** are devised to solve a general problem: many Tix widgets (both existing and planned ones) display many items of many types simultaneously.

For example, a hierarchical listbox widget (see *Tk::HList*) can display items of images, plain text and subwindows in the form of a hierarchy. Another widget, the tabular listbox widget (see *Tk::TList*) also displays items of the same types, although it arranges the items in a tabular form. Yet another widget, the spreadsheet widget (see *Tk::TixGrid*), also displays similar types items, but in yet another format.

In these examples, the display items in different widgets are only different in how they are arranged by the **host widget**. In Tix, display items are clearly separated from the host widgets. The advantage is two–fold: first, the creation and configuration of display items become uniform across different host widgets. Second, new display item types can be added without the need to modify the existing host widgets.

In a way, Tix display items are similar to the items inside Tk the canvas widget. However, unlike the Tix display items, the canvas items are not independent of the canvas widget; this makes it impossible to use the canvas items inside other types of TK widgets.

The appearance of a display item is controlled by a set of *attributes*. It is observed that each the attributes usually fall into one of two categroies: ‘‘*individual*’’ or ‘‘*collective*’’. For example, the text items inside a HList widget may all display a different text string; however, in most cases, the text items share the same color, font and spacing. Instead of keeping a duplicated version of the same attributes inside each display item, it will be advantageous to put the collective attributes in a special object called a **display style**. First, there is the space concern: a host widget may have many thousands of items; keeping dupilcated attributes will be very wasteful. Second, when it becomes necessary to change a collective attribute, such as changing all the text items’ foreground color to red, it will be more efficient to change only the display style object than to modify all the text items one by one.

The attributes of the a display item are thus stored in two places: it has a set of **item options** to store its individual attributes. Each display item is also associated with a *display style*, which specifies the collective attributes of all items associated with itself.

The division between the individual and collective attributes are fixed and cannot be changed. Thus, when it becomes necessary for some items to differ in their collective attributes, two or more **display styles** can be used. For example, suppose you want to display two columns of text items inside an HList widget, one column in red and the other in blue. You can create a TextStyle object called ‘‘`$red`’’ which defines a red foreground, and another called ‘‘`$blue`’’, which defines a blue foreground. You can then associate all text items of the first column to ‘‘`$red`’’ and the second column to ‘‘`$blue`’’

DISPLAY ITEM TYPES AND OPTIONS

Currently there are three types of display items: **text**, **imagetext** and *window*.

IMAGETEXT ITEMS

Display items of the type **imagetext** are used to display an image together with a text string. Imagetext items support the following options:

Imagetext Item Options

Name: **bitmap**
Class: **Bitmap**
Switch: **–bitmap**

Specifies the bitmap to display in the item.

Name: **image**
Class: **Image**
Switch: **–image**

Specifies the image to display in the item. When both the **–bitmap** and **–image** options are specified, only the image will be displayed.

Name: **imageTextStyle**
Class: **ImageTextStyle**
Switch: **–style**

Specifies the display style to use for this item. Must be the name of a **imagetext** display style that has already be created with **ItemStyle**.

Name: **showImage**
Class: **ShowImage**
Switch: **–showimage**

A Boolean value that specifies whether the image/bitmap should be displayed.

Name: **showText**
Class: **ShowText**
Switch: **–showtext**

A Boolean value that specifies whether the text string should be displayed.

Name: **text**
Class: **Text**
Switch: **–text**

Specifies the text string to display in the item.

Name: **underline**
Class: **Underline**
Switch: **–underline**

Specifies the integer index of a character to underline in the text string in the item. 0 corresponds to the first character of the text displayed in the widget, 1 to the next character, and so on.

Imagetext Style Options

The style information of **imagetext** items are stored in the **imagetext** display style. The following options are supported:

STANDARD OPTIONS

–activebackground –activeforeground –anchor –background –disabledbackground –disabledforeground –foreground –font –justify –padx –pady –selectbackground –selectforeground –wraplength

See *Tk::options* for details of the standard options.

STYLE–SPECIFIC OPTIONS

Name: **gap**
Class: **Gap**
Switch: **–gap**

Specifies the distance between the bitmap/image and the text string, in number of pixels.

Name: **textAnchor**
Class: **TextAnchor**
Switch: **–textanchor**

The anchor position on the image to which text part is attached. This is a perl/Tk addition. Defaults to **e** for compatibility with standard Tix. The interesting cases are

n Text is centred above the image.

s Text is centred below the image

e Text is centred to right of the image.

w Text is centred to left of the image.

c Text is centred over the image.

The **sw**, **se**, **ne**, and b<nw cases look rather odd.

To get items to line up correctly it will usually be necessary to specify **–anchor** as well. e.g. with default **e** then anchoring item as a whole **w** lines images up down left with text stuck to right side.

TEXT ITEMS

Display items of the type **text** are used to display a text string in a widget. Text items support the following options:

Text Item Options

Name: **textStyle**
Class: **TextStyle**
Switch: **–style**

Specifies the display style to use for this text item. Must be the name of a **text** display style that has already be created with **ItemStyle**.

Name: **text**
Class: **Text**
Switch: **–text**

Specifies the text string to display in the item.

Name: **underline**
Class: **Underline**
Switch: **–underline**

Specifies the integer index of a character to underline in the item. 0 corresponds to the first character of the text displayed in the widget, 1 to the next character, and so on.

Text Style Options

STANDARD OPTIONS

–activebackground –activeforeground –anchor –background –disabledbackground –disabledforeground –foreground –font –justify –padx –pady –selectbackground –selectforeground –wraplength

See *Tk::options* for details of the standard options.

WINDOW ITEMS

Display items of the type *window* are used to display a sub–window in a widget. **Window** items support the following options:

Window Item Options

Name: **windowStyle**
Class: **WindowStyle**
Switch: **–style**

Specifies the display style to use for this window item. Must be the name of a *window* display style that has already be created with the **ItemStyle** method.

Name: **window**
Class: **Window**
Switch: **–window**
Alias: **–widget**

Specifies the sub–window to display in the item.

Window Style Options

STYLE STANDARD OPTIONS

–anchor **–padx** **–pady**

See *Tk::options* for details of the standard options.

CREATING DISPLAY ITEMS

Display items do not exist on their and thus they cannot be created independently of the widgets they reside in. As a rule, display items are created by special methods of their ‘‘host‘‘ widgets. For example, the HList widgets has a method **item** which can be used to create new display items. The following code creates a new text item at the third column of the entry foo inside an HList widget:

```
my $hlist = $parent->HList(-columns=>3);
$hlist->add('foo');
$hlist->itemCreate('foo', 2, -itemtype=>'text', -text=>'Hello');
```

The **itemCreate** method of the HList widget accepts a variable number of arguments. The special argument **–itemtype** specifies which type of display item to create. Options that are valid for this type of display items can then be specified by one or more *option–value* pairs.

After the display item is created, they can then be configured or destroyed using the methods provided by the host widget. For example, the HList widget has the methods **itemConfigure**, **itemCget** and **itemDelete** for accessing the display items.

CREATING AND MANIPULATING ITEM STYLES

Item styles are created with **ItemStyle**:

SYNOPSIS

gt
*$widget–>***ItemStyle***(itemType ?,***–stylename***=>name? ?,***–refwindow***=>pathName? ?,option=>value, ...?);*

itemType must be one of the existing display items types such as **text**, **imagetext**, **window** or any new types added by the user. Additional arguments can be given in one or more *option–value* pairs. *option* can be any of the valid option for this display style or any of the following:

–stylename => *name*

Specifies a name for this style. If unspecified, then a default name will be chosen for this style.

–refwindow => *$otherwidget*

Specifies a window to use for determine the default values of the display type. If unspecified, the *$widget* will be used. Default values for the display types can be set via the options database. The following example sets the **–disablebackground** and **–disabledforeground** options of a **text** display style via the option database:

```
$widget->optionAdd('*table.list*disabledForeground' => 'blue');
$widget->optionAdd('*table.list*disabledBackground' => 'darkgray');
$widget->ItemStyle('text', -refwindow => $table_list, -fg => 'red');
```

By using the option database to set the options of the display styles, we can advoid hard–coding the option values and give the user more flexibility in customization. See *Tk::option* for a detailed description of the option database.

STYLE METHODS

The **ItemStyle** method creates an object. This object supports the **configure** and **cget** methods described in *Tk::options* which can be used to enquire and modify the options described above.

The following additional methods are available for item styles:

$style->**delete**

Destroy this display style object.

EXAMPLE

The following example creates two columns of data in a HList widget. The first column is in red and the second column in blue. The colors of the columns are controlled by two different **text** styles. Also, the anchor and font of the second column is chosen so that the income data is aligned properly.

```
use strict;
use Tk;
use Tk::HList;
use Tk::ItemStyle;

my $mw = MainWindow->new();

my $hlist = $mw->HList(-columns=>2)->pack;

my $red  = $hlist->ItemStyle('text', -foreground=>'#800000');
my $blue = $hlist->ItemStyle('text', -foreground=>'#000080', -anchor=>'e');

my $e;
foreach ([Joe => '$10,000'], [Peter => '$20,000'],
         [Raj => '$90,000'],  [Zinh => '$0']) {
    $e = $hlist->addchild("");
    $hlist->itemCreate($e, 0, -itemtype=>'text',
             -text=>$_->[0], -style=>$red );
    $hlist->itemCreate($e, 1, -itemtype=>'text',
             -text=>$_->[1], -style=>$blue);
}

Tk::MainLoop;
```

SEE ALSO

Tk::HList\Tk::HList Tk::TixGrid\Tk::TixGrid Tk::TList\Tk::TList

KEYWORDS

display item, display style, item style

NAME

Tk::Entry – Create and manipulate Entry widgets

=for category Tk Widget Classes

SYNOPSIS

$entry = *$parent*->**Entry***(?options?);*

STANDARD OPTIONS

–background –highlightbackground–insertontime –selectforeground –borderwidth –highlightcolor –insertwidth –takefocus –cursor –highlightthickness –justify –textvariable –exportselection –insertbackground –relief –xscrollcommand –font –insertborderwidth –selectbackground –foreground –insertofftime –selectborderwidth

WIDGET–SPECIFIC OPTIONS

Name: **invalidCommand**
Class: **InvalidCommand**
Switch: **–invalidcommand**
Alias: **–invcmd**

Specifies a script to eval when **validateCommand** returns 0. Setting it to <undef disables this feature (the default). The best use of this option is to set it to *bell*. See **Validation** below for more information.

Name: **show**
Class: **Show**
Switch: **–show**

If this option is specified, then the true contents of the entry are not displayed in the window. Instead, each character in the entry‘s value will be displayed as the first character in the value of this option, such as ‘‘*’’. This is useful, for example, if the entry is to be used to enter a password. If characters in the entry are selected and copied elsewhere, the information copied will be what is displayed, not the true contents of the entry.

Name: **state**
Class: **State**
Switch: **–state**

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

Name: **validate**
Class: **Validate**
Switch: **–validate**

Specifies the mode in which validation should operate: **none**, **focus**, **focusin**, **focusout**, **key**, or **all**. It defaults to **none**. When you want validation, you must explicitly state which mode you wish to use. See **Validation** below for more.

Name: **validateCommand**
Class: **ValidateCommand**
Switch: **–validatecommand**
Alias: **–vcmd**

Specifies a script to eval when you want to validate the input into the entry widget. Setting it to `undef` disables this feature (the default). This command must return a valid boolean value. If it returns 0 (or the valid boolean equivalent) then it means you reject the new edition and it will not occur and the **invalidCommand** will be evaluated if it is set. If it returns 1, then the new edition occurs. See **Validation** below for more information.

Name: **width**
Class: **Width**
Switch: **–width**

Specifies an integer value indicating the desired width of the entry window, in average–size characters of the widget's font. If the value is less than or equal to zero, the widget picks a size just large enough to hold its current text.

DESCRIPTION

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

An entry is a widget that displays a one–line text string and allows that string to be edited using methods 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; methods described below may be used to change the view in the window. Entries use the standard **xScrollCommand** mechanism for interacting with scrollbars (see the description of the **–xscrollcommand** option for details). They also support scanning, as described below.

VALIDATION

Validation of entry widgets is derived from part of the patch written by jhobbs@cs.uoregon.edu. This works by setting the **validateCommand** option to a callback which will be evaluated according to the **validate** option as follows:

none

Default. This means no validation will occur.

focus

validateCommand will be called when the entry receives or loses focus.

focusin

validateCommand will be called when the entry receives focus.

focusout

validateCommand will be called when the entry loses focus.

key **validateCommand** will be called when the entry is edited.

all **validateCommand** will be called for all above conditions.

The **validateCommand** and **invalidCommand** are called with the following arguments:

- The proposed value of the entry. If you are configuring the

  entry widget to have a new textvariable, this will be the value of that textvariable.

- The characters to be added (or deleted). This will be `undef`

  if validation is due to focus, explcit call to validate or if change is due to `–textvariable` changing.

- The current value of entry i.e. before the proposed change.
- index of char string to be added/deleted, if any. −1 otherwise
- type of action. 1 == INSERT, 0 == DELETE,

  −1 if it‘s a forced validation or textvariable validation

In general, the **textVariable** and **validateCommand** can be dangerous to mix. If you try set the **textVariable** to something that the **validateCommand** will not accept it will be set back to the value of the entry widget. Using the **textVariable** for read−only purposes will never cause problems.

The **validateCommand** will turn itself off by setting **validate** to **none** when an error occurs, for example when the **validateCommand** or **invalidCommand** encounters an error in its script while evaluating, or **validateCommand** does not return a valid boolean value.

With the perl/Tk version **validate** option is supposed to be "suspended" while executing the **validateCommand** or the **invalidCommand**. This is experimental but in theory either callback can "correct" the value of the widget, and override the proposed change. (**validateCommand** should still return false to inhibit the change from happening when it returns.)

WIDGET METHODS

The **Entry** method creates a widget object. This object supports the **configure** and **cget** methods described in *Tk::options* which can be used to enquire and modify the options described above. The widget also inherits all the methods provided by the generic *Tk::Widget|Tk::Widget* class.

Many of the additional methods for entries take one or more indices as arguments. An index specifies a particular character in the entry‘s string, in any of the following ways:

number

Specifies the character as a numerical index, where 0 corresponds to the first character in the string.

anchor

Indicates the anchor point for the selection, which is set with the **selectionFrom** and **selectionAdjust** methods.

end Indicates the character just after the last one in the entry‘s string. This is equivalent to specifying a numerical index equal to the length of the entry‘s string.

insert

Indicates the character adjacent to and immediately following the insertion cursor.

sel.first

Indicates the first character in the selection. It is an error to use this form if the selection isn‘t in the entry window.

sel.last

Indicates the character just after the last one in the selection. It is an error to use this form if the selection isn‘t in the entry window.

@*number*

In this form, *number* is treated as an x−coordinate in the entry‘s window; the character spanning that x−coordinate is used. For example, ‘‘**@0**’’ indicates the left−most character in the window.

Abbreviations may be used for any of the forms above, e.g. ‘‘**e**’’ or ‘‘**sel.f**’’. In general, out−of−range indices are automatically rounded to the nearest legal value.

The following additional methods are available for entry widgets:

`$entry->`**bbox***(index)*

Returns a list of four numbers describing the bounding box of the character given by *index*. The first two elements of the list give the x and y coordinates of the upper−left corner of the screen area covered by the character (in pixels relative to the widget) and the last two elements give the width and height of

the character, in pixels. The bounding box may refer to a region outside the visible area of the window.

$entry->**delete**(*first*, ?*last*?)

Returns the current value of the configuration option given by *option*. *Option* may have any of the values accepted by the **entry** command.

$entry->**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 `$entry` (see *Tk::configure* 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.

$entry->**delete**(*first*, ?*last*?)

Delete one or more elements of the entry. *First* is the index of the first character to delete, and *last* is the index of the character just after the last one to delete. If *last* isn't specified it defaults to *first*+1, i.e. a single character is deleted. This method returns an empty string.

$entry->**get**

Returns the entry's string.

$entry->**icursor**(*index*)

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

$entry->**index**(*index*)

Returns the numerical index corresponding to *index*.

$entry->**insert**(*index, string*)

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

$entry–**scan**(*option,* args)
$entry–**scan***Option*(*args*)

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

$entry–**scanMark**(*x*)

Records *x* and the current view in the entry widget; used in conjunction with later **scanDragto** methods. Typically this method is associated with a mouse button press in the widget. It returns an empty string.

$entry–**scanDragto**(*x*)

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

$entry–**selection**(*option,* arg)
$entry–**selection***Option*(*arg*)

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

$entry->**selectionAdjust**(*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 **selectionTo** methods. 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.

$entry->**selectionClear**

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

$entry->**selectionFrom***(index)*

Set the selection anchor point to just before the character given by *index*. Doesn't change the selection. Returns an empty string.

$entry->**selectionPresent**

Returns 1 if there is are characters selected in the entry, 0 if nothing is selected.

$entry->**selectionRange***(start, end)*

Sets the selection to include the characters starting with the one indexed by *start* and ending with the one just before *end*. If *end* refers to the same character as *start* or an earlier one, then the entry's selection is cleared.

$entry->**selectionTo***(index)*

If *index* is before the anchor point, set the selection to the characters from *index* up to but not including the anchor point. If *index* is the same as the anchor point, do nothing. If *index* is after the anchor point, set the selection to the characters from the anchor point up to but not including *index*. The anchor point is determined by the most recent **selectionFrom** or **selectionAdjust** method 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.

$entry->**validate**

This command is used to force an evaluation of the **validateCommand** independent of the conditions specified by the **validate** option. It returns 0 or 1.

$entry->**xview***(args)*

This command is used to query and change the horizontal position of the text in the widget's window. It can take any of the following forms:

$entry->**xview**

Returns a list containing two elements. Each element is a real fraction between 0 and 1; together they describe the horizontal span that is visible in the window. For example, if the first element is .2 and the second element is .7, 20% of the entry's text is off−screen to the left, the middle 50% is visible in the window, and 30% of the text is off−screen to the right. These are the same values passed to scrollbars via the **−xscrollcommand** option.

$entry->**xview***(index)*

Adjusts the view in the window so that the character given by *index* is displayed at the left edge of the window.

$entry->**xviewMoveto***(fraction)*

Adjusts the view in the window so that the character *fraction* of the way through the text appears at the left edge of the window. *Fraction* must be a fraction between 0 and 1.

$entry->**xviewScroll***(number, what)*

This method shifts the view in the window left or right according to *number* and *what*. *Number* must be an integer. *What* must be either **units** or **pages** or an abbreviation of one of these. If *what* is **units**, the view adjusts left or right by *number* average−width characters on the display; if it is **pages** then the view adjusts by *number* screenfuls. If *number* is negative then characters farther to the left become visible; if it is positive then characters

farther to the right become visible.

DEFAULT BINDINGS

Tk automatically creates class bindings for entries that give them the following default behavior. In the descriptions below, ‘‘word’’ refers to a contiguous group of letters, digits, or ‘‘_’’ characters, or any single character other than these.

[1] Clicking mouse button 1 positions the insertion cursor just before the character underneath the mouse cursor, sets the input focus to this widget, and clears any selection in the widget. Dragging with mouse button 1 strokes out a selection between the insertion cursor and the character under the mouse.

[2] Double−clicking with mouse button 1 selects the word under the mouse and positions the insertion cursor at the beginning of the word. Dragging after a double click will stroke out a selection consisting of whole words.

[3] Triple−clicking with mouse button 1 selects all of the text in the entry and positions the insertion cursor before the first character.

[4] The ends of the selection can be adjusted by dragging with mouse button 1 while the Shift key is down; this will adjust the end of the selection that was nearest to the mouse cursor when button 1 was pressed. If the button is double−clicked before dragging then the selection will be adjusted in units of whole words.

[5] Clicking mouse button 1 with the Control key down will position the insertion cursor in the entry without affecting the selection.

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

[7] The view in the entry can be adjusted by dragging with mouse button 2. If mouse button 2 is clicked without moving the mouse, the selection is copied into the entry at the position of the mouse cursor.

[8] If the mouse is dragged out of the entry on the left or right sides while button 1 is pressed, the entry will automatically scroll to make more text visible (if there is more text off−screen on the side where the mouse left the window).

[9] The Left and Right keys move the insertion cursor one character to the left or right; they also clear any selection in the entry and set the selection anchor. If Left or Right is typed with the Shift key down, then the insertion cursor moves and the selection is extended to include the new character. Control−Left and Control−Right move the insertion cursor by words, and Control−Shift−Left and Control−Shift−Right move the insertion cursor by words and also extend the selection. Control−b and Control−f behave the same as Left and Right, respectively. Meta−b and Meta−f behave the same as Control−Left and Control−Right, respectively.

[10] The Home key, or Control−a, will move the insertion cursor to the beginning of the entry and clear any selection in the entry. Shift−Home moves the insertion cursor to the beginning of the entry and also extends the selection to that point.

[11] The End key, or Control−e, will move the insertion cursor to the end of the entry and clear any selection in the entry. Shift−End moves the cursor to the end and extends the selection to that point.

[12] The Select key and Control−Space set the selection anchor to the position of the insertion cursor. They don‘t affect the current selection. Shift−Select and Control−Shift−Space adjust the selection to the current position of the insertion cursor, selecting from the anchor to the insertion cursor if there was not any selection previously.

[13] Control−/ selects all the text in the entry.

[14] Control−\ clears any selection in the entry.

[15] The F16 key (labelled Copy on many Sun workstations) or Meta−w copies the selection in the widget to the clipboard, if there is a selection.

[16] The F20 key (labelled Cut on many Sun workstations) or Control−w copies the selection in the widget to the clipboard and deletes the selection. If there is no selection in the widget then these keys have no effect.

[17] The F18 key (labelled Paste on many Sun workstations) or Control−y inserts the contents of the clipboard at the position of the insertion cursor.

[18] The Delete key deletes the selection, if there is one in the entry. If there is no selection, it deletes the character to the right of the insertion cursor.

[19] The BackSpace key and Control−h delete the selection, if there is one in the entry. If there is no selection, it deletes the character to the left of the insertion cursor.

[20] Control−d deletes the character to the right of the insertion cursor.

[21] Meta−d deletes the word to the right of the insertion cursor.

[22] Control−k deletes all the characters to the right of the insertion cursor.

[23] Control−t reverses the order of the two characters to the right of the insertion cursor.

If the entry is disabled using the **−state** option, then the entry‘s view can still be adjusted and text in the entry can still be selected, but no insertion cursor will be displayed and no text modifications will take place.

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

KEYWORDS

entry, widget

NAME

Tk::Error – Method invoked to process background errors

=for category Binding Events and Callbacks

SYNOPSIS

Customization:

```
require Tk::ErrorDialog;
```

or

```
sub Tk::Error
{
  my ($widget,$error,@locations) = @_;
  ...
}
```

DESCRIPTION

The **Tk::Error** method is invoked by perl/Tk when a background error occurs. Two possible implementations are provided in the distribution and individual applications or users can (re)define a **Tk::Error** method (e.g. as a perl sub) if they wish to handle background errors in some other manner.

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 *callback\|Tk::callbacks* specified with a *bind\|Tk::bind* or a *after\|Tk::after* command, then it is a background error. For a non–background error, the error can simply be returned up through nested subroutines 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 saves information about the error and invokes the **Tk::Error** method later when Tk is idle.

Tk::Error is invoked by perl/Tk as if by the perl code:

$mainwindow–>**Tk::Error**(*"error message", location ...);*

$mainwindow is the **MainWindow** associated with widget which detected the error, *"error message"* is a string describing the error that has been detected, *location* is a list of one or more "locations" which describe the call sequence at the point the error was detected.

The locations are a typically a mixture of perl location reports giving script name and line number, and simple strings describing locations in core Tk or perl/Tk C code.

Tk will ignore any result returned by the **Tk::Error** method. If another error occurs within the **Tk::Error** method (for example if it calls **die**) then Tk reports this error itself by writing a message to stderr (this is to avoid infinite loops due to any bugs in **Tk::Error**).

If several background errors accumulate before **Tk::Error** is invoked to process them, **Tk::Error** will be invoked once for each error, in the order they occurred. However, if **Tk::Error** calls **Tk–>break**, then any remaining errors are skipped without calling **Tk::Error**.

The **Tk** module includes a default **Tk::Error** subroutine that simply reports the error on stderr.

An alternate definition is provided via :

```
require Tk::ErrorDialog;
```

that posts a dialog box containing the error message and offers the user a chance to see a stack trace showing where the error occurred.

BUGS

If **after** or **fileevent** are not invoked as methods of a widget then perl/Tk is unable to provide a *$mainwindow* argument. To support such code from earlier versions of perl/Tk perl/Tk therefore calls **Tk::Error** with string 'Tk' instead: **Tk->Tk::Error\(...\)**. In this case the **Tk::Error** in **Tk::ErrorDialog** and similar implementations cannot "popup" a window as they don't know which display to use. A mechanism to supply *the* **MainWindow** in applications which only have one (a very common case) should be provided.

SEE ALSO

Tk::bind\Tk::bind Tk::after\Tk::after Tk::fileevent\Tk::fileevent

KEYWORDS

background error, reporting

NAME

Tk::event – Miscellaneous event facilities: define virtual events and generate events

=for category Binding Events and Callbacks

SYNOPSIS

$widget->**event***Action*(*?arg, arg, ...?*);

DESCRIPTION

The **event***Action* methods provides several facilities for dealing with window system events, such as defining virtual events and synthesizing events. Virtual events are shared by all widgets of the same **MainWindow**. Different *MainWindow|Tk::MainWindow*s can have different virtual event.

The following methods are currently supported:

$widget->**eventAdd**(**'<<***virtual***>>'**, *sequence* ?,*sequence, ...*?)

Associates the virtual event *virtual* with the physical event sequence(s) given by the *sequence* arguments, so that the virtual event will trigger whenever any one of the *sequence*s occurs. *Virtual* may be any string value and *sequence* may have any of the values allowed for the *sequence* argument to the *bind|Tk::bind* method. If *virtual* is already defined, the new physical event sequences add to the existing sequences for the event.

$widget->**eventDelete**(**'<<***virtual***>>'** ?,*sequence, sequence, ...*?)

Deletes each of the *sequence*s from those associated with the virtual event given by *virtual*. *Virtual* may be any string value and *sequence* may have any of the values allowed for the *sequence* argument to the *bind|Tk::bind* method. Any *sequence*s not currently associated with *virtual* are ignored. If no *sequence* argument is provided, all physical event sequences are removed for *virtual*, so that the virtual event will not trigger anymore.

$widget->**eventGenerate**(*event* ?,*option => value, option => value, ...*?)

Generates a window event and arranges for it to be processed just as if it had come from the window system. *$window* is a reference to the window for which the event will be generated. *Event* provides a basic description of the event, such as **<Shift-Button-2>** or **<<Paste>>**. If *Window* is empty the whole screen is meant, and coordinates are relative to the screen. *Event* may have any of the forms allowed for the *sequence* argument of the *bind|Tk::bind* method except that it must consist of a single event pattern, not a sequence. *Option-value* pairs may be used to specify additional attributes of the event, such as the x and y mouse position; see *"EVENT FIELDS"* below. If the **-when** option is not specified, the event is processed immediately: all of the handlers for the event will complete before the **eventGenerate** method returns. If the **-when** option is specified then it determines when the event is processed.

$widget->**eventInfo**(*?***'<<***virtual***>>'***?*)

Returns information about virtual events. If the **<<***virtual***>>** argument is omitted, the return value is a list of all the virtual events that are currently defined. If **<<***virtual***>>** is specified then the return value is a list whose elements are the physical event sequences currently defined for the given virtual event; if the virtual event is not defined then **undef** is returned.

EVENT FIELDS

The following options are supported for the **eventGenerate** method. These correspond to the ''%'' expansions allowed in binding callback for the *bind|Tk::bind* method.

-above = *window*

Window specifies the *above* field for the event, either as a window path name or as an integer window id. Valid for **Configure** events. Corresponds to the *'a'* substitution for binding scripts.

-borderwidth = *size*

Size must be a screen distance; it specifies the *border_width* field for the event. Valid for **Configure** events. Corresponds to the *'B'* substitution for binding scripts.

–button = *number*

Number must be an integer; it specifies the *detail* field for a **ButtonPress** or **ButtonRelease** event, overriding any button number provided in the base *event* argument. Corresponds to the *'b'* substitution for binding scripts.

–count = *number*

Number must be an integer; it specifies the *count* field for the event. Valid for **Expose** events. Corresponds to the *'c'* substitution for binding scripts.

–detail = *detail*

Detail specifies the *detail* field for the event and must be one of the following:

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

Valid for **Enter**, **Leave**, **FocusIn** and **FocusOut** events. Corresponds to the *'d'* substitution for binding scripts.

–focus *boolean*

Boolean must be a boolean value; it specifies the *focus* field for the event. Valid for **Enter** and **Leave** events. Corresponds to the *'f'* substitution for binding scripts.

–height *size*

Size must be a screen distance; it specifies the *height* field for the event. Valid for **Configure** events. Corresponds to the *'h'* substitution for binding scripts.

–keycode *number*

Number must be an integer; it specifies the *keycode* field for the event. Valid for **KeyPress** and **KeyRelease** events. Corresponds to the *'k'* substitution for binding scripts.

–keysym *name*

Name must be the name of a valid keysym, such as **g**, **space**, or **Return**; its corresponding keycode value is used as the *keycode* field for event, overriding any detail specified in the base *event* argument. Valid for **KeyPress** and **KeyRelease** events. Corresponds to the *'K'* substitution for binding scripts.

–mode *notify*

Notify specifies the *mode* field for the event and must be one of **NotifyNormal**, **NotifyGrab**, **NotifyUngrab**, or **NotifyWhileGrabbed**. Valid for **Enter**, **Leave**, **FocusIn**, and **FocusOut** events. Corresponds to the *'m'* substitution for binding scripts.

–override *boolean*

Boolean must be a boolean value; it specifies the *override_redirect* field for the event. Valid for **Map**, **Reparent**, and **Configure** events. Corresponds to the *'o'* substitution for binding scripts.

–place *where*

Where specifies the *place* field for the event; it must be either **PlaceOnTop** or **PlaceOnBottom**. Valid for **Circulate** events. Corresponds to the *'p'* substitution for binding scripts.

–root *window*

Window must be either a window path name or an integer window identifier; it specifies the *root* field for the event. Valid for **KeyPress**, **KeyRelease**, **ButtonPress**, **ButtonRelease**, **Enter**, **Leave**, and **Motion** events. Corresponds to the *'R'* substitution for binding scripts.

–rootx *coord*

Coord must be a screen distance; it specifies the *x_root* field for the event. Valid for **KeyPress**, **KeyRelease**, **ButtonPress**, **ButtonRelease**, **Enter**, **Leave**, and **Motion** events. Corresponds to the *'X'*

substitution for binding scripts.

–rooty *coord*

Coord must be a screen distance; it specifies the *y_root* field for the event. Valid for **KeyPress**, **KeyRelease**, **ButtonPress**, **ButtonRelease**, **Enter**, **Leave**, and **Motion** events. Corresponds to the *'Y'* substitution for binding scripts.

–sendevent *boolean*

Boolean must be a boolean value; it specifies the *send_event* field for the event. Valid for all events. Corresponds to the *'E'* substitution for binding scripts.

–serial *number*

Number must be an integer; it specifies the *serial* field for the event. Valid for all events. Corresponds to the *'#'* substitution for binding scripts.

–state *state*

State specifies the *state* field for the event. For **KeyPress**, **KeyRelease**, **ButtonPress**, **ButtonRelease**, **Enter**, **Leave**, and **Motion** events it must be an integer value. For **Visibility** events it must be one of **VisibilityUnobscured**, **VisibilityPartiallyObscured**, or **VisibilityFullyObscured**. This option overrides any modifiers such as **Meta** or **Control** specified in the base *event*. Corresponds to the *'s'* substitution for binding scripts.

–subwindow *window*

Window specifies the *subwindow* field for the event, either as a path name for a Tk widget or as an integer window identifier. Valid for **KeyPress**, **KeyRelease**, **ButtonPress**, **ButtonRelease**, **Enter**, **Leave**, and **Motion** events. Similar to *'S'* substitution for binding scripts.

–time *integer*

Integer must be an integer value; it specifies the *time* field for the event. Valid for **KeyPress**, **KeyRelease**, **ButtonPress**, **ButtonRelease**, **Enter**, **Leave**, **Motion**, and **Property** events. Corresponds to the *'t'* substitution for binding scripts.

–warp *boolean*

boolean must be a boolean value; it specifies whether the screen pointer should be warped as well. Valid for **KeyPress**, **KeyRelease**, **ButtonPress**, **ButtonRelease**, and **Motion** events.

–width *size*

Size must be a screen distance; it specifies the *width* field for the event. Valid for **Configure** events. Corresponds to the *'w'* substitution for binding scripts.

–when *when*

When determines when the event will be processed; it must have one of the following values:

now Process the event immediately, before the command returns. This also happens if the **–when** option is omitted.

tail Place the event on perl/Tk's event queue behind any events already queued for this application.

head Place the event at the front of perl/Tk's event queue, so that it will be handled before any other events already queued.

mark Place the event at the front of perl/Tk's event queue but behind any other events already queued with **–when mark**. This option is useful when generating a series of events that should be processed in order but at the front of the queue.

–x *coord*

Coord must be a screen distance; it specifies the *x* field for the event. Valid for **KeyPress**, **KeyRelease**, **ButtonPress**, **ButtonRelease**, **Motion**, **Enter**, **Leave**, **Expose**, **Configure**, **Gravity**, and **Reparent** events. Corresponds to the the *'x'* substitution for binding scripts. If *Window* is empty the

coordinate is relative to the screen, and this option corresponds to the *'X'* substitution for binding scripts.

–y *coord*

Coord must be a screen distance; it specifies the *y* field for the event. Valid for **KeyPress**, **KeyRelease**, **ButtonPress**, **ButtonRelease**, **Motion**, **Enter**, **Leave**, **Expose**, **Configure**, **Gravity**, and **Reparent** events. Corresponds to the the *'y'* substitution for binding scripts. If *Window* is empty the coordinate is relative to the screen, and this option corresponds to the *'Y'* substitution for binding scripts.

Any options that are not specified when generating an event are filled with the value 0, except for *serial*, which is filled with the next X event serial number.

VIRTUAL EVENT EXAMPLES

In order for a virtual event binding to trigger, two things must happen. First, the virtual event must be defined with the **eventAdd** method. Second, a binding must be created for the virtual event with the **bind** method. Consider the following virtual event definitions:

```
$widget->eventAdd('<<Paste>>' => '<Control-y>');
$widget->eventAdd('<<Paste>>' => '<Button-2>');
$widget->eventAdd('<<Save>>' => '<Control-X><Control-S>');
$widget->eventAdd('<<Save>>' => '<Shift-F12>');
```

In the **bind** method, a virtual event can be bound like any other builtin event type as follows:

```
$entry->bind('Tk::Entry', '<<Paste>>' => sub {
               $entry->Insert($entry->selectionGet) });
```

The double angle brackets are used to specify that a virtual event is being bound. If the user types Control–y or presses button 2, or if a **<<Paste>>** virtual event is synthesized with **eventGenerate**, then the **<<Paste>>** binding will be invoked.

If a virtual binding has the exact same sequence as a separate physical binding, then the physical binding will take precedence. Consider the following example:

```
$mw->eventAdd('<<Paste>>' => '<Control-y>','<Meta-Control-y>');
$mw->bind('Tk::Entry', '<Control-y>' => sub{print 'Control-y'});
$mw->bind('Tk::Entry', '<<Paste>>'   => sub{print 'Paste'});
```

When the user types Control–y the **<Control–y>** binding will be invoked, because a physical event is considered more specific than a virtual event, all other things being equal. However, when the user types Meta–Control–y the **<<Paste>>** binding will be invoked, because the **Meta** modifier in the physical pattern associated with the virtual binding is more specific than the **<Control–y>** sequence for the physical event.

Bindings on a virtual event may be created before the virtual event exists. Indeed, the virtual event never actually needs to be defined, for instance, on platforms where the specific virtual event would meaningless or ungeneratable.

When a definition of a virtual event changes at run time, all windows will respond immediately to the new definition. Starting from the preceding example, if the following code is executed:

```
$entry->bind(ref($entry), '<Control-y>' => undef);
$entry->eventAdd('<<Paste>>' => '<Key-F6>');
```

the behavior will change such in two ways. First, the shadowed **<<Paste>>** binding will emerge. Typing Control–y will no longer invoke the **<Control–y>** binding, but instead invoke the virtual event **<<Paste>>**. Second, pressing the F6 key will now also invoke the **<<Paste>>** binding.

SEE ALSO

Tk::bind\|Tk::bind Tk::callbacks\|Tk::callbacks

KEYWORDS

event, binding, define, handle, virtual event

NAME

Tk::Event – ToolKit for Events

=for category Implementation

SYNOPSIS

```
use Tk::Event;

Tk::Event->fileevent(\*FH, 'readable' => callback);

Tk::Event->lineavail(\*FH, callback);

use Tk::Event::Signal qw(INT);

$SIG{'INT'} = callback;

use Tk::Event::process;

Tk::Event->proc($pid, callback);

QueueEvent(callback [, position])
```

DESCRIPTION

That is better than nothing but still hard to use. Most scripts want higher level result (a line, a "block" of data etc.)

So it has occured to me that we could use new–ish TIEHANDLE thus:

my $obj = tie SOMEHANDLE,Tk::Event::IO;

while (<SOMEHANDLE)
{
}

Then the READLINE routine registers a callback and looks something like:

```
sub READLINE
 {
 my $obj = shift;
 Event–io(*$obj,'readable',sub { sysread(*$obj,${*$obj},1,length(${*$obj}) });
 my $pos;
 while (($pos = index(${*$obj},$/) < 0)
  {
   DoOneEvent();
  }
 Event–io(*$obj,'readable','');  # unregister
 $pos += length($/);
 my $result = substr(${*$obj},0,$pos);
 substr(${*$obj},0,$pos) = '';
 return $result;
}
```

This is using the scalar part of the glob representing the _inner_ IO as a buffer in which to accumulate chars.

NAME

Tk::exit – End the application

=for category Binding Events and Callbacks

SYNOPSIS

```
use Tk qw(exit);
...
B<exit>?(I<returnCode>)?;
```

DESCRIPTION

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

If calling `exit` from code invoked via a Tk callback then this Tk version of `exit` cleans up more reliably than using the perl `exit`.

KEYWORDS

exit, process

NAME

Tk::fileevent − Execute a callback when a filehandle becomes readable or writable

=for category Binding Events and Callbacks

SYNOPSIS

$widget−>**fileevent**(*fileHandle,***readable***?,callback?)*

$widget−>**fileevent**(*fileHandle,***writable***?,callback?)*

DESCRIPTION

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

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

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

A filehandle is considered to be readable if there is unread data available on the underlying device. A filehandle is also considered to be readable if an end of file or error condition is present on the underlying file or device. It is important for *callback* to check for these conditions and handle them appropriately; for example, if there is no special check for end of file, an infinite loop may occur where *callback* reads no data, returns, and is immediately invoked again.

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

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

The callback for a file event is executed in the context of *`$widget`* with which **fileevent** was invoked. If an error occurs while executing the callback then the *Tk::Error* mechanism is used to report the error. In addition, the file event handler is deleted if it ever returns an error; this is done in order to prevent infinite loops due to buggy handlers.

BUGS

On windows platforms **fileevent** is limited in the types of filehandles that behave correctly. Making filefhandles non−blocking is only implemented on a subset of UNIX platforms (see *Tk::IO*).

CREDITS

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

SEE ALSO

Tk::IO|Tk::IO Tk::callbacks|Tk::callbacks

KEYWORDS

asynchronous I/O, blocking, filehandle, event handler, nonblocking, readable, callback, writable.

NAME

Tk::FileSelect – a widget for choosing files

=for pm Tk/FileSelect.pm

=for category Popups and Dialogs

SYNOPSIS

```
use Tk::FileSelect;

$FSref = $top->FileSelect(-directory => $start_dir);
             $top            - a window reference, e.g. MainWindow->new
             $start_dir      - the starting point for the FileSelect
$file = $FSref->Show;
             Executes the fileselector until either a filename is
             accepted or the user hits Cancel. Returns the filename
             or the empty string, respectively, and unmaps the
             FileSelect.
$FSref->configure(option => value[, ...])
             Please see the Populate subroutine as the configuration
             list changes rapidly.
```

DESCRIPTION

This Module pops up a Fileselector box, with a directory entry on top, a list of directories in the current directory, a list of files in the current directory, an entry for entering/modifying a file name, an accept button and a cancel button.

You can enter a starting directory in the directory entry. After hitting Return, the listboxes get updated. Double clicking on any directory shows you the respective contents. Single clicking on a file brings it into the file entry for further consideration, double clocking on a file pops down the file selector and calls the optional command with the complete path for the selected file. Hitting return in the file selector box or pressing the accept button will also work. *NOTE* the file selector box will only then get destroyed if the file name is not zero length. If you want yourself take care of it, change the if(length(.. in sub accept_file.

AUTHORS

Based on original FileSelect by Klaus Lichtenwalder, Lichtenwalder@ACM.org, Datapat GmbH, Munich, April 22, 1995 adapted by Frederick L. Wagner, derf@ti.com, Texas Instruments Incorporated, Dallas, 21Jun95

HISTORY

950621 — The following changes were made:

- Rewrote Tk stuff to take advantage of new Compound widget module, so FileSelect is now composed of 2 LabEntry and 2 ScrlListbox2 subwidgets.
- Moved entry labels (from to the left of) to above the entry fields.
- Caller is now able to control these aspects of widget, in both FileSelect (new) and configure :

  (Please see subroutine Populate for details, as these options change rapidly!)
- I changed from Double−Button−1 to Button−1 in the Files listbox, to work with multiple mode in addition to browse mode. I also made some name changes (LastPath — saved_path, ...).
- The show method is not yet updated.
- The topLevel stuff is not done yet. I took it out while I toy with the idea of FileSelect as a subwidget. Then the ‘normal’ topLevel thing with Buttons along the bottom could be build on top of it.

- By request of Henry Katz <katz@fs09.webo.dg.com, I added the functionality of using the Directory entry as a filter. So, if you want to only see the *.c files, you add a .c (the *‘s already there :) and hit return.

95/10/17, SOL, LUCC. lusol@Lehigh.EDU

-
```
Allow either file or directory names to be accepted.
```
- Require double click to move into a new directory rather than a single click. This allows a single click to select a directory name so it can be accepted.
- Add −verify list option so that standard Perl file test operators (like −d and −x) can be specified for further name validation. The default value is the special value ‘!−d’ (not a directory), so any name can be selected as long as it‘s not a directory − after all, this IS FileSelect!

  For example:

  ```
  $fs->configure(-verify => ['-d', [\&verify_code, $P1, $P2, ... $Pn]]);
  ```

  ensures that the selected name is a directory. Further, if an element of the list is an array reference, the first element is a code reference to a subroutine and the remaining optional elements are it‘s parameters. The subroutine is called like this:

  ```
  &verify_code($cd, $leaf, $P1, $P2, ... $Pn);
  ```

  where `$cd` is the current directory, `$leaf` is a directory or file name, and `$P1` .. `$Pn` are your optional parameters. The subroutine should return TRUE if success or FALSE if failure.

961008 — derf@ti.com :

By request of Jim Stern <js@world.northgrum.com and Brad Vance <bvance@ti.com, I updated the Accept and Show functions to support selection of multiple files. I also corrected a typo in the −verify code.

NAME

focus – Manage the input focus

=for category User Interaction

SYNOPSIS

$widget->**focus**

$widget->**focus***Option*

$widget->**focusNext**

$widget->**focusPrev**

$widget->**focusFollowsMouse**

DESCRIPTION

The **focus** methods are used to manage the Tk input focus. At any given time, one window on each display is designated as the *focus window*; any key press or key release events for the display are sent to that window. It is normally up to the window manager to redirect the focus among the top–level windows of a display. For example, some window managers automatically set the input focus to a top–level window whenever the mouse enters it; others redirect the input focus only when the user clicks on a window. Usually the window manager will set the focus only to top–level windows, leaving it up to the application to redirect the focus among the children of the top–level.

Tk remembers one focus window for each top–level (the most recent descendant of that top–level to receive the focus); when the window manager gives the focus to a top–level, Tk automatically redirects it to the remembered window. Within a top–level Tk uses an *explicit* focus model by default. Moving the mouse within a top–level does not normally change the focus; the focus changes only when a widget decides explicitly to claim the focus (e.g., because of a button click), or when the user types a key such as Tab that moves the focus.

The method **focusFollowsMouse** may be invoked to create an *implicit* focus model: it reconfigures Tk so that the focus is set to a window whenever the mouse enters it. The methods **focusNext** and **focusPrev** implement a focus order among the windows of a top–level; they are used in the default bindings for Tab and Shift–Tab, among other things.

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

$widget->**focusCurrent**

Returns the focus window on the display containing the *$widget*, or an empty string if no window in this application has the focus on that display.

$widget->**focus**

If the application currently has the input focus on *$widget*'s display, this command resets the input focus for *$widget*'s display to *$widget* and returns an empty string. If the application doesn't currently have the input focus on *$widget*'s display, *$widget* will be remembered as the focus for its top–level; the next time the focus arrives at the top–level, Tk will redirect it to *$widget*.

$widget->**focusForce**

Sets the focus of *$widget*'s display to *$widget*, even if the application doesn't currently have the input focus for the display. This command should be used sparingly, if at all. In normal usage, an application should not claim the focus for itself; instead, it should wait for the window manager to give it the focus.

$widget->**focusLast**

Returns the name of the most recent window to have the input focus among all the windows in the same top–level as *$widget*. If no window in that top–level has ever had the input focus, or if the most recent focus window has been deleted, then the top–level is returned. The return value is the window that will receive the input focus the next time the window manager gives the focus to the

top–level.

$widget–>**focusNext**
$widget–>**focusPrev**

focusNext is a utility method used for keyboard traversal, but can be useful in other contexts. It sets the focus to the ''next'' window after *$widget* in focus order. The focus order is determined by the stacking order of windows and the structure of the window hierarchy. Among siblings, the focus order is the same as the stacking order, with the lowest window being first. If a window has children, the window is visited first, followed by its children (recursively), followed by its next sibling. Top–level windows other than *$widget* are skipped, so that **focusNext** never returns a window in a different top–level from *$widget*.

After computing the next window, **focusNext** examines the window's **–takefocus** option to see whether it should be skipped. If so, **focusNext** continues on to the next window in the focus order, until it eventually finds a window that will accept the focus or returns back to *$widget*.

focusPrev is similar to **focusNext** except that it sets the focus to the window just before *$widget* in the focus order.

$widget–>**focusFollowsMouse**

focusFollowsMouse changes the focus model for the application to an implicit one where the window under the mouse gets the focus. After this procedure is called, whenever the mouse enters a window Tk will automatically give it the input focus. The **focus** command may be used to move the focus to a window other than the one under the mouse, but as soon as the mouse moves into a new window the focus will jump to that window. Note: at present there is no built–in support for returning the application to an explicit focus model; to do this you'll have to write a script that deletes the bindings created by **focusFollowsMouse**.

QUIRKS

When an internal window receives the input focus, Tk doesn't actually set the X focus to that window; as far as X is concerned, the focus will stay on the top–level window containing the window with the focus. However, Tk generates FocusIn and FocusOut events just as if the X focus were on the internal window. This approach gets around a number of problems that would occur if the X focus were actually moved; the fact that the X focus is on the top–level is invisible unless you use C code to query the X server directly.

CAVEATS

Note that for the **Canvas** widget, the call to **focus** has to be fully qualified. This is because there is already a focus method for the **Canvas** widget, which sets the focus on individual canvas tags.

$canvas–>**Tk::focus**

KEYWORDS

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

NAME

font – Create and inspect fonts.

=for category Tk Generic Methods

SYNOPSIS

$widget–**Font**(*option*?, *arg, arg, ...?)*

$font–*Option*?(*arg, arg, ...)?*

DESCRIPTION

The **Font** method provides several facilities for dealing with fonts, such as defining named fonts and inspecting the actual attributes of a font. The command has several different forms, determined by the first argument. The following forms are currently supported:

$font–>**actual**(*–option?)*
$widget–>**fontActual**(*font?, –option*?)

Returns information about the actual attributes that are obtained when *font* is used on *$font's* display; the actual attributes obtained may differ from the attributes requested due to platform–dependant limitations, such as the availability of font families and pointsizes. *font* is a font description; see *"FONT DESCRIPTION"* below. If *option* is specified, returns the value of that attribute; if it is omitted, the return value is a list of all the attributes and their values. See *"FONT OPTIONS"* below for a list of the possible attributes.

$font–>**configure**(*–option??=value, –option=value, ...*?)

Query or modify the desired attributes for *$font*. If no *–option* is specified, returns a list describing all the options and their values for *fontname*. If a single *–option* is specified with no *value*, then returns the current value of that attribute. If one or more *option–value* pairs are specified, then the method modifies the given named font to have the given values; in this case, all widgets using that font will redisplay themselves using the new attributes for the font. See *"FONT OPTIONS"* below for a list of the possible attributes.

Note: the above behaviour differs in detail to **configure** on widgets, images etc.

$font = *$widget*–>**Font**(*–option=value, ...*?)
$font = *$widget*–>**fontCreate**(*?fontname??, –option=value, ...*?)

Creates a new font object and returns a reference to it. *fontname* specifies the name for the font; if it is omitted, then Tk generates a new name of the form **font***x*, where *x* is an integer. There may be any number of *option–value* pairs, which provide the desired attributes for the new named font. See *"FONT OPTIONS"* below for a list of the possible attributes.

Note: the created font is *not* shared between widgets of different *MainWindow|Tk::MainWindow*s.

$font–>**delete**
$widget–>**fontDelete**(*fontname?, fontname, ...*?)

Delete the specified named fonts. If there are widgets using the named font, the named font won't actually be deleted until all the instances are released. Those widgets will continue to display using the last known values for the named font. If a deleted named font is subsequently recreated with another call to **fontCreate**, the widgets will use the new named font and redisplay themselves using the new attributes of that font.

$widget–>**fontFamilies**

The return value is a list of the case–insensitive names of all font families that exist on *$widget's* display.

$font–>**measure**(*text)*
$widget–>**fontMeasure**(*font, text*)

Measures the amount of space the string *text* would use in the given *font* when displayed in *$widget*.

font is a font description; see *"FONT DESCRIPTION"* below. The return value is the total width in pixels of *text*, not including the extra pixels used by highly exagerrated characters such as cursive ‘‘*f*’’. If the string contains newlines or tabs, those characters are not expanded or treated specially when measuring the string.

$font–>**metrics***(–option?)*
$widget–>**fontMetrics***(font?, –option*?)

Returns information about the metrics (the font–specific data), for *font* when it is used on *$widget*‘*s* display. *font* is a font description; see *"FONT DESCRIPTION"* below. If *option* is specified, returns the value of that metric; if it is omitted, the return value is a list of all the metrics and their values. See *"FONT METRICS"* below for a list of the possible metrics.

$widget–>**fontNames**

The return value is a list of all font objects that are currently defined for *$widget*‘*s* MainWindow.

FONT DESCRIPTION

The following formats are accepted as a font description anywhere *font* is specified as an argument above; these same forms are also permitted when specifying the **–font** option for widgets.

[1] *fontname*

The name of a named font, created using the **fontCreate** method. When a widget uses a named font, it is guaranteed that this will never cause an error, as long as the named font exists, no matter what potentially invalid or meaningless set of attributes the named font has. If the named font cannot be displayed with exactly the specified attributes, some other close font will be substituted automatically.

[1a] *$font*

A font object created using the **Font** method. This is essentially the same as using a named font. The object is a reference to the name, and carries additional information e.g. which MainWindow it relates to in an manner peculiar to perl/Tk.

[3] *systemfont*

The platform–specific name of a font, interpreted by the graphics server. This also includes, under X, an XLFD (see *[4]*) for which a single ‘‘*’’ character was used to elide more than one field in the middle of the name. See *"PLATFORM–SPECIFIC ISSUES"* for a list of the system fonts.

[3] [*family*,?*size*,??*style*,??*style ...*?]

A properly formed list whose first element is the desired font *family* and whose optional second element is the desired *size*. The interpretation of the *size* attribute follows the same rules described for *–size* in *"FONT OPTIONS"* below. Any additional optional arguments following the *size* are font *style*s. Possible values for the *style* arguments are as follows:

```
normal      bold    roman   italic
underline   overstrike
```

[4] X–font names (XLFD)

A Unix–centric font name of the form *–foundry–family–weight–slant–setwidth–addstyle–pixel–point–resx–resy–spacing–width–charset–encoding*. The ‘‘*’’ character may be used to skip individual fields that the user does not care about. There must be exactly one ‘‘*’’ for each field skipped, except that a ‘‘*’’ at the end of the XLFD skips any remaining fields; the shortest valid XLFD is simply ‘‘*’’, signifying all fields as defaults. Any fields that were skipped are given default values. For compatibility, an XLFD always chooses a font of the specified pixel size (not point size); although this interpretation is not strictly correct, all existing applications using XLFDs assumed that one ‘‘point’’ was in fact one pixel and would display incorrectly (generally larger) if the correct size font were actually used.

[5] *option value* ?*option value ...*?

A properly formed list of *option–value* pairs that specify the desired attributes of the font, in the same format used when defining a named font; see *"FONT OPTIONS"* below.

When font description *font* is used, the system attempts to parse the description according to each of the above five rules, in the order specified. Cases [1] and [2] must match the name of an existing named font or of a system font. Cases [3], [4], and [5] are accepted on all platforms and the closest available font will be used. In some situations it may not be possible to find any close font (e.g., the font family was a garbage value); in that case, some system–dependant default font is chosen. If the font description does not match any of the above patterns, an error is generated.

FONT METRICS

The following options are used by the **metrics/fontMetrics** method to query font–specific data determined when the font was created. These properties are for the whole font itself and not for individual characters drawn in that font. In the following definitions, the ‘‘baseline‘‘ of a font is the horizontal line where the bottom of most letters line up; certain letters, such as lower–case ‘‘g‘‘ stick below the baseline.

–ascent

The amount in pixels that the tallest letter sticks up above the baseline of the font, plus any extra blank space added by the designer of the font. (*$font–<gt*__ascent__ is provided for compatibility.)

–descent

The largest amount in pixels that any letter sticks down below the baseline of the font, plus any extra blank space added by the designer of the font. (*$font–<gt*__descent__ is provided for compatibility.)

–linespace

Returns how far apart vertically in pixels two lines of text using the same font should be placed so that none of the characters in one line overlap any of the characters in the other line. This is generally the sum of the ascent above the baseline line plus the descent below the baseline.

–fixed

Returns a boolean flag that is ‘‘**1**‘‘ if this is a fixed–width font, where each normal character is the the same width as all the other characters, or is ‘‘‘‘ if this is a proportionally–spaced font, where individual characters have different widths. The widths of control characters, tab characters, and other non–printing characters are not included when calculating this value.

FONT OPTIONS

The following options are supported on all platforms, and are used when constructing a named font or when specifying a font using style [5] as above:

–family = *name*

The case–insensitive font family name. Tk guarantees to support the font families named **Courier** (a monospaced ‘‘typewriter‘‘ font), **Times** (a serifed ‘‘newspaper‘‘ font), and **Helvetica** (a sans–serif ‘‘European‘‘ font). The most closely matching native font family will automatically be substituted when one of the above font families is used. The *name* may also be the name of a native, platform–specific font family; in that case it will work as desired on one platform but may not display correctly on other platforms. If the family is unspecified or unrecognized, a platform–specific default font will be chosen.

–size = *size*

The desired size of the font. If the *size* argument is a positive number, it is interpreted as a size in points. If *size* is a negative number, its absolute value is interpreted as a size in pixels. If a font cannot be displayed at the specified size, a nearby size will be chosen. If *size* is unspecified or zero, a platform–dependent default size will be chosen.

The original Tcl/Tk authors believe sizes should normally be specified in points so the application will remain the same ruler size on the screen, even when changing screen resolutions or moving scripts across platforms. While this is an admirable goal it does not work as well in practice as they hoped. The mapping between points and pixels is set when the application starts, based on alleged properties of the installed monitor, but it can be overridden by calling the *scaling* command. However this can be problematic when system has no way of telling if (say) an 11" or 22" monitor is attached, also if it *can*

tell then some monitor sizes may result in poorer quality scaled fonts being used rather than a "tuned" bitmap font. In addition specifying pixels is useful in certain circumstances such as when a piece of text must line up with respect to a fixed−size bitmap.

At present the Tcl/Tk scheme is used unchanged, with "point" size being returned by *actual* (as an integer), and used internally. Suggestions for work−rounds to undesirable behaviour welcome.

−weight = *weight*

The nominal thickness of the characters in the font. The value **normal** specifies a normal weight font, while **bold** specifies a bold font. The closest available weight to the one specified will be chosen. The default weight is **normal**.

−slant = *slant*

The amount the characters in the font are slanted away from the vertical. Valid values for slant are **roman** and **italic**. A roman font is the normal, upright appearance of a font, while an italic font is one that is tilted some number of degrees from upright. The closest available slant to the one specified will be chosen. The default slant is **roman**.

−underline = *boolean*

The value is a boolean flag that specifies whether characters in this font should be underlined. The default value for underline is **false**.

−overstrike = *boolean*

The value is a boolean flag that specifies whether a horizontal line should be drawn through the middle of characters in this font. The default value for overstrike is **false**.

PLATFORM−SPECIFIC ISSUES

The following named system fonts are supported:

X Windows:

All valid X font names, including those listed by xlsfonts(1), are available.

MS Windows:

```
system       ansi       device
systemfixed  ansifixed  oemfixed
```

Macintosh:

```
system       application
```

COMPATIBILITY WITH PREVIOUS VERSIONS

In prior versions of perl/Tk the *$widget*−>**Font** method was a perl wrapper on the original "[4] X−font names (XLFD)" style as described above (which was the only form supported by versions of core tk prior to version tk8.0). This module is provided in its original form (it has just been renamed) via:

```
use Tk::X11Font;
I<$widget>-E<gt>B<X11Font>(...)
```

However the methods of the old scheme have been mimiced as closely as possible with the new scheme. It is intended that code should work without modification, except for the case of using :

```
@names = $font->Name;
```

i.e. the *Name* method in an array/list context. This now returns one element on all platforms (as it did on Win32), while previously on X systems it returned a list of fonts that matched an under−specified pattern.

Briefly the methods supported for compatibilty are as follows:

$newfont = *$font*−>**Clone***(−option=value, ...?)*

Returns a new font object *$newfont* related to the original *$font* by changing the values of the specified −*option*s.

$font–>*Family* **– maps to –family**
$font–>*Weight* – maps to –weight
$font–>*Slant* – maps to –slant
$font–>*Pixel* and Point – map to –size

New code should use *$font*–>**configure** to achieve same effect as last four items above.

Foundry, Swidth, Adstyle, Xres, Yres, Space, Avgwidth, Registry, Encoding

Are all ignored if set, and return '*' if queried.

$font–>**Name**

Returns the name of a named font, or a string representation of an unnamed font. Using *$font* in a scalar context does the same. Note this is distinctly different from behaviour of *Name([$max])* in a list context.

$font–>**Pattern**

Returns a XLFD string for the font based on *actual* values, and some heuristics to map Tk's forms to the "standard" X conventions.

SEE ALSO

Tk::options|Tk::options

Tk::X11Font|Tk::X11Font

KEYWORDS

font

NAME

Tk::form – Geometry manager based on attachment rules

=for category Tk Geometry Management

SYNOPSIS

$widget–**form**?(*args)?*

$widget–**form***Option*?(*args)?*

DESCRIPTION

The **form** method is used to communicate with the **form** Geometry Manager, a geometry manager that arranges the geometry of the children in a parent window according to attachment rules. The **form** geometry manager is very flexible and powerful; it can be used to emulate all the existing features of the Tk packer and placer geometry managers (see *pack\|Tk::pack*, *place\|Tk::place*). The **form** method can have any of several forms, depending on *Option*:

$slave–**form**?(*options*)?

Sets or adjusts the attachment values of the slave window according to the –*option*=*value* argument pairs.

–b = *attachment*

Abbreviation for the **–bottom** option.

–bottom = *attachment*

Specifies an attachment for the bottom edge of the slave window. The attachment must specified according to *"SPECIFYING ATTACHMENTS"* below.

–bottomspring = *weight*

Specifies the weight of the spring at the bottom edge of the slave window. See *"USING SPRINGS"* below.

–bp = *value*

Abbreviation for the **–padbottom** option.

–bs = *weight*

Abbreviation for the **–bottomspring** option.

–fill = *style*

Specifies the fillings when springs are used for this widget. The value must be **x**, **y**, **both** or **none**.

–in = *$master*

Places the slave window into the specified *$master* window. If the slave was originally in another master window, all attachment values with respect to the original master window are discarded. Even if the attachment values are the same as in the original master window, they need to be specified again. The **–in** flag, when needed, must appear as the first flag of *options*. Otherwise an error is generated.

–l = *attachment*

Abbreviation for the **–left** option.

–left = *attachment*

Specifies an attachment for the left edge of the slave window. The attachment must specified according to *"SPECIFYING ATTACHMENTS"* below.

–leftspring = *weight*

Specifies the weight of the spring at the left edge of the slave window. See *"USING SPRINGS"* below.

–lp = *value*

Abbreviation for the **–padleft** option.

–ls = *weight*

Abbreviation for the **–leftspring** option.

–padbottom = *value*

Specifies the amount of external padding to leave on the bottom side of the slave. The *value* may have any of the forms acceptable to **Tk_GetPixels**.

–padleft = *value*

Specifies the amount of external padding to leave on the left side of the slave.

–padright = *value*

Specifies the amount of external padding to leave on the right side of the slave.

–padtop = *value*

Specifies the amount of external padding to leave on the top side of the slave.

–padx = *value*

Specifies the amount of external padding to leave on both the left and the right sides of the slave.

–pady = *value*

Specifies the amount of external padding to leave on both the top and the bottom sides of the slave.

–r = *attachment*

Abbreviation for the **–right** option.

–right = *attachment*

Specifies an attachment for the right edge of the slave window. The attachment must specified according to *"SPECIFYING ATTACHMENTS"* below.

–rightspring = *weight*

Specifies the weight of the spring at the right edge of the slave window. See *"USING SPRINGS"* below.

–rp = *value*

Abbreviation for the **–padright** option.

–rs = *weight*

Abbreviation for the **–rightspring** option.

–t = *attachment*

Abbreviation for the **–top** option.

–top = *attachment*

Specifies an attachment for the top edge of the slave window. The attachment must specified according to *"SPECIFYING ATTACHMENTS"* below.

–topspring = *weight*

Specifies the weight of the spring at the top edge of the slave window. See *"USING SPRINGS"* below.

–tp = *value*

Abbreviation for the **–padtop** option.

–ts = *weight*

Abbreviation for the **–topspring** option.

$master–**formCheck**

This method checks whether there is circular dependency in the attachments of the master's slaves (see *"CIRCULAR DEPENDENCY"* below). It returns the Boolean value **TRUE** if it discover circular dependency and **FALSE** otherwise.

$slave–**formForget**

Removes the slave from its master and unmaps its window. The slave will no longer be managed by form. All attachment values with respect to its master window are discarded. If another slave is attached to this slave, then the attachment of the other slave will be changed to grid attachment based on its geometry.

$master–**formGrid**?(*x_size,* y_size)?

When *x_size* and *y_size* are given, this method returns the number of grids of the *$master* window in a pair of integers of the form (*x_size, y_size*). When both *x_size* and *y_size* are given, this method changes the number of horizontal and vertical grids on the master window.

$slave–**formInfo**?(*–option)*?

Queries the attachment options of a slave window. *–option* can be any of the options accepted by the **form** method. If *–option* is given, only the value of that option is returned. Otherwise, this method returns a list whose elements are the current configuration state of the slave given in the same *option–value* form that might be specified to **form**. The first two elements in this list list are "**–in**=*$master*" where *$master* is the slave's master window.

$master–**formSlaves**

Returns a list of all of the slaves for the master window. 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.

SPECIFYING ATTACHMENTS

One can specify an attachment for each side of a slave window managed by form. An attachment is specified in the the form "*–side* = [*anchor_point*, *offset*]". *–side* can be one of **–top**, **–bottom**, **–left** or **–right**.

Offset is given in screen units (i.e. any of the forms acceptable to **Tk_GetPixels**). A positive offset indicates shifting to a position to the right or bottom of an anchor point. A negative offset indicates shifting to a position to the left or top of an anchor point.

Anchor_point can be given in one of the following forms:

Grid Attachment

The master window is divided into a number of horizontal and vertical grids. By default the master window is divided into 100x100 grids; the number of grids can be adjusted by the **formGrid** method. A grid attachment anchor point is given by a **%** sign followed by an integer value. For example, **'%0'** specifies the first grid line (the top or left edge of the master window). **'%100'** specifies the last grid line (the bottom or right edge of the master window).

Opposite Side Attachment

Opposite attachment specifies an anchor point located on the **opposite** side of another slave widget, which must be managed by form in the same master window. An opposite attachment anchor point is given by the name of another widget. For example, "*$b*–**form**(**–top**=[*$a*,0])" attaches the top side of the widget *$b* to the bottom of the widget *$a*.

Parallel Side Attachment

Opposite attachment specifies an anchor point located on the **same** side of another slave widget, which must be managed by form in the same master window. An parallel attachment anchor point is given by the sign **&** follwed by the name of another widget. For example, "*$b*–**form**(**–top**=['&',*$a*,0])" attaches the top side of the widget *$b* to the top of the widget *$a,* making the top sides of these two

widgets at the same vertical position in their parent window.

No Attachment

Specifies a side of the slave to be attached to nothing, indicated by the keyword **none**. When the **none** anchor point is given, the offset must be zero (or not present). When a side of a slave is attached to **['none', 0]**, the position of this side is calculated by the position of the other side and the natural size of the slave. For example, if a the left side of a widget is attached to **['%0', 100]**, its right side attached to **['none', 0]**, and the natural size of the widget is **50** pixels, the right side of the widget will be positioned at pixel **['%0', 149]**. When both **–top** and **–bottom** are attached to **none**, then by default **–top** will be attached to **['%0', 0]**. When both **–left** and **–right** are attached to none, then by default **–left** will be attached to **['%0', 0]**.

Shifting effects can be achieved by specifying a non–zero offset with an anchor point. In the following example, the top side of widget *\$b* is attached to the bottom of *\$a;* hence *\$b* always appears below *\$a.* Also, the left edge of *\$b* is attached to the left side of *\$a* with a 10 pixel offest. Therefore, the left edge of *\$b* is always shifted 10 pixels to the right of *\$a's* left edge:

\$b–**form**(**–left**=[*\$a*,10], **–top**=[*\$a,0]);*

ABBREVIATIONS:

Certain abbreviations can be made on the attachment specifications: First an offset of zero can be omitted. Thus, the following two lines are equivalent:

\$b–**form**(**–top**=[*\$a*,0], **–right**=['%100',0]);

\$b–**form**(**–top**=[*\$a*], **–right**='%100');

In the second case, when the anchor point is omitted, the offset must be given. A default anchor point is chosen according to the value of the offset. If the anchor point is or positive, the default anchor point %0 is used; thus, "*\$b*–**form**(**–top**=15)" attaches the top edge of *\$b* to a position 15 pixels below the top edge of the master window. If the anchor point is "**–0**" or negative, the default anchor point **%100** is used; thus, "*\$a*–**form**(**–right**=–2)" attaches the right edge of *\$a* to a position 2 pixels to the left of the master window's right edge. An further example below shows a method with its equivalent abbreviation.

\$b–**form**(**–top**=['%0',10], **–bottom**=['%100',0]);

\$b–**form**(**–top**=10, **–bottom**=–0);

USING SPRINGS

To be written.

ALGORITHM OF FORM

form starts with any slave in the list of slaves of the master window. Then it tries to determine the position of each side of the slave.

If the attachment of a side of the slave is grid attachment, the position of the side is readily determined.

If the attachment of this side is **none**, then form tries to determine the position of the opposite side first, and then use the position of the opposite side and the natural size of the slave to determine the position of this side.

If the attachment is opposite or parallel widget attachments, then form tries to determine the positions of the other widget first, and then use the positions of the other widget and the natural size of the slave determine the position of this side. This recursive algorithmis carried on until the positions of all slaves are determined.

CIRCULAR DEPENDENCY

The algorithm of form will fail if a circular dependency exists in the attachments of the slaves. For example:

\$c–**form**(**–left**=*\$b);*

\$b–**form**(**–right**=*\$c);*

In this example, the position of the left side of *$b* depends on the right side of *$c*, which in turn depends on the left side of *$b*.

When a circular dependency is discovered during the execution of the form algorithm, form will generate a background error and the geometry of the slaves are undefined (and will be arbitrary). Notice that form only executes the algorithm when the specification of the slaves' attachments is complete. Therefore, it allows intermediate states of circular dependency during the specification of the slaves' attachments. Also, unlike the Motif Form manager widget, form defines circular dependency as ''*dependency in the same dimension*''. Therefore, the following code fragment will does not have circular dependency because the two widgets do not depend on each other in the same dimension (*$b* depends *$c* in the horizontal dimension and *$c* depends on *$b* in the vertical dimension):

$b*−**form**(**−left**=*$c);

$c*−**form**(**−top**=*$b);

BUGS

Springs have not been fully implemented yet.

SEE ALSO

Tk::grid\Tk::grid Tk::pack\Tk::pack Tk::place\Tk::place

KEYWORDS

geometry manager, form, attachment, spring, propagation, size, pack, tix, master, slave

NAME

Tk::Frame – Create and manipulate Frame widgets

=for category Tk Widget Classes

SYNOPSIS

$frame = *$parent*->**Frame**(*?options?*);

STANDARD OPTIONS

–borderwidth **–highlightbackground** **–highlightthickness** **–takefocus** **–class**
–highlightcolor **–relief** **–cursor**

See *Tk::options* for details of the standard options.

WIDGET–SPECIFIC OPTIONS

Name: **background**
Class: **Background**
Switch: **–background**

This option is the same as the standard **background** option except that its value may also be specified as an undefined value. In this case, the widget will display no background or border, and no colors will be consumed from its colormap for its background and border.

Name: **colormap**
Class: **Colormap**
Switch: **–colormap**

Specifies a colormap to use for the window. The value may be either **new**, in which case a new colormap is created for the window and its children, or the name of another window (which must be on the same screen and have the same visual as `$widget)`, in which case the new window will use the colormap from the specified window. If the **colormap** option is not specified, the new window uses the same colormap as its parent. This option may not be changed with the **configure** method.

Name: **container**
Class: **Container**
Switch: **–container**

The value must be a boolean. If true, it means that this window will be used as a container in which some other application will be embedded (for example, a Tk toplevel can be embedded using the **–use** option). The window will support the appropriate window manager protocols for things like geometry requests. The window should not have any children of its own in this application. This option may not be changed with the **configure** method.

Name: **height**
Class: **Height**
Switch: **–height**

Specifies the desired height for the window in any of the forms acceptable to **Tk_GetPixels**. If this option is less than or equal to zero then the window will not request any size at all.

Name: **visual**
Class: **Visual**
Switch: **–visual**

Specifies visual information for the new window in any of the forms accepted by **Tk_GetVisual**. If this option is not specified, the new window will use the same visual as its parent. The **visual** option may not be modified with the **configure** method.

Name: **width**
Class: **Width**

Switch: **–width**

Specifies the desired width for the window in any of the forms acceptable to **Tk_GetPixels**. If this option is less than or equal to zero then the window will not request any size at all.

DESCRIPTION

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

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

WIDGET METHODS

The **Frame** method creates a widget object. This object supports the **configure** and **cget** methods described in *Tk::options* which can be used to enquire and modify the options described above. The widget also inherits all the methods provided by the generic *Tk::Widget\Tk::Widget* class.

BINDINGS

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

KEYWORDS

frame, widget

NAME

getOpenFile, getSaveFile – pop up a dialog box for the user to select a file to open or save.

=for category Popups and Dialogs

SYNOPSIS

$widget->**getOpenFile**(*?–option=>value, ...?*)

$widget->**getSaveFile**(*?–option=>value, ...?*)

DESCRIPTION

The methods **getOpenFile** and **getSaveFile** pop up a dialog box for the user to select a file to open or save.

The **getOpenFile** method is usually associated with the **Open** command in the **File** menu. Its purpose is for the user to select an existing file *only*. If the user enters an non–existent file, the dialog box gives the user an error prompt and requires the user to give an alternative selection. If an application allows the user to create new files, it should do so by providing a separate **New** menu command.

The **getSaveFile** method is usually associated with the **Save** as command in the **File** menu. If the user enters a file that already exists, the dialog box prompts the user for confirmation whether the existing file should be overwritten or not.

If the user selects a file, both **getOpenFile** and **getSaveFile** return the full pathname of this file. If the user cancels the operation, both commands return an undefined value.

The following *option–value* pairs are possible as command line arguments to these two commands:

–defaultextension => *extension*

Specifies a string that will be appended to the filename if the user enters a filename without an extension. The default value is the empty string, which means no extension will be appended to the filename in any case. This option is ignored on the Macintosh platform, which does not require extensions to filenames.

–filetypes => [*filePattern* ?, ...?]

If a **File types** listbox exists in the file dialog on the particular platform, this option gives the *filetype*s in this listbox. When the user choose a filetype in the listbox, only the files of that type are listed. If this option is unspecified, or if it is set to the empty list, or if the **File types** listbox is not supported by the particular platform then all files are listed regardless of their types. See *"SPECIFYING FILE PATTERNS"* below for a discussion on the contents of *filePattern*s.

–initialdir => *directory*

Specifies that the files in *directory* should be displayed when the dialog pops up. If this parameter is not specified, then the files in the current working directory are displayed. This option may not always work on the Macintosh. This is not a bug. Rather, the *General Controls* control panel on the Mac allows the end user to override the application default directory.

–initialfile => *filename*

Specifies a filename to be displayed in the dialog when it pops up. This option is ignored by the **getOpenFile** method.

–title => *titleString*

Specifies a string to display as the title of the dialog box. If this option is not specified, then a default title is displayed. This option is ignored on the Macintosh platform.

SPECIFYING FILE PATTERNS

The *filePattern*s given by the **–filetypes** option are a list of file patterns. Each file pattern is a list of the form

```
typeName [extension ?extension ...?] ?[macType ?macType ...?]?
```

typeName is the name of the file type described by this file pattern and is the text string that appears in the

File types listbox. *extension* is a file extension for this file pattern. *macType* is a four−character Macintosh file type. The list of *macType*s is optional and may be omitted for applications that do not need to execute on the Macintosh platform.

Several file patterns may have the same *typeName,* in which case they refer to the same file type and share the same entry in the listbox. When the user selects an entry in the listbox, all the files that match at least one of the file patterns corresponding to that entry are listed. Usually, each file pattern corresponds to a distinct type of file. The use of more than one file patterns for one type of file is necessary on the Macintosh platform only.

On the Macintosh platform, a file matches a file pattern if its name matches at least one of the *extension*(s) AND it belongs to at least one of the *macType*(s) of the file pattern. For example, the **C Source Files** file pattern in the sample code matches with files that have a **\.c** extension AND belong to the *macType* **TEXT**. To use the OR rule instead, you can use two file patterns, one with the *extensions* only and the other with the *macType* only. The **GIF Files** file type in the sample code matches files that EITHER have a **\.gif** extension OR belong to the *macType* **GIFF**.

On the Unix and Windows platforms, a file matches a file pattern if its name matches at at least one of the *extension*(s) of the file pattern. The *macType*s are ignored.

SPECIFYING EXTENSIONS

On the Unix and Macintosh platforms, extensions are matched using glob−style pattern matching. On the Windows platforms, extensions are matched by the underlying operating system. The types of possible extensions are: (1) the special extension * matches any file; (2) the special extension "" matches any files that do not have an extension (i.e., the filename contains no full stop character); (3) any character string that does not contain any wild card characters (* and ?).

Due to the different pattern matching rules on the various platforms, to ensure portability, wild card characters are not allowed in the extensions, except as in the special extension *. Extensions without a full stop character (e.g, ~) are allowed but may not work on all platforms.

EXAMPLE

```
my $types = [
    ['Text Files',       ['.txt', '.text']],
    ['TCL Scripts',      '.tcl'           ],
    ['C Source Files',   '.c',      'TEXT'],
    ['GIF Files',        '.gif',          ],
    ['GIF Files',        '',        'GIFF'],
    ['All Files',        '*',             ],
];
my $filename = $widget->getOpenFile(-filetypes=>$types);

if ($filename ne "") {
    # Open the file ...
}
```

SEE ALSO

Tk::FBox\Tk::FBox, *Tk::FileSelect\Tk::FileSelect*

KEYWORDS

file selection dialog

NAME

grab − Confine pointer and keyboard events to a window sub−tree

=for category User Interaction

SYNOPSIS

$widget−>**grab**

$widget−>**grab***Option*

DESCRIPTION

This set of methods implement 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 *$widget's* tree, button presses and releases and mouse motion events are reported to *$widget,* 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** method 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** methods take any of the following forms:

$widget−>**grabCurrent**

Returns the current grab window in this application for *$widget's* display, or an empty string if there is no such window.

$widget−>**grabs**

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.

Not implemented yet!

$widget−>**grabRelease**

Releases the grab on *$widget* if there is one, otherwise does nothing. Returns an empty string.

$widget−>**grab**

Sets a local grab on *$widget*. If a grab was already in effect for this application on *$widget's* display then it is automatically released. If there is already a local grab on *$widget,* then the command does nothing. Returns an empty string.

$widget->**grabGlobal**

Sets a global grab on *$widget*. If a grab was already in effect for this application on *$widget*'*s* display then it is automatically released. If there is already a global grab on *$widget*, then the command does nothing. Returns an empty string.

$widget->**grabStatus**

Returns **none** if no grab is currently set on *$widget*, **local** if a local grab is set on *$widget*, and **global** if a global grab is set.

BUGS

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

If a single process is managing several different Tk applications, only one of those applications can have a local grab for a given display at any given time. If the applications are in different processes, this restriction doesn't exist.

KEYWORDS

grab, keyboard events, pointer events, window

NAME

Tk::grid – Geometry manager that arranges widgets in a grid

=for category Tk Geometry Management

SYNOPSIS

$widget->**grid***?(?widget ...,? ?arg ?...?)?*

$widget->**grid***Option?(arg ?,arg ...?)?*

DESCRIPTION

The **grid** method is used to communicate with the grid geometry manager that arranges widgets in rows and columns inside of another window, called the geometry master (or master window). The **grid** method can have any of several forms, depending on the *option* argument:

$slave*->**grid***(?$slave, ...??, *options*?)

The arguments consist of the optional references to more slave windows followed by pairs of arguments that specify how to manage the slaves. The characters **–**, **x** and **^**, can be specified instead of a window reference to alter the default location of a *$slave*, as described in *"RELATIVE PLACEMENT"*, below.

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.

The following options are supported:

–column = *n*

Insert the *$slave* so that it occupies the *n*th column in the grid. Column numbers start with 0. If this option is not supplied, then the *$slave* is arranged just to the right of previous slave specified on this call to **grid**, or column "0" if it is the first slave. For each **x** that immediately precedes the *$slave*, the column position is incremented by one. Thus the **x** represents a blank column for this row in the grid.

–columnspan = *n*

Insert the slave so that it occupies *n* columns in the grid. The default is one column, unless the window name is followed by a **–**, in which case the columnspan is incremented once for each immediately following **–**.

–in = *$other*

Insert the slave(s) in the master window given by *$other*. The default is the first slave's parent window.

–ipadx = *amount*

The *amount* specifies how much horizontal internal padding to leave on each side of the slave(s). This is space is added inside the slave(s) border. The *amount* must be a valid screen distance, such as **2** or **'.5c'**. It defaults to 0.

–ipady = *amount*

The *amount* specifies how much vertical internal padding to leave on on the top and bottom of the slave(s). This space is added inside the slave(s) border. The *amount* defaults to 0.

–padx = *amount*

The *amount* specifies how much horizontal external padding to leave on each side of the slave(s), in screen units. The *amount* defaults to 0. This space is added outside the slave(s) border.

–pady = *amount*

The *amount* specifies how much vertical external padding to leave on the top and bottom of the slave(s), in screen units. The *amount* defaults to 0. This space is added outside the slave(s) border.

–row = *n* Insert the slave so that it occupies the *n*th row in the grid. Row numbers start with 0. If this option is not supplied, then the slave is arranged on the same row as the previous slave specified on this call to **grid**, or the first unoccupied row if this is the first slave.

–rowspan = *n*

Insert the slave so that it occupies *n* rows in the grid. The default is one row. If the next **grid** method contains **^** characters instead of *$slaves* that line up with the columns of this *$slave*, then the **rowspan** of this *$slave* is extended by one.

–sticky = *style*

If a slave's cell is larger than its requested dimensions, this option may be used to position (or stretch) the slave within its cell. *Style* is a string that contains zero or more of the characters **n**, **s**, **e** or **w**. The string can optionally contain spaces or commas, but they are ignored. Each letter refers to a side (north, south, east, or west) that the slave will "stick" to. If both **n** and **s** (or **e** and **w**) are specified, the slave will be stretched to fill the entire height (or width) of its cavity. The **sticky** option subsumes the combination of **–anchor** and **–fill** that is used by *pack\|Tk::pack*. The default is **''**, which causes the slave to be centered in its cavity, at its requested size.

$master–>**gridBbox***(?column, row,*? ?*column2, row2*?)

With no arguments, the bounding box (in pixels) of the grid is returned. The return value consists of 4 integers. The first two are the pixel offset from the master window (x then y) of the top–left corner of the grid, and the second two integers are the width and height of the grid, also in pixels. If a single *column* and *row* is specified on the command line, then the bounding box for that cell is returned, where the top left cell is numbered from zero. If both *column* and *row* arguments are specified, then the bounding box spanning the rows and columns indicated is returned.

$master–>**gridColumnconfigure***(index?, –option=value, ...*?)

Query or set the column properties of the *index* column of the geometry master, *$master*. The valid options are **–minsize**, **–weight** and **–pad**. If one or more options are provided, then *index* may be given as a list of column indices to which the configuration options will operate on. The **–minsize** option sets the minimum size, in screen units, that will be permitted for this column. The **–weight** option (an integer value) sets the relative weight for apportioning any extra spaces among columns. A weight of zero (0) indicates the column will not deviate from its requested size. A column whose weight is two will grow at twice the rate as a column of weight one when extra space is allocated to the layout. The **–pad** option specifies the number of screen units that will be added to the largest window contained completely in that column when the grid geometry manager requests a size from the containing window. If only an option is specified, with no value, the current value of that option is returned. If only the master window and index is specified, all the current settings are returned in an list of "–option value" pairs.

$slave*–>**gridConfigure***(?$slave, ...?, *options*?)

The same as **grid** method.

$slave–>**gridForget***?($slave, ...*)?

Removes each of the *$slaves* from grid for its master and unmaps their windows. The slaves will no longer be managed by the grid geometry manager. The configuration options for that window are forgotten, so that if the slave is managed once more by the grid geometry manager, the initial default settings are used.

$slave->**gridInfo**

Returns a list whose elements are the current configuration state of the slave given by *$slave* in the same option−value form that might be specified to **gridConfigure**. The first two elements of the list are ‘‘**−in**=*$master*’’ where *$master* is the slave‘s master.

$master->**gridLocation***(x, y)*

Given *x* and *y* values in screen units relative to the master window, the column and row number at that *x* and *y* location is returned. For locations that are above or to the left of the grid, **−1** is returned.

$master->**gridPropagate***?(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 method returns or **1** to indicate whether propagation is currently enabled for *$master*. Propagation is enabled by default.

$master->**gridRowconfigure***(index?, −option=value, ...*?)

Query or set the row properties of the *index* row of the geometry master, *$master*. The valid options are **−minsize**, **−weight** and **−pad**. If one or more options are provided, then *index* may be given as a list of row indeces to which the configuration options will operate on. The **−minsize** option sets the minimum size, in screen units, that will be permitted for this row. The **−weight** option (an integer value) sets the relative weight for apportioning any extra spaces among rows. A weight of zero (0) indicates the row will not deviate from its requested size. A row whose weight is two will grow at twice the rate as a row of weight one when extra space is allocated to the layout. The **−pad** option specifies the number of screen units that will be added to the largest window contained completely in that row when the grid geometry manager requests a size from the containing window. If only an option is specified, with no value, the current value of that option is returned. If only the master window and index is specified, all the current settings are returned in an list of "option−value" pairs.

$slave->**gridRemove***?($slave, ...*)?

Removes each of the *$slaves* from grid for its master and unmaps their windows. The slaves will no longer be managed by the grid geometry manager. However, the configuration options for that window are remembered, so that if the slave is managed once more by the grid geometry manager, the previous values are retained.

$master->**gridSize**

Returns the size of the grid (in columns then rows) for *$master*. The size is determined either by the *$slave* occupying the largest row or column, or the largest column or row with a **−minsize**, **−weight**, or **−pad** that is non−zero.

$master->**gridSlaves***?(−option=value*)?

If no options are supplied, a list of all of the slaves in *$master* are returned, most recently manages first. *−option* can be either **−row** or **−column** which causes only the slaves in the row (or column) specified by *value* to be returned.

RELATIVE PLACEMENT

The **grid** method contains a limited set of capabilities that permit layouts to be created without specifying the row and column information for each slave. This permits slaves to be rearranged, added, or removed without the need to explicitly specify row and column information. When no column or row information is specified for a *$slave*, default values are chosen for **−column**, **−row**, **−columnspan** and **−rowspan** at the time the *$slave* is managed. The values are chosen based upon the current layout of the grid, the position of the *$slave* relative to other *$slaves* in the same grid method, and the presence of the characters **−**, **^**, and **^** in **grid** method where *$slave* names are normally expected.

- **–** This increases the columnspan of the *$slave* to the left. Several **–**'s in a row will successively increase the columnspan. A **–** may not follow a **^** or a **x**.
- **x** This leaves an empty column between the *$slave* on the left and the *$slave* on the right.
- **^** This extends the **–rowspan** of the *$slave* above the **^**'s in the grid. The number of **^**'s in a row must match the number of columns spanned by the *$slave* above it.

THE GRID ALGORITHM

The grid geometry manager lays out its slaves in three steps. In the first step, the minimum size needed to fit all of the slaves is computed, then (if propagation is turned on), a request is made of the master window to become that size. In the second step, the requested size is compared against the actual size of the master. If the sizes are different, then space is added to or taken away from the layout as needed. For the final step, each slave is positioned in its row(s) and column(s) based on the setting of its *sticky* flag.

To compute the minimum size of a layout, the grid geometry manager first looks at all slaves whose columnspan and rowspan values are one, and computes the nominal size of each row or column to be either the *minsize* for that row or column, or the sum of the *pad*ding plus the size of the largest slave, whichever is greater. Then the slaves whose rowspans or columnspans are greater than one are examined. If a group of rows or columns need to be increased in size in order to accommodate these slaves, then extra space is added to each row or column in the group according to its *weight*. For each group whose weights are all zero, the additional space is apportioned equally.

For masters whose size is larger than the requested layout, the additional space is apportioned according to the row and column weights. If all of the weights are zero, the layout is centered within its master. For masters whose size is smaller than the requested layout, space is taken away from columns and rows according to their weights. However, once a column or row shrinks to its minsize, its weight is taken to be zero. If more space needs to be removed from a layout than would be permitted, as when all the rows or columns are at there minimum sizes, the layout is clipped on the bottom and right.

GEOMETRY PROPAGATION

The grid geometry manager normally computes how large a master must be to just exactly meet the needs of its slaves, and it sets the requested width and height of the master to these dimensions. This causes geometry information to propagate up through a window hierarchy to a top–level window so that the entire sub–tree sizes itself to fit the needs of the leaf windows. However, the **gridPropagate** method may be used to turn off propagation for one or more masters. If propagation is disabled then grid will not set the requested width and height of the master window. This may be useful if, for example, you wish for a master window to have a fixed size that you specify.

RESTRICTIONS ON MASTER WINDOWS

The master for each slave must either be the slave's parent (the default) or a descendant of the slave's parent. This restriction is necessary to guarantee that the slave can be placed over any part of its master that is visible without danger of the slave being clipped by its parent. In addition, all slaves in one call to **grid** must have the same master.

STACKING ORDER

If the master for a slave is not its parent then you must make sure that the slave is higher in the stacking order than the master. Otherwise the master will obscure the slave and it will appear as if the slave hasn't been managed correctly. The easiest way to make sure the slave is higher than the master is to create the master window first: the most recently created window will be highest in the stacking order.

CREDITS

The **grid** method is based on ideas taken from the *GridBag* geometry manager written by Doug. Stein, and the **blt_table** geometry manager, written by George Howlett.

SEE ALSO

Tk::form\Tk::form Tk::pack\Tk::pack Tk::place\Tk::place

KEYWORDS

geometry manager, location, grid, cell, propagation, size, pack, master, slave

NAME

Tk::HList – Create and manipulate Tix Hierarchial List widgets

=for category Tk Widget Classes

SYNOPSIS

$hlist = *$parent*->**HList***(?options?);*

STANDARD OPTIONS

–background **–borderwidth** **–cursor** **–exportselection** **–foreground** **–font** **–height** **–highlightcolor** **–highlightthickness** **–relief** **–selectbackground** **–selectforeground** **–xscrollcommand** **–yscrollcommand** **–width**

See *Tk::options* for details of the standard options.

WIDGET–SPECIFIC OPTIONS

Name: **browsecmd**
Class: **BrowseCmd**
Switch: **–browsecmd**

Specifies a perl/Tk *callback|Tk::callbacks* to be executed when the user browses through the entries in the HList widget.

Name: **columns**
Class: **Columns**
Switch: **–columns**

Specifies the number of columns in this HList widget. This option can only be set during the creation of the HList widget and cannot be changed subsequently.

Name: **command**
Class: **Command**
Switch: **–command**

Specifies the perl/Tk *callback|Tk::callbacks* to be executed when the user invokes a list entry in the HList widget. Normally the user invokes a list entry by double–clicking it or pressing the Return key.

Name: **drawBranch**
Class: **DrawBranch**
Switch: **–drawbranch**

A Boolean value to specify whether branch line should be drawn to connect list entries to their parents.

Name: **foreground**
Class: **Foreground**
Switch: **–foreground**
Alias: **–fg**
[OBSOLETE] Specifies the default foreground color for the list entries.

Name: **gap**
Class: **Gap**
Switch: **–gap**

[OBSOLETE] The default distance between the bitmap/image and the text in list entries.

Name: **header**
Class: **Header**
Switch: **–header**

A Boolean value specifying whether headers should be displayed for this HList widget (see the **header** method below).

Name: **height**
Class: **Height**
Switch: **–height**

Specifies the desired height for the window in number of characters.

Name: **indent**
Class: **Indent**
Switch: **–indent**

Specifies the amount of horizontal indentation between a list entry and its children. Must be a valid screen distance value.

Name: **indicator**
Class: **Indicator**
Switch: **–indicator**

Specifies whether the indicators should be displayed inside the HList widget. See the **indicator** method below.

Name: **indicatorCmd**
Class: **IndicatorCmd**
Switch: **–indicatorcmd**

Specifies a perl/Tk *callback|Tk::callbacks* to be executed when the user manipulates the indicator of an HList entry. The **–indicatorcmd** is triggered when the user press or releases the mouse button over the indicator in an HList entry. By default the perl/Tk **callback** specified by **–indicatorcmd** is executed with two additional arguments, the entryPath of the entry whose indicator has been triggered and additional information about the event. The additional information is one of the following strings: **<Arm>**, **<Disarm>**, and **<Activate>**.

Name: **itemType**
Class: **ItemType**
Switch: **–itemtype**

Specifies the default type of display item for this HList widget. When you call the **itemCreate**, **add** and **addchild** methods, display items of this type will be created if the **–itemtype** option is not specified .

Name: **padX**
Class: **Pad**
Switch: **–padx**

[OBSOLETE] The default horizontal padding for list entries.

Name: **padY**
Class: **Pad**
Switch: **–padx**

[OBSOLETE] The default vertical padding for list entries.

Name: **selectBackground**
Class: **SelectBackground**
Switch: **–selectbackground**

Specifies the background color for the selected list entries.

Name: **selectBorderWidth**
Class: **BorderWidth**
Switch: **–selectborderwidth**

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**.

Name: **selectForeground**
Class: **SelectForeground**
Switch: **–selectforeground**

Specifies the foreground color for the selected list entries.

Name: **selectMode**
Class: **SelectMode**
Switch: **–selectmode**

Specifies one of several styles for manipulating the selection. The value of the option may be arbitrary, but the default bindings expect it to be either **single**, **browse**, **multiple**, or **extended**; the default value is **single**.

Name: **sizeCmd**
Class: **SizeCmd**
Switch: **–sizecmd**

Specifies a perl/Tk *callback*\|*Tk::callbacks* to be called whenever the HList widget changes its size. This method can be useful to implement ‘‘*user scroll bars when needed*‘’ features.

Name: **separator**
Class: **Separator**
Switch: **–separator**

Specifies the character to used as the separator character when intepreting the path–names of list entries. By default the character "." is used.

Name: **width**
Class: **Width**
Switch: **–width**

Specifies the desired width for the window in characters.

DESCRIPTION

The **HList** method creates a new window (given by the `$widget` argument) and makes it into a HList widget. Additional options, described above, may be specified on the command line or in the option database to configure aspects of the HList widget such as its cursor and relief.

The HList widget can be used to display any data that have a hierarchical structure, for example, file system directory trees. The list entries are indented and connected by branch lines according to their places in the hierachy.

Each list entry is identified by an **entryPath**. The entryPath is a sequence of **entry names** separated by the separator charactor (specified by the **–separator** option). An **entry name** can be any string that does not contain the separator charactor, or it can be the a string that contains only one separator charactor.

For example, when "." is used as the separator charactor, "one.two.three" is the entryPath for a list entry whose parent is "one.two", whose parent is "one", which is a toplevel entry (has no parents).

Another examples: ".two.three" is the entryPath for a list entry whose parent is ".two", whose parent is ".", which is a toplevel entry.

DISPLAY ITEMS

Each list entry in an HList widget is associated with a **display** item. The display item determines what visual information should be displayed for this list entry. Please see *Tk::DItem* for a list of all display items. When a list entry is created by the **itemCreate**, **add** or **addchild** widget methods, the type of its display item is determined by the **–itemtype** option passed to these methods. If the **–itemtype** is omitted, then by default the type specified by this HList widget‘s **–itemtype** option is used.

WIDGET METHODS

The **HList** method creates a widget object. This object supports the **configure** and **cget** methods described in *Tk::options* which can be used to enquire and modify the options described above. The widget also inherits all the methods provided by the generic *Tk::Widget|Tk::Widget* class.

The following additional methods are available HList widgets:

`$hlist`**->add**(*$entryPath* ?,*option*=>*value*, ...?)

Creates a new list entry with the pathname `$entryPath`. A list entry must be created after its parent is created (unless this entry is a top–level entry, which has no parent). See also *"BUGS"* below. This method returns the entryPath of the newly created list entry. The following configuration options can be given to configure the list entry:

–at => *position*

Insert the new list at the position given by *position*. *position* must be a valid integer. The position indicates the first position, **1** indicates the second position, and so on.

–after => *afterWhich*

Insert the new list entry after the entry identified by *afterWhich*. *afterWhich* must be a valid list entry and it mush have the same parent as the new list entry

–before => *beforeWhich*

Insert the new list entry before the entry identified by *beforeWhich*. *beforeWhich* must be a valid list entry and it mush have the same parent as the new list entry

–data => *string*

Specifies a string to associate with this list entry. This string can be queried by the **info** method. The application programmer can use the **–data** option to associate the list entry with the data it represents.

–itemtype => *type*

Specifies the type of display item to be display for the new list entry. **type** must be a valid display item type. Currently the available display item types are **imagetext**, **text**, and `$widget`. If this option is not specified, then by default the type specified by this HList widget's **–itemtype** option is used.

–state => *state*

Specifies whether this entry can be selected or invoked by the user. Must be either **normal** or **disabled**.

The **add** method accepts additional configuration options to configure the display item associated with this list entry. The set of additional configuration options depends on the type of the display item given by the **–itemtype** option. Please see *Tk::DItem* for a list of the configuration options for each of the display item types.

`$hlist`**->addchild**(*$parentPath, ?option, value, ..., ?*)

Adds a new child entry to the children list of the list entry identified by `$parentPath`. Or, if `$parentPath` is set to be the empty string, then creates a new toplevel entry. The name of the new list entry will be a unique name automatically generated by the HList widget. Usually if `$parentPath` is **foo**, then the entryPath of the new entry will be **foo.0**, **foo.1**, ... etc. This method returns the entryPath of the newly created list entry. *option* can be any option for the **add** method. See also *"BUGS"* below.

`$hlist`**->anchorSet**(*$entryPath)*

Sets the anchor to the list entry identified by `$entryPath`. The anchor is the end of the selection that is fixed while the user is dragging out a selection with the mouse.

$hlist->**anchorClear**

Removes the anchor, if any, from this HList widget. This only removes the surrounding highlights of the anchor entry and does not affect its selection status.

$hlist*->**columnWidth***($col?, –char?, ?*width*?)

Querys or sets the width of a the column *$col* in the HList widget. The value of *$col* is zero–based: 0 stands for the first column, 1 stands for the second, and so on. If no further parameters are given, returns the current width of this column (in number of pixels). Additional parameters can be given to set the width of this column:

$hlist*->**columnWidth***($col, '')

An empty string indicates that the width of the column should be just wide enough to display the widest element in this column. In this case, the width of this column may change as a result of the elements in this column changing their sizes.

$hlist*->**columnWidth***($col, width)

width must be in a form accepted by **Tk_GetPixels**.

$hlist*->**columnWidth***($col, **–char,** *nChars*)

The width is set to be the average width occupied by *nChars* number of characters of the font specified by the **–font** option of this HList widget.

$hlist->**delete***(option, $entryPath)*

Delete one or more list entries. *option* may be one of the following:

all Delete all entries in the HList. In this case the *$entryPath* does not need to be specified.

entry Delete the entry specified by *$entryPath* and all its offsprings, if any.

offsprings

Delete all the offsprings, if any, of the entry specified by *$entryPath*. However, *$entryPath* itself is not deleted.

siblings Delete all the list entries that share the same parent with the entry specified by *$entryPath*. However, *$entryPath* itself is not deleted.

$hlist*->**dragsiteSet***($entryPath)

Sets the dragsite to the list entry identified by *$entryPath*. The dragsite is used to indicate the source of a drag–and–drop action. Currently drag–and–drop functionality has not been implemented in Tix yet.

$hlist->**dragsiteClear**

Remove the dragsite, if any, from the this HList widget. This only removes the surrounding highlights of the dragsite entry and does not affect its selection status.

$hlist*->**dropsiteSet***($entryPath)

Sets the dropsite to the list entry identified by *$entryPath*. The dropsite is used to indicate the target of a drag–and–drop action. Currently drag–and–drop functionality has not been implemented in Tix yet.

$hlist->**dropsiteClear**

Remove the dropsite, if any, from the this HList widget. This only removes the surrounding highlights of the dropsite entry and does not affect its selection status.

$hlist*->**entrycget***($entryPath, option)

Returns the current value of the configuration option given by *option* for the entry indentfied by *$entryPath*. *Option* may have any of the values accepted by the **add** method.

$hlist*->**entryconfigure**(*$entryPath **?**,*option***?**, **?***value***=>***option***, ...?**)

Query or modify the configuration options of the list entry indentfied by *$entryPath*. If no *option* is specified, returns a list describing all of the available options for *$entryPath* (see *Tk::options* for information on the format of this list.) If *option* is specified with no *value*, then the method 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 method modifies the given option(s) to have the given value(s); in this case the method returns an empty string. *Option* may have any of the values accepted by the **add** or **addchild** method. The exact set of options depends on the value of the **−itemtype** option passed to the the **add** or **addchild** method when this list entry is created.

$hlist->**header**(*option, $col* ?,*args*, ...?)

Manipulates the header items of this HList widget. If the **−header** option of this HList widget is set to true, then a header item is displayed at the top of each column. The *$col* argument for this method must be a valid integer. 0 indicates the first column, 1 the second column, ... and so on. This method supports the following options:

$hlist->**header**(**cget**, *$col, option*)

If the *$col*−*th* column has a header display item, returns the value of the specified *option* of the header item. If the header doesn‘t exist, returns an error.

$hlist*->**header**(**configure,** *$col, ?*option*?, *?value, option, value, ...*?)

Query or modify the configuration options of the header display item of the *$col*−*th* column. The header item must exist, or an error will result. If no *option* is specified, returns a list describing all of the available options for the header display item (see *Tk::options* for information on the format of this list.) If *option* is specified with no *value*, then the method 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 method modifies the given option(s) to have the given value(s); in this case the method returns an empty string. *Option* may have any of the values accepted by the **header create** method. The exact set of options depends on the value of the **−itemtype** option passed to the the **header** create method when this display item was created.

$hlist*->**header**(**create,** *$col, ?*−itemtype type*? ?*option value ...*?

Creates a new display item as the header for the *$col*−*th* column. See also *"BUGS"* below. If an header display item already exists for this column, it will be replaced by the new item. An optional parameter *−itemtype* can be used to specify what type of display item should be created. If the *−itemtype* is not given, then by default the type specified by this HList widget‘s **−itemtype** option is used. Additional parameters, in *option*−*value* pairs, can be passed to configure the appearance of the display item. Each *option*−*value* pair must be a valid option for this type of display item or one of the following:

−borderwidth => *color*

Specifies the border width of this header item.

−headerbackground => *color*

Specifies the background color of this header item.

−relief => *type* Specifies the relief type of the border of this header item.

$hlist*->**header**(**delete,** *$col)

Deletes the header display item for the *$col*−*th* column.

$hlist*->**header**(**exists,** *$col)

Return true if an header display item exists for the *$col*−*th* column; return false otherwise.

$hlist->**header***(***size***, $entryPath)*

If an header display item exists for the *$col-th* column , returns its size in pixels in a two element list *(width, height)*; returns an error if the header display item does not exist.

$hlist->**hide***(option* ?,*$entryPath?)*

Makes some of entries invisible without deleting them. *Option* can be one of the following:

entry Hides the list entry identified by *$entryPath*.

Currently only the **entry** option is supported. Other options will be added in the next release.

$hlist->**indicator***(option, $entryPath, ?args, ...*?)

Manipulates the indicator on the list entries. An indicator is usually a small display item (such as an image) that is displayed to the left to an entry to indicate the status of the entry. For example, it may be used to indicate whether a directory is opened or closed. *Option* can be one of the following:

$hlist->**indicator***(***cget***, $entryPath, option*)

If the list entry given by *$entryPath* has an indicator, returns the value of the specified *option* of the indicator. If the indicator doesn't exist, returns an error.

$hlist->**indicator***(***configure***, $entryPath, ?option*?, *?value, option, value, ...*?)

Query or modify the configuration options of the indicator display item of the entry specified by *$entryPath*. The indicator item must exist, or an error will result. If no *option* is specified, returns a list describing all of the available options for the indicator display item (see *Tk::options* for information on the format of this list). If *option* is specified with no *value*, then the method 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 method modifies the given option(s) to have the given value(s); in this case the method returns an empty string. *Option* may have any of the values accepted by the **indicator create** method. The exact set of options depends on the value of the **–itemtype** option passed to the the **indicator create** method when this display item was created.

$hlist*->**indicator***(***create,** *$entryPath, ?, *–itemtype type*? ?*option value ...*?)

Creates a new display item as the indicator for the entry specified by *$entryPath*. If an indicator display item already exists for this entry, it will be replaced by the new item. An optional parameter *–itemtype* can be used to specify what type of display item should be created. If the *–itemtype* is not given, then by default the type specified by this HList widget's **–itemtype** option is used. Additional parameters, in *option–value* pairs, can be passed to configure the appearance of the display item. Each *option–value* pair must be a valid option for this type of display item.

$hlist->**indicator***(***delete***, $entryPath)*

Deletes the indicator display item for the entry given by *$entryPath*.

$hlist->**indicator***(***exists***, $entryPath)*

Return true if an indicator display item exists for the entry given by *$entryPath;* return false otherwise.

$hlist->**indicator***(***size***, $entryPath)*

If an indicator display item exists for the entry given by *$entryPath*, returns its size in a two element list of the form {*width height*}; returns an error if the indicator display item does not exist.

$hlist->**info***(option, arg, ...*)

Query information about the HList widget. *option* can be one of the following:

$hlist–>**info***(***anchor***)*

Returns the entryPath of the current anchor, if any, of the HList widget. If the anchor is not set, returns the empty string.

*$hlist*–>**infoBbox***(**$entryPath**)*

Returns a list of four numbers describing the visible bounding box of the entry given *$entryPath*. The first two elements of the list give the x and y coordinates of the upper–left corner of the screen area covered by the entry (specified in pixels relative to the widget) and the last two elements give the lower–right corner of the area, in pixels. If no part of the entry given by index is visible on the screen then the result is an empty string; if the entry is partially visible, the result gives the only the visible area of the entry.

$hlist–>**info***(***children** ?,*$entryPath?)*

If *$entryPath* is given, returns a list of the entryPath's of its children entries. Otherwise returns a list of the toplevel entryPath's.

$hlist–>**info***(***data** ?,*$entryPath?)*

Returns the data associated with *$entryPath*.

$hlist–>**info***(***dragsite***)*

Returns the entryPath of the current dragsite, if any, of the HList widget. If the dragsite is not set, returns the empty string.

$hlist–>**info***(***dropsite***)*

Returns the entryPath of the current dropsite, if any, of the HList widget. If the dropsite is not set, returns the empty string.

$hlist–>**info***(***exists***, $entryPath)*

Returns a boolean value indicating whether the list entry *$entryPath* exists.

$hlist–>**info***(***hidden***, $entryPath)*

Returns a boolean value indicating whether the list entry **$entryPath** is hidden or not.

$hlist–>**info***(***next***, $entryPath)*

Returns the entryPath of the list entry, if any, immediately below this list entry. If this entry is already at the bottom of the HList widget, returns an empty string.

$hlist–>**info***(***parent***, $entryPath)*

Returns the name of the parent of the list entry identified by *$entryPath*. If *entryPath* is a toplevel list entry, returns the empty string.

$hlist–>**info***(***prev***, $entryPath)*

Returns the entryPath of the list entry, if any, immediately above this list entry. If this entry is already at the top of the HList widget, returns an empty string.

$hlist–>**info***(***selection***)*

Returns a list of selected entries in the HList widget. If no entries are selected, returns an empty string.

$hlist–>**item***(option, ?args, ...?)*

Creates and configures the display items at individual columns the entries. The form of additional of arguments depends on the choice of *option*:

$hlist*–>**itemCget***(**$entryPath, $col, option)

Returns the current value of the configure *option* of the display item at the column designated by *$col* of the entry specified by *$entryPath*.

$hlist*–>**itemConfigure***($entryPath, $col **?,***option***?,** *?value, option, value, ...***?)**

Query or modify the configuration options of the display item at the column designated by *$col* of the entry specified by *$entryPath*. If no *option* is specified, returns a list describing all of the available options for *$entryPath* (see *Tk::options* for information on the format of this list). If *option* is specified with no *value*, then the method 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 method modifies the given option(s) to have the given value(s); in this case the method returns an empty string. *Option* may have any of the values accepted by the **item** create method. The exact set of options depends on the value of the **–itemtype** option passed to the the **item** create method when this display item was created.

$hlist*–>**itemCreate***($entryPath, $col ?,*–itemtype*=>*type*? ?,*option value ...*?)

Creates a new display item at the column designated by *$col* of the entry specified by *$entryPath*. An optional parameter *–itemtype* can be used to specify what type of display items should be created. If the *–itemtype* is not specified, then by default the type specified by this HList widget's **–itemtype** option is used. Additional parameters, in *option*–*value* pairs, can be passed to configure the appearance of the display item. Each *option*– *value* pair must be a valid option for this type of display item.

$hlist*–>**itemDelete***($entryPath, $col)

Deletes the display item at the column designated by *$col* of the entry specified by *$entryPath*.

$hlist*–>**itemExists***($entryPath, $col)

Returns true if there is a display item at the column designated by *$col* of the entry specified by *$entryPath;* returns false otherwise.

$hlist–>**nearest***(y)*

$hlist–>**nearest***(y)* Given a y–coordinate within the HList window, this method returns the entryPath of the (visible) HList element nearest to that y–coordinate.

$hlist*–>**see***($entryPath)

Adjust the view in the HList so that the entry given by *$entryPath* is visible. If the entry is already visible then the method has no effect; if the entry is near one edge of the window then the HList scrolls to bring the element into view at the edge; otherwise the HList widget scrolls to center the entry.

$hlist–>**selection***(option, arg, ...)*
$hlist–>**selection***Option(arg, ...)*

This method is used to adjust the selection within a HList widget. It has several forms, depending on *option*:

$hlist–>**selectionClear***(?from?,* ?*to*?)

When no extra arguments are given, deselects all of the list entrie(s) in this HList widget. When only *from* is given, only the list entry identified by *from* is deselected. When both *from* and *to* are given, deselects all of the list entrie(s) between between *from* and *to*, inclusive, without affecting the selection state of elements outside that range.

$hlist–>**selectionGet**

This is an alias for the **infoSelection** method.

$hlist*–>**selectionIncludes***($entryPath)

Returns 1 if the list entry indicated by *$entryPath* is currently selected; returns 0 otherwise.

$hlist->**selectionSet***(from?, to***?)**

Selects all of the list entrie(s) between between *from* and *to*, inclusive, without affecting the selection state of entries outside that range. When only *from* is given, only the list entry identified by *from* is selected.

$hlist->**show***(option* ?,*$entryPath?)*

Show the entries that are hidden by the **hide** method, *option* can be one of the following:

entry Shows the list entry identified by *$entryPath*.

Currently only the **entry** option is supported. Other options will be added in future releases.

$hlist->**xview***(args)*

This method is used to query and change the horizontal position of the information in the widget's window. It can take any of the following forms:

$hlist->**xview**

Returns a list containing two elements. Each element is a real fraction between 0 and 1; together they describe the horizontal span that is visible in the window. For example, if the first element is .2 and the second element is .6, 20% of the HList entry is off−screen to the left, the middle 40% is visible in the window, and 40% of the entry is off−screen to the right. These are the same values passed to scrollbars via the **−xscrollcommand** option.

$hlist*->**xview***($entryPath)

Adjusts the view in the window so that the list entry identified by *$entryPath* is aligned to the left edge of the window.

$hlist->**xview***(***moveto** => *fraction*)

Adjusts the view in the window so that *fraction* of the total width of the HList is off−screen to the left. *fraction* must be a fraction between 0 and 1.

$hlist->**xview***(***scroll** => *number, what*)

This method shifts the view in the window left or right according to *number* and *what*. *Number* must be an integer. *What* must be either **units** or **pages** or an abbreviation of one of these. If *what* is **units**, the view adjusts left or right by *number* character units (the width of the character) on the display; if it is **pages** then the view adjusts by *number* screenfuls. If *number* is negative then characters farther to the left become visible; if it is positive then characters farther to the right become visible.

$hlist->**yview***(?args?)*

This method is used to query and change the vertical position of the entries in the widget's window. It can take any of the following forms:

$hlist->**yview**

Returns a list containing two elements, both of which are real fractions between 0 and 1. The first element gives the position of the list element at the top of the window, relative to the HList as a whole (0.5 means it is halfway through the HList, for example). The second element gives the position of the list entry just after the last one in the window, relative to the HList as a whole. These are the same values passed to scrollbars via the **−yscrollcommand** option.

$hlist*->**yview***($entryPath)

Adjusts the view in the window so that the list entry given by *$entryPath* is displayed at the top of the window.

$hlist->**yview***(***moveto** => *fraction*)

Adjusts the view in the window so that the list entry given by *fraction* appears at the top of the window. *Fraction* is a fraction between 0 and 1; 0 indicates the first entry in the HList,

0.33 indicates the entry one−third the way through the HList, and so on.

$hlist−>**yview**(**scroll** => *number, what*)

This method adjust the view in the window up or down according to *number* and *what*. *Number* must be an integer. *What* must be either **units** or **pages**. If *what* is **units**, the view adjusts up or down by *number* lines; if it is **pages** then the view adjusts by *number* screenfuls. If *number* is negative then earlier entries become visible; if it is positive then later entries become visible.

BINDINGS

[1] If the **−selectmode** is "browse", when the user drags the mouse pointer over the list entries, the entry under the pointer will be highlighted and the **−browsecmd** callback will be called with one parameter, the entryPath of the highlighted entry. Only one entry can be highlighted at a time. The **−command** callback will be called when the user double−clicks on a list entry.

[2] If the **−selectmode** is "single", the entries will only be highlighted by mouse <ButtonRelease−1> events. When a new list entry is highlighted, the **−browsecmd** callback will be called with one parameter indicating the highlighted list entry. The **−command** callback will be called when the user double−clicks on a list entry.

[3] If the **−selectmode** is "multiple", when the user drags the mouse pointer over the list entries, all the entries under the pointer will be highlighted. However, only a contiguous region of list entries can be selected. When the highlighted area is changed, the **−browsecmd** callback will be called with an undefined parameter. It is the responsibility of the **−browsecmd** callback to find out the exact highlighted selection in the HList. The **−command** callback will be called when the user double−clicks on a list entry.

[4] If the **−selectmode** is "extended", when the user drags the mouse pointer over the list entries, all the entries under the pointer will be highlighted. The user can also make disjointed selections using <Control−ButtonPress−1>. When the highlighted area is changed, the **−browsecmd** callback will be called with an undefined parameter. It is the responsibility of the **−browsecmd** callback to find out the exact highlighted selection in the HList. The **−command** callback will be called when the user double−clicks on a list entry.

[5] **Arrow key bindings:** <Up> arrow key moves the anchor point to the item right on top of the current anchor item. <Down> arrow key moves the anchor point to the item right below the current anchor item. <Left> arrow key moves the anchor to the parent item of the current anchor item. <Right> moves the anchor to the first child of the current anchor item. If the current anchor item does not have any children, moves the anchor to the item right below the current anchor item.

EXAMPLE

This example demonstrates how to use an HList to store a file directory structure and respond to the user's browse events:

```
use strict;
use Tk;
use Tk::Label;
use Tk::HList;

my $mw = MainWindow->new();
my $label = $mw->Label(-width=>15);
my $hlist = $mw->HList(
                    -itemtype   => 'text',
                    -separator  => '/',
                    -selectmode => 'single',
                    -browsecmd  => sub {
                              my $file = shift;
                              $label->configure(-text=>$file);
```

```
                                }
                        );

foreach ( qw(/ /home /home/ioi /home/foo /usr /usr/lib) ) {
    $hlist->add($_, -text=>$_);
}

$hlist->pack;
$label->pack;

MainLoop;
```

BUGS

The fact that the display item at column 0 is implicitly associated with the whole entry is probably a design bug. This was done for backward compatibility purposes. The result is that there is a large overlap between the **item** method and the **add**, **addchild**, **entrycget** and **entryconfigure** methods. Whenever multiple columns exist, the programmer should use ONLY the **item** method to create and configure the display items in each column; the **add**, **addchild**, **entrycget** and **entryconfigure** should be used ONLY to create and configure entries.

KEYWORDS

Hierarchical Listbox

SEE ALSO

Tk::DItem|Tk::DItem

NAME

Tk::Image – Create and manipulate images

=for category Tk Image Classes

SYNOPSIS

$image = *$widget*->*type(?arg arg ...?)*

$image->*method(?arg arg ...?)*

DESCRIPTION

The **image** constructors and methods are used to create, delete, and query images. It can take several different forms, depending on the *type*.

The constructors require a *$widget* to invoke them, this is used to locate a **MainWindow**. (This is because the underlying Tk code registers the images in the data structure for the **MainWindow**.)

The legal forms are:

$widget->*type?(?name?,?option=>value ...)?*

Creates a new image and returns an object. *type* specifies the type of the image, which must be one of the types currently defined (e.g., **Bitmap**). *name* specifies the name for the image; if it is omitted then Tk picks a name of the form **image***x*, where *x* is an integer. There may be any number of *option*=>*value* pairs, which provide configuration options for the new image. The legal set of options is defined separately for each image type; see below for details on the options for built–in image types. If an image already exists by the given name then it is replaced with the new image and any instances of that image will redisplay with the new contents.

$image->**delete**

Deletes the image *$image* and returns an empty string. If there are instances of the image displayed in widgets, the image won't actually be deleted until all of the instances are released. However, the association between the instances and the image manager will be dropped. Existing instances will retain their sizes but redisplay as empty areas. If a deleted image is recreated (with the same *name*) the existing instances will use the new image.

$image->**height**

Returns a decimal string giving the height of image *name* in pixels.

$widget->**imageNames**

Returns a list containing all existing images for *$widget*'*s* **MainWindow**.

$image->**type**

Returns the type of *$image* (the value of the *type* method when the image was created).

$widget->**imageTypes**

Returns a list whose elements are all of the valid image types (i.e., all of the values that may be supplied for the *type* to create an image).

$image->**width**

Returns a decimal string giving the width of image *name* in pixels.

BUILT–IN IMAGE TYPES

The following image types are defined by Tk so they will be available in any Tk application. Individual applications or extensions may define additional types.

Bitmap

Each pixel in the image displays a foreground color, a background color, or nothing. See *Tk::Bitmap* for more information.

Pixmap

Pixmap is slightly more general than **Bitmap**, each pixel can be any available color or "transparent" (rendered as background color of the widget image is displayed in). **Pixmap** is best used for icons and other simple graphics with only a few colors.

Pixmap is derived from Tix. See *Tk::Pixmap* for more information.

Photo

Displays a variety of full−color images, using dithering to approximate colors on displays with limited color capabilities. See *Tk::Photo* documentation for more information.

SEE ALSO

Tk::Bitmap|Tk::Bitmap Tk::Pixmap|Tk::Pixmap Tk::Photo|Tk::Photo

KEYWORDS

height, image, types of images, width

NAME

Tk::InputO – Create and manipulate TIX InputO widgets

=for category Tix Extensions

SYNOPSIS

$inputonly = *$parent*->**InputO***(?options?);*

STANDARD OPTIONS

Only the following three standard options are supported by **InputO**:

cursor **width** **height**

See *Tk::options* for details of the standard options.

WIDGET–SPECIFIC OPTIONS

InputO does not have any widget specific options.

DESCRIPTION

The **InputO** method creates a new window (given by the `$widget` argument) and makes it into a **InputO** widget. Additional options, described above, may be specified on the command line or in the option database to configure aspects of the **InputO** such as its cursor or width.

InputO widgets are not visible to the user. The only purpose of **InputO** widgets are to accept inputs from the user, which can be done with the **bind** method.

WIDGET METHODS

The **InputO** method creates a widget object. This object supports the **configure** and **cget** methods described in *Tk::options* which can be used to enquire and modify the options described above. The widget also inherits all the methods provided by the generic *Tk::Widget\Tk::Widget* class.

BINDINGS

InputO widgets have no default bindings.

NAME

CallingTk – what is Perl Tk interface doing when you call Tk functions.

=for category C Programming

This information is worse than useless for `perlTk` users, but can of some help for people interested in using modified Tk source with `perlTk`.

This document is under construction. The information is believed to be pertinent to the version of `portableTk` *available when it was created. All the details are subject to change.*

DESCRIPTION

PreCompiling

Before the actual compilation stage a script scans the source and extracts the subcommands of different commands. This information resides in the file `pTk/Methods.def`.

Compilation

During compilation the above file is included in the source of booting routine of dynamic (or static) library. More precisely, the booting code of module `Tk` calls the subroutine `Boot_Glue()` from the module `tkGlue.c`, and this subroutine includes the file (with appropriate macro definitions).

Inside `use Tk;`

The module bootstraps the C code, then loads the Perl libraries. The heart of the Perl code is contained in the `Tk::Widget` library, all the widgets inherit from this module. Code for toplevels is loaded from `Tk::MainWindow`.

During bootstrap of the C glue code the `Xevent::?` codes and a handful of `Tk::Widget` and `Tk::Image` routines are defined. (Much more XSUBs are created from `Tk.xs` code.) The widget subcommands are glued to Perl basing on the list included from `pTk/Methods.def`. In fact all the subcommands are glued to XSUBs that are related to the same C subroutine `XStoWidget()`, but have different data parts.

During the Perl code bootstrap the method `Tk::Widget::import` is called. This call requires all the code from particular widget packages.

Code from the widget packages calls an obscure command like

```
(bless \"Text")->WidgetClass;
```

This command (actually `Tk::Widget::WidgetClass()`) creates three routines: `Tk::Widget::Text()`, `Tk::Widget::isText()`, and `Tk::Text::isText()`. The first one is basically `new` of `Tk::Text`, the other two return constants. It also puts the class into depository.

Inside `$top = MainWindow->new;`

This is quite intuitive. This call goes direct to `Tk::MainWindow::new`, that calls XSUB `Tk::MainWindow::CreateMainWindow`, that calls C subroutine `Tk_CreateMainWindow()`. It is a `Tk` subroutine, so here black magic ends (almost).

The only remaining black magic is that the `Tk` initialization routine creates a lot of commands, but the subroutine for creation is usurped by **portableTk** and the commands are created in the package `Tk`. They are associated to XSUBs that are related to one of three C subroutines `XStoSubCmd()`, `XStoBind()`, or `XStoTk()`, but have different data parts.

The result of the call is blessed into `Tk::MainWindow`, as it should.

Inside `$top->title('Text demo');`

The package `Tk::Toplevel` defines a lot of subroutines on the fly on some list. All the commands from the list are converted to the corresponding subcommands of `wm` method of the widget. Here subcommand is a command with some particular second argument (in this case `"title"`). Recall that

the first argument is `$self`.

Now `Tk::Toplevel` @ISA `Tk::Widget`, that in turn @ISA `Tk`. So a call to `$top->wm('title','Text demo')` calls `Tk::wm`, that is defined during call to `Tk_CreateMainWindow()`. As it is described above, the XSUB associated to `XStoSubCmd()` is called.

This C routine is defined in `tkGlue.c`. It gets the data part of XSUB, creates a `SV` with the name of the command, and calls `Call_Tk()` with the XSUB data as the first argument, and with the name of XSUB stuffed into the Perl stack in the place there `tk` expects it. (In fact it can also reorder the arguments if it thinks it is what you want).

The latter procedure extracts name of `tk` procedure and `clientData` from the first argument and makes a call, using Perl stack as `argv` for the procedure. A lot of black magic is performed afterwards to convert result of the procedure to a Perl array return.

Inside `$text = $top->Text(background => $txtBg);`

Above we discussed how the command `Tk::Widget::Text` is created. The above command calls it via inheritance. It is translated to

```
Tk::Text::new($top, background => $txtBg);
```

The package `Tk::Text` has no method `new`, so the `Tk::Widget::new` is called. In turn it calls `Tk::Text->DoInit($top)`, that is `Tk::Widget::DoInit(Tk::Text,$top)`, that initializes the bindings if necessary. Then it creates the name for the widget of the form `.text0`, and calls `Tk::text('.text0', background => $txtBg)` (note lowercase). The result of the call is blessed into `Tk::Text`, and the method `bindtags` for this object is called.

Now the only thing to discuss is who defines the methods `text` and `bindtags`. The answer is that they are defined in `tkWindow.c`, and these commands are created in the package `Tk` in the same sweep that created the command `Tk::wm` discussed above.

So the the same C code that corresponds to the processing of corresponding TCL commands is called here as well (this time via `XStoTk` interface).

Inside `$text->insert('insert','Hello, world!');`

As we discussed above, the subcommands of widget procedures correspond to XSUB `XStoWidget`. This XSUB substitutes the first argument `$text` (that is a hash reference) to an appropriate value from this hash, adds the additional argument after the first one that contains the name of the subcommand extracted from the data part of XSUB, and calls the corresponding Tk C subroutine via `Call_Tk`.

Ilya Zakharevich <ilya@math.ohio-state.edu

NAME

Tk::IO – high level interface to Tk's 'fileevent' mechanism

=for pm IO/IO.pm

=for category Binding Events and Callbacks

SYNOPSIS

```
my $fh = Tk::IO->new(-linecommand => callback, -childcommand => callback);
$fh->exec("command")
$fh->wait
$fh->kill
```

WARNING

INTERFACES TO THIS MODULE MAY CHANGE AS PERL'S IO EVOLVES AND WITH PORT OF TK4.1

DESCRIPTION

Tk::IO is now layered on perl's IO::Handle class. Interfaces have changed, and are still evolving.

In theory C methods which enable non–blocking IO as in earlier Tk–b* release(s) are still there. I have not changed them to use perl's additional Configure information, or tested them much.

Assumption is that **exec** is used to fork a child process and a callback is called each time a complete line arrives up the implied pipe.

"line" should probably be defined in terms of perl's input record separator but is not yet.

The –childcommand callback is called when end–of–file occurs.

`$fh–`**wait** can be used to wait for child process while processing other Tk events.

`$fh–`**kill** can be used to send signal to child process.

BUGS

Still not finished. Idea is to use "exec" to emulate "system" in a non–blocking manner.

NAME

Tk::Menu::Item – Base class for Menu items

=for pm Tk/Menu/Item.pm

=for category Implementation

SYNOPSIS

```
require Tk::Menu::Item;

my $but = $menu->Button(...);
$but->configure(...);
my $what = $but->cget();

package Whatever;
require Tk::Menu::Item;
@ISA = qw(Tk::Menu::Item);

sub PreInit
{
 my ($class,$menu,$info) = @_;
 $info->{'-xxxxx'} = ...
 my $y = delete $info->{'-yyyy'};
}
```

DESCRIPTION

Tk::Menu::Item is the base class from which Tk::Menu::Button, Tk::Menu::Cascade, Tk::Menu::Radiobutton and Tk::Menu::Checkbutton are derived. There is also a Tk::Menu::Separator.

Constructors are declared so that `$menu->Button(...)` etc. do what you would expect.

The `-label` option is pre–processed allowing ~ to be prefixed to the character to derive a `-underline` value. Thus

```
$menu->Button(-label => 'Goto ~Home',...)

is equivalent to

$menu->Button(-label => 'Goto Home', -underline => 6, ...)
```

The `Cascade` menu item creates a sub–menu and accepts these options:

–menuitems

A list of items for the sub–menu. Within this list (which is also accepted by Menu and Menubutton) the first two elements of each item should be the "constructor" name and the label:

```
-menuitems => [
               [Button      => '~Quit', -command => [destroy => $mw]],
               [Checkbutton => '~Oil',  -variable => \$oil],
              ]
```

–postcommand

A callback to be invoked before posting the menu.

–tearoff

Specifies whether sub–menu can be torn–off or not.

–menuvar

Scalar reference that will be set to the newly–created sub–menu.

The returned object is currently a blessed reference to an array of two items: the containing Menu and the 'label'. Methods `configure` and `cget` are mapped onto underlying `entryconfigure` and

`entrycget`.

The main purpose of the OO interface is to allow derived item classes to be defined which pre−set the options used to create a more basic item.

BUGS

This OO interface is very new. Using the label as the "key" is a problem for separaror items which don‘t have one. The alternative would be to use an index into the menu but that is a problem if items are deleted (or inserted other than at the end).

There should probably be a PostInit entry point too, or a more widget like defered ‘configure’.

NAME

Tk::Label – Create and manipulate Label widgets

=for category Tk Widget Classes

SYNOPSIS

$label = *$parent*->**Label***(?options?);*

STANDARD OPTIONS

–anchor –font –image –takefocus –background –foreground –justify –text –bitmap –highlightbackground –padx –textvariable –borderwidth –highlightcolor –pady –underline –cursor –highlightthickness –relief –wraplength

See *Tk::options* for details of the standard options.

WIDGET–SPECIFIC OPTIONS

Name: **height**
Class: **Height**
Switch: **–height**

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

Name: **width**
Class: **Width**
Switch: **–width**

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

DESCRIPTION

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

A label is a widget that displays a textual string, bitmap or image. If text is displayed, it must all be in a single font, but it can occupy multiple lines on the screen (if it contains newlines or if wrapping occurs because of the **wrapLength** option) and one of the characters may optionally be underlined using the **underline** option. The label can be manipulated in a few simple ways, such as changing its relief or text, using the commands described below.

WIDGET METHODS

The **Label** method creates a widget object. This object supports the **configure** and **cget** methods described in *Tk::options* which can be used to enquire and modify the options described above. The widget also inherits all the methods provided by the generic *Tk::Widget|Tk::Widget* class.

BINDINGS

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

KEYWORDS

label, widget

NAME

Tk::LabFrame – labeled frame.

=for pm Tixish/LabFrame.pm

=for category Tix Extensions

SYNOPSIS

use Tk::LabFrame;

$f = $parent->**LabFrame**(?**–label**=>*text,* **–labelside**=>*where, ...?);*

DESCRIPTION

LabFrame is exactly like **Frame** and additionaly allows to add a label to the frame.

WIDGET–OPTIONS

LabFrame supports the same options as the *STANDARD OPTIONS in Frame\Tk::Frame* widget.

Additional options of **LabFrame** are:

–label => *text*

The text of the label to be placed with the Frame.

–labelside => *where*

Where can be one of **left**, **right**, **top**, **bottom** or **acrosstop**. The first four work as might be expected and place the label to the left, right, above or below the frame respectively. The **acrosstop** creates a grooved frame around the central frame and puts the label near the northwest corner such that it appears to "overwrite" the groove.

EXAMPLE

Run the following test program to see this in action:

```
use strict;
use Tk;
require Tk::LabFrame;
require Tk::LabEntry;

my $test = 'Test this';
my $mw = Tk::MainWindow->new;
my $f = $mw->LabFrame(-label => "This is a label",
                      -labelside => "acrosstop");
$f->LabEntry(-label => "Testing", -textvariable => \$test)->pack;
$f->pack;
Tk::MainLoop;
```

BUGS

Perhaps **LabFrame** should be subsumed within the generic pTk labeled widget mechanism.

AUTHOR

Rajappa Iyer rsi@earthling.net

This code is derived from LabFrame.tcl and LabWidg.tcl in the Tix4.0 distribution by Ioi Lam. The code may be redistributed under the same terms as Perl.

NAME

Tk::Listbox – Create and manipulate Listbox widgets

=for category Tk Widget Classes

SYNOPSIS

$listbox = *$parent*->**Listbox***(?options?);*

STANDARD OPTIONS

–background –foreground –relief –takefocus –borderwidth –height –selectbackground –width –cursor –highlightbackground –selectborderwidth –xscrollcommand –exportselection –highlightcolor –selectforeground –yscrollcommand –font –highlightthickness –setgrid

See *Tk::options* for details of the standard options.

WIDGET–SPECIFIC OPTIONS

Name: **height**
Class: **Height**
Switch: **–height**

Specifies the desired height for the window, in lines. If zero or less, then the desired height for the window is made just large enough to hold all the elements in the listbox.

Name: **selectMode**
Class: **SelectMode**
Switch: **–selectmode**

Specifies one of several styles for manipulating the selection. The value of the option may be arbitrary, but the default bindings expect it to be either **single**, **browse**, **multiple**, or **extended**; the default value is **browse**.

Name: **width**
Class: **Width**
Switch: **–width**

Specifies the desired width for the window in characters. If the font doesn‘t have a uniform width then the width of the character ‘‘0’’ is used in translating from character units to screen units. If zero or less, then the desired width for the window is made just large enough to hold all the elements in the listbox.

DESCRIPTION

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

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

It is not necessary for all the elements to be displayed in the listbox window at once; commands described below may be used to change the view in the window. Listboxes allow scrolling in both directions using the standard **xScrollCommand** and **yScrollCommand** options. They also support scanning, as described below.

INDICES

Many of the methods for listboxes take one or more indices as arguments. An index specifies a particular element of the listbox, in any of the following ways:

number

Specifies the element as a numerical index, where 0 corresponds to the first element in the listbox.

active

Indicates the element that has the location cursor. This element will be displayed with an underline when the listbox has the keyboard focus, and it is specified with the **activate** method.

anchor

Indicates the anchor point for the selection, which is set with the **selection anchor** method.

end Indicates the end of the listbox. For most commands this refers to the last element in the listbox, but for a few commands such as **index** and **insert** it refers to the element just after the last one.

@x,y

Indicates the element that covers the point in the listbox window specified by *x* and *y* (in pixel coordinates). If no element covers that point, then the closest element to that point is used.

In the method descriptions below, arguments named *index*, *first*, and *last* always contain text indices in one of the above forms.

WIDGET METHODS

The **Listbox** method creates a widget object. This object supports the **configure** and **cget** methods described in *Tk::options* which can be used to enquire and modify the options described above. The widget also inherits all the methods provided by the generic *Tk::Widget|Tk::Widget* class.

The following additional methods are available for listbox widgets:

`$listbox->`**activate***(index)*

Sets the active element to the one indicated by *index*. If *index* is outside the range of elements in the listbox then the closest element is activated. The active element is drawn with an underline when the widget has the input focus, and its index may be retrieved with the index **active**.

`$listbox->`**bbox***(index)*

Returns a list of four numbers describing the bounding box of the text in the element given by *index*. The first two elements of the list give the x and y coordinates of the upper–left corner of the screen area covered by the text (specified in pixels relative to the widget) and the last two elements give the width and height of the area, in pixels. If no part of the element given by *index* is visible on the screen, or if *index* refers to a non–existent element, then the result is an empty string; if the element is partially visible, the result gives the full area of the element, including any parts that are not visible.

`$listbox->`**curselection**

Returns a list containing the numerical indices of all of the elements in the listbox that are currently selected. If there are no elements selected in the listbox then an empty string is returned.

`$listbox->`**delete***(first, ?last?)*

Deletes one or more elements of the listbox. *First* and *last* are indices specifying the first and last elements in the range to delete. If *last* isn't specified it defaults to *first*, i.e. a single element is deleted.

`$listbox->`**get***(first, ?last?)*

If *last* is omitted, returns the contents of the listbox element indicated by *first*, or an empty string if *first* refers to a non–existent element. If *last* is specified, the command returns a list whose elements are all of the listbox elements between *first* and *last*, inclusive. Both *first* and *last* may have any of the standard forms for indices.

`$listbox->`**index***(index)*

Returns the integer index value that corresponds to *index*. If *index* is **end** the return value is a count of the number of elements in the listbox (not the index of the last element).

`$listbox->`**insert***(index, ?element, element, ...?)*

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

`$listbox->`**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.

`$listbox->`**scan***(option, args)*

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

`$listbox->`**scanMark***(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.

`$listbox->`**scanDragto***(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.

`$listbox->`**see***(index)*

Adjust the view in the listbox so that the element given by *index* is visible. If the element is already visible then the command has no effect; if the element is near one edge of the window then the listbox scrolls to bring the element into view at the edge; otherwise the listbox scrolls to center the element.

`$listbox->`**selection***(option, arg)*

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

`$listbox->`**selectionAnchor***(index)*

Sets the selection anchor to the element given by *index*. If *index* refers to a non–existent element, then the closest element is used. The selection anchor is the end of the selection that is fixed while dragging out a selection with the mouse. The index **anchor** may be used to refer to the anchor element.

`$listbox->`**selectionClear***(first, ?last?)*

If any of the elements between *first* and *last* (inclusive) are selected, they are deselected. The selection state is not changed for elements outside this range.

`$listbox->`**selectionIncludes***(index)*

Returns 1 if the element indicated by *index* is currently selected, 0 if it isn't.

`$listbox->`**selectionSet***(first, ?last?)*

Selects all of the elements in the range between *first* and *last*, inclusive, without affecting the selection state of elements outside that range.

`$listbox->`**size**

Returns a decimal string indicating the total number of elements in the listbox.

$listbox->**xview***(args)*

This command is used to query and change the horizontal position of the information in the widget's window. It can take any of the following forms:

$listbox->**xview**

Returns a list containing two elements. Each element is a real fraction between 0 and 1; together they describe the horizontal span that is visible in the window. For example, if the first element is .2 and the second element is .6, 20% of the listbox's text is off−screen to the left, the middle 40% is visible in the window, and 40% of the text is off−screen to the right. These are the same values passed to scrollbars via the **−xscrollcommand** option.

$listbox->**xview***(index)*

Adjusts the view in the window so that the character position given by *index* is displayed at the left edge of the window. Character positions are defined by the width of the character .

$listbox->**xview***(***moveto** => *fraction*)

Adjusts the view in the window so that *fraction* of the total width of the listbox text is off−screen to the left. *fraction* must be a fraction between 0 and 1.

$listbox->**xview***(***scroll** => *number, what*)

This command shifts the view in the window left or right according to *number* and *what*. *Number* must be an integer. *What* must be either **units** or **pages** or an abbreviation of one of these. If *what* is **units**, the view adjusts left or right by *number* character units (the width of the character) on the display; if it is **pages** then the view adjusts by *number* screenfuls. If *number* is negative then characters farther to the left become visible; if it is positive then characters farther to the right become visible.

$listbox->**yview***(?args?)*

This command is used to query and change the vertical position of the text in the widget's window. It can take any of the following forms:

$listbox->**yview**

Returns a list containing two elements, both of which are real fractions between 0 and 1. The first element gives the position of the listbox element at the top of the window, relative to the listbox as a whole (0.5 means it is halfway through the listbox, for example). The second element gives the position of the listbox element just after the last one in the window, relative to the listbox as a whole. These are the same values passed to scrollbars via the **−yscrollcommand** option.

$listbox->**yview***(index)*

Adjusts the view in the window so that the element given by *index* is displayed at the top of the window.

$listbox->**yview***(***moveto** => *fraction*)

Adjusts the view in the window so that the element given by *fraction* appears at the top of the window. *Fraction* is a fraction between 0 and 1; 0 indicates the first element in the listbox, 0.33 indicates the element one−third the way through the listbox, and so on.

$listbox->**yview***(***scroll** => *number, what*)

This command adjusts the view in the window up or down according to *number* and *what*. *Number* must be an integer. *What* must be either **units** or **pages**. If *what* is **units**, the view adjusts up or down by *number* lines; if it is **pages** then the view adjusts by *number* screenfuls. If *number* is negative then earlier elements become visible; if it is positive then later elements become visible.

DEFAULT BINDINGS

Tk automatically creates class bindings for listboxes that give them Motif−like behavior. Much of the behavior of a listbox is determined by its **selectMode** option, which selects one of four ways of dealing with the selection.

If the selection mode is **single** or **browse**, at most one element can be selected in the listbox at once. In both modes, clicking button 1 on an element selects it and deselects any other selected item. In **browse** mode it is also possible to drag the selection with button 1.

If the selection mode is **multiple** or **extended**, any number of elements may be selected at once, including discontiguous ranges. In **multiple** mode, clicking button 1 on an element toggles its selection state without affecting any other elements. In **extended** mode, pressing button 1 on an element selects it, deselects everything else, and sets the anchor to the element under the mouse; dragging the mouse with button 1 down extends the selection to include all the elements between the anchor and the element under the mouse, inclusive.

Most people will probably want to use **browse** mode for single selections and **extended** mode for multiple selections; the other modes appear to be useful only in special situations.

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

[1] In **extended** mode, the selected range can be adjusted by pressing button 1 with the Shift key down: this modifies the selection to consist of the elements between the anchor and the element under the mouse, inclusive. The un−anchored end of this new selection can also be dragged with the button down.

[2] In **extended** mode, pressing button 1 with the Control key down starts a toggle operation: the anchor is set to the element under the mouse, and its selection state is reversed. The selection state of other elements isn't changed. If the mouse is dragged with button 1 down, then the selection state of all elements between the anchor and the element under the mouse is set to match that of the anchor element; the selection state of all other elements remains what it was before the toggle operation began.

[3] If the mouse leaves the listbox window with button 1 down, the window scrolls away from the mouse, making information visible that used to be off−screen on the side of the mouse. The scrolling continues until the mouse re−enters the window, the button is released, or the end of the listbox is reached.

[4] Mouse button 2 may be used for scanning. If it is pressed and dragged over the listbox, the contents of the listbox drag at high speed in the direction the mouse moves.

[5] If the Up or Down key is pressed, the location cursor (active element) moves up or down one element. If the selection mode is **browse** or **extended** then the new active element is also selected and all other elements are deselected. In **extended** mode the new active element becomes the selection anchor.

[6] In **extended** mode, Shift−Up and Shift−Down move the location cursor (active element) up or down one element and also extend the selection to that element in a fashion similar to dragging with mouse button 1.

[7] The Left and Right keys scroll the listbox view left and right by the width of the character . Control−Left and Control−Right scroll the listbox view left and right by the width of the window. Control−Prior and Control−Next also scroll left and right by the width of the window.

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

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

[10] Control−Home sets the location cursor to the the first element in the listbox, selects that element, and deselects everything else in the listbox.

[11] Control−End sets the location cursor to the the last element in the listbox, selects that element, and deselects everything else in the listbox.

[12] In **extended** mode, Control−Shift−Home extends the selection to the first element in the listbox and Control−Shift−End extends the selection to the last element.

[13] In **multiple** mode, Control−Shift−Home moves the location cursor to the first element in the listbox and Control−Shift−End moves the location cursor to the last element.

[14] The space and Select keys make a selection at the location cursor (active element) just as if mouse button 1 had been pressed over this element.

[15] In **extended** mode, Control−Shift−space and Shift−Select extend the selection to the active element just as if button 1 had been pressed with the Shift key down.

[16] In **extended** mode, the Escape key cancels the most recent selection and restores all the elements in the selected range to their previous selection state.

[17] Control−slash selects everything in the widget, except in **single** and **browse** modes, in which case it selects the active element and deselects everything else.

[18] Control−backslash deselects everything in the widget, except in **browse** mode where it has no effect.

[19] The F16 key (labelled Copy on many Sun workstations) or Meta−w copies the selection in the widget to the clipboard, if there is a selection.

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

KEYWORDS

listbox, widget

NAME

Tk::MainWindow – Root widget of a widget tree

=for pm Tk/MainWindow.pm

=for category Creating and Configuring Widgets

SYNOPSIS

```
use Tk;

my $mw = MainWindow->new( ... options ... );

my $this = $mw->ThisWidget -> pack ;
my $that = $mw->ThatWidget;
...

MainLoop;
```

DESCRIPTION

Perl/Tk applications (which have windows associated with them) create one or more **MainWindow**s which act as the containers and parents of the other widgets.

Tk::MainWindow is a special kind of *Toplevel\Tk::Toplevel* widget. It is the root of a widget tree. Therefore `$mw->Parent` returns `undef`.

The default title of a MainWindow is the basename of the script (actually the Class name used for options lookup, i.e. with basename with inital caps) or 'Ptk' as the fallback value. If more than one MainWindow is created or several instances of the script are running at the same time the string `" #n"` is appended where the number `n` is set to get a unique value.

Unlike the standard Tcl/Tk's wish, perl/Tk allows you to create several MainWindows. When the *last* **MainWindow** is destroyed the Tk eventloop exits (the eventloop is entered with the call of `MainLoop`). Various resources (bindings, fonts, images, colors) are maintained or cached for each MainWindow, so each MainWindow consumes more resources than a Toplevel. However multiple MainWindows can make sense when the user can destroy them independently.

METHODS

You can apply all methods that a *Toplevel\Tk::Toplevel* widget accepts.

The method *$w*`->`**MainWindow** applied to any widget will return the **MainWindow** to which the widget belongs (the MainWindow belongs to itself).

MISSING

Documentation is incomplete. Here are *some* missing items that should be explained in more detail:

- The new mechanism for MainWindows is slightly different to other widgets.
- There no explanation about what resources are bound to a MainWindow (e.g., ClassInit done per MainWindow)
- Passing of command line options to override or augment arguments of the `new` method (see *Tk::CmdLine*).

SEE ALSO

Tk::Toplevel\Tk::Toplevel Tk::CmdLine\Tk::CmdLine

NAME

Tk::mega – perl/Tk support to write widgets in perl

=for category Derived Widgets

SYNOPSIS

package Tk::*Whatever;*

Construct Tk::*ValidFor 'Whatever';*

sub ClassInit { *my ($mega, $args) = @_; ... }*

For composite widget classes:

sub Populate { *my ($composite, $args) = @_; ... }*

For derived widget classes:

sub InitObject { *my ($derived, $args) = @_; ... }*

DESCRIPTION

The goal of the mega widget support of perl/Tk is to make it easy to write mega widgets that obey the same protocol and interface that the Tk core widgets support. There are two kinds of mega widgets:

- Composite Widgets

  A composite widget is composed with one or more existing widgets. The composite widget looks to the user like a simple single widget. A well known example is the file selection box.

- Derived Widgets

  A derived widget adds/modifies/removes properties and methods from a single widget (this widget may itself be a mega widget).

MEGA WIDGET SUPPORT

Advertise

Give a subwidget a symbolic name.

Usage:

$cw*->**Advertise**(**name**=>*$widget);

Gives a subwidget *$widget* of the composite widget *$cw* the name **name**. One can retrieve the reference of an advertised subwidget with the *Subwidget|"Subwidget"* method.

Comment: Mega Widget Writers: Please make sure to document the advertised widgets that are intended for *public* use. If there are none, document this fact, e.g.:

```
=head1 ADVERTISED WIDGETS

None.
```

Callback

Invoke a callback specified with an option.

Usage:

$mega->**Callback**(*-option ?,args ...?);*

Callback executes the *callback|Tk::callbacks* defined with *$mega*->**ConfigSpecs**(*-option,* [**CALLBACK**, ...]); If *args* are given they are passed to the callback. If *-option* is not defined it does nothing.

ClassInit

Initialization of the mega widget class.

Usage:

sub ClassInit { *my($class, $mw) = @_; ... }*

ClassInit is called once for *each MainWindow|Tk::MainWindow* just before the first widget instance of a class is created in the widget tree of **MainWindow**.

ClassInit is often used to define bindings and/or other resources shared by all instances, e.g., images.

Examples:

```
$mw->bind($class,"<Tab>", sub { my $w = shift; $w->Insert("\t"); $w->focus; $w->brea
$mw->bind($class,"<Return>", ['Insert',"\n"]);
$mw->bind($class,"<Delete>",'Delete');
```

Notice that *$class* is the class name (e.g. **Tk::MyText**) and *$mw* is the mainwindow.

Don't forget to call *$class*->**SUPER::ClassInit($mw)** in **ClassInit**.

Component

Convenience function to create subwidgets.

Usage:

```
$cw->Component('Whatever', 'AdvertisedName',
               -delegate => ['method1', 'method2', ...],
               ... Whatever widget options ...,
              );
```

Component does several things for you with one call:

o Creates the widget

o Advertises it with a given name (overridden by 'Name' option)

o Delegates a set of methods to this widget (optional)

Example:

```
$cw->Component('Button', 'quitButton', -command => sub{$mw->'destroy'});
```

ConfigSpecs

Defines options and their treatment

Usage:

```
$cw->ConfigSpecs(
        -option => [ where, dbname, dbclass, default],
        ...,
        DEFAULT => [where],
        );
```

Defines the options of a mega widget and what actions are triggered by configure/cget of an option (see *Tk::ConfigSpecs* and *Tk::Derived* for details).

Construct

Make the new mega widget known to **Tk**.

Usage:

Construct *baseclass* **'Name';**

Construct declares the new widget class so that your mega widget works like normal Perl/Tk widgets.

Examples:

Construct Tk::Widget *'Whatever';* **Construct Tk::Menu** *'MyItem';*

First example lets one use *$widget->***Whatever** to create new **Whatever** widget.

The second example restricts the usage of the **MyItem** constructor method to widgets that are derived from **Menu**: *$isamenu->MyItem.*

CreateArgs

Mess with options before any widget is created

sub CreateArgs { *my ($package, $parent, $args) = @_; ...; return @newargs; }*

$package is the package of the mega widget (e.g., **Tk::MyText**, *$parent* the parent of the widget to be created and `$args` the hash reference to the options specified in the widget constructor call.

Don't forget to call *$package->***SUPER::CreateArgs***($parent, $args)* in **CreateArgs**.

Delegates

Redirect a method of the mega widget to a subwidget of the composite widget

Usage:

```
$cw->Delegates(
            'method1' => $subwidget1,
            'method2' => 'advertived_name',
            ...,
            'Construct' => $subwidget2,
            'DEFAULT'   => $subwidget3,
            );
```

The **'Construct'** delegation has a special meaning. After 'Construct' is delegated all Widget constructors are redirected. E.g. after

$mega->***Delegates***('Construct'=>$subframe);

a *$mega->***Button** does really a *$subframe->***Button** so the created button is a child of *$subframe* and not *$mega*.

Comment: Delegates works only with methods that *$cw* does not have itself.

InitObject

Defines construction and interface of derived widgets.

Usage:

```
sub InitObject {
    my ($derived, $args) = @_;
    ...
}
```

where *$derived* is the widget reference of the already created baseclass widget and *$args* is the reference to a hash of *-option-value* pairs.

InitObject is almost identical to *Populate\"Populate"* method. **Populate** does some more 'magic' things useful for mega widgets with several widgets.

Don't forget to call *$derived->***SUPER::InitObject***($args)* in **InitObject**.

OnDestroy

Define callback invoked when widget is destroyed.

Usage:

$widget->**OnDestroy***(callback);*

OnDestroy installs a *callback|Tk::callbacks* that's called when a widget is going to to be destroyed. Useful for special cleanup actions. It differs from a normal **destroy** in that all the widget's data structures are still intact.

Comment: This method could be used with any widgets not just for mega widgets. It's listed here because of it's usefulness.

Populate

Defines construction and interface of the composite widget.

Usage:

```
sub Populate {
    my ($mega, $args) = @_;
    ...
}
```

where *$mega* is the widget reference of the already created baseclass widget and *$args* is the reference to a hash of *-option-value* pairs.

Most the other support function are normally used inside the **Populate** subroutine.

Don't forget to call *$cw*->**SUPER::Populate***($args)* in **Populate**.

privateData

Set/get a private hash of a widget to storage composite internal data

Usage:

$hashref = *$mega*->**privateData**();

$another = *$mega*->**privateData***(unique_key|package);*

Subwidget

Get the widget reference of an advertised subwidget.

$subwidget = *$cw*->**Subwidget***(name);*

@*subwidget* = *$cw*->**Subwidget***(name ?,...?);*

Returns the widget reference(s) of the subwidget known under the name *name*. See *Advertise|"Advertise"* method how to define *name* for a subwidget.

Comment: Mega Widget Users: Use **Subwidget** to get *only* documented subwidgets.

PITFALLS

- Resource DB class name

  Some of the standard options use a resource date base class that is not equal to the resource database name. E.g.,

```
Switch:            Name:              Class:

-padx              padX               Pad
-activerelief      activeRelief       Relief
-activebackground  activeBackground   Foreground
-status            undef              undef
```

One should do the same when one defines one of these options via **ConfigSpecs**.

- Method delegation

  Redirecting methods to a subwidget with **Delegate** can only work if the base widget itself does have a method with this name. Therefore one can't ''*delegate*'' any of the methods listed in *Tk::Widget|Tk::Widget*. A common problematic method is **bind**. In this case one as to explicitely redirect the method.

```
sub bind
  {
    my $mega = shift;
    my $to = $mega->privateData->{'my_bind_target'};
    $to->bind(@_);
  }
```

- privateData

  Graham Barr wrote: ... It is probably more private than most people think. Not all calls to privateData will return that same HASH reference. The HASH reference that is returned depends on the package it was called from, a different HASH is returned for each package. This allows a widget to hold private data, but then if it is sub−classed the sub−class will get a different HASH and so not cause duplicate name clashes.

  But privateData does take an optional argument if you want to force which HASH is returned.

- Scrolled and Composite

  Scrolled(*Kind*,...) constructor can not be used with **Composite**. One has to use `$cw`−**Composite**(**Scrl***Kind* => **'name'**, ...);

MISSING

Of course perl/Tk does not define support function for all necessities. Here's a short list of things you have to handle yourself:

- no support to define construction−time only options.
- no support to remove an option that is known to the base widget.
- it's hard to define **undef** as fallback for an widget option that is not already **undef**.
- Frame in perl/Tk carries magic and overhead not needed for composite widget class definition.
- No support methods for bindings that are shared between all widgets of a composite widget (makes sense at all?)

KEYWORDS

mega, composite, derived, widget

SEE ALSO

Tk::composite|Tk::composite Tk::ConfigSpecs|Tk::ConfigSpecs Tk::option|Tk::option Tk::callbacks|Tk::callbacks Tk::bind|Tk::bind

NAME

Tk::Menu – Create and manipulate Menu widgets

=for category Tk Widget Classes

SYNOPSIS

*$menu = $parent->***Menu***(?options?);*

STANDARD OPTIONS

–activebackground –background –disabledforeground –relief –activeborderwidth –borderwidth –font –takefocus –activeforeground –cursor –foreground

See *Tk::options* for details of the standard options.

WIDGET–SPECIFIC OPTIONS

Name: **postCommand**
Class: **Command**
Switch: **–postcommand**

If this option is specified then it provides a *callback|Tk::callbacks* to execute each time the menu is posted. The callback is invoked by the **post** method before posting the menu. Note that in 8.0 on Macintosh and Windows, all commands in a menu systems are executed before any are posted. This is due to the limitations in the individual platforms' menu managers.

Name: **selectColor**
Class: **Background**
Switch: **–selectcolor**

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

Name: **tearOff**
Class: **TearOff**
Switch: **–tearoff**

This option must have a proper boolean value, which specifies whether or not the menu should include a tear–off entry at the top. If so, it will exist as entry 0 of the menu and the other entries will number starting at 1. The default menu bindings arrange for the menu to be torn off when the tear–off entry is invoked.

Name: **tearOffCommand**
Class: **TearOffCommand**
Switch: **–tearoffcommand**

If this option has a non–empty value, then it specifies a *Tk callback|Tk::callbacks* to invoke whenever the menu is torn off. The actual command will consist of the value of this option, followed by a space, followed by the name of the menu window, followed by a space, followed by the name of the name of the torn off menu window. For example, if the option's is ''**a b**'' and menu **.x.y** is torn off to create a new menu **.x.tearoff1**, then the command ''**a b .x.y .x.tearoff1**'' will be invoked.

Name: **title**
Class: **Title**
Switch: **–title**

The string will be used to title the window created when this menu is torn off. If the title is NULL, then the window will have the title of the menubutton or the text of the cascade item from which this menu was invoked.

Name: **type**

Class: **Type**
Switch: **–type**

This option can be one of **menubar**, **tearoff**, or **normal**, and is set when the menu is created. While the string returned by the configuration database will change if this option is changed, this does not affect the menu widget's behavior. This is used by the cloning mechanism and is not normally set outside of the Tk library.

DESCRIPTION

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

A menu is a widget that displays a collection of one–line entries arranged in one or more columns. There exist several different types of entries, each with different properties. Entries of different types may be combined in a single menu. Menu entries are not the same as entry widgets. In fact, menu entries are not even distinct widgets; the entire menu is one widget.

Menu entries are displayed with up to three separate fields. The main field is a label in the form of a text string, a bitmap, or an image, controlled by the **–label**, **–bitmap**, and **–image** options for the entry. If the **–accelerator** option is specified for an entry then a second textual field is displayed to the right of the label. The accelerator typically describes a keystroke sequence that may be typed in the application to cause the same result as invoking the menu entry. The third field is an *indicator*. The indicator is present only for checkbutton or radiobutton entries. It indicates whether the entry is selected or not, and is displayed to the left of the entry's string.

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

Entries may be *disabled*, which causes their labels and accelerators to be displayed with dimmer colors. The default menu bindings will not allow a disabled entry to be activated or invoked. Disabled entries may be re–enabled, at which point it becomes possible to activate and invoke them again.

Whenever a menu's active entry is changed, a <<MenuSelect>> virtual event is sent to the menu. The active item can then be queried from the menu, and an action can be taken, such as setting context–sensitive help text for the entry.

COMMAND ENTRIES

The most common kind of menu entry is a command entry, which behaves much like a button widget. When a command entry is invoked, a callback is executed. The callback is specified with the **–command** option.

SEPARATOR ENTRIES

A separator is an entry that is displayed as a horizontal dividing line. A separator may not be activated or invoked, and it has no behavior other than its display appearance.

CHECKBUTTON ENTRIES

A checkbutton menu entry behaves much like a checkbutton widget. When it is invoked it toggles back and forth between the selected and deselected states. When the entry is selected, a particular value is stored in a particular global variable (as determined by the **–onvalue** and **–variable** options for the entry); when the entry is deselected another value (determined by the **–offvalue** option) is stored in the global variable. An indicator box is displayed to the left of the label in a checkbutton entry. If the entry is selected then the indicator's center is displayed in the color given by the **–selectcolor** option for the entry; otherwise the indicator's center is displayed in the background color for the menu. If a **–command** option is specified for a checkbutton entry, then its value is evaluated each time the entry is invoked; this happens after toggling the entry's selected state.

RADIOBUTTON ENTRIES

A radiobutton menu entry behaves much like a radiobutton widget. Radiobutton entries are organized in groups of which only one entry may be selected at a time. Whenever a particular entry becomes selected it stores a particular value into a particular global variable (as determined by the **−value** and **−variable** options for the entry). This action causes any previously−selected entry in the same group to deselect itself. Once an entry has become selected, any change to the entry's associated variable will cause the entry to deselect itself. Grouping of radiobutton entries is determined by their associated variables: if two entries have the same associated variable then they are in the same group. An indicator diamond is displayed to the left of the label in each radiobutton entry. If the entry is selected then the indicator's center is displayed in the color given by the **−selectcolor** option for the entry; otherwise the indicator's center is displayed in the background color for the menu. If a **−command** option is specified for a radiobutton entry, then its value is evaluated each time the entry is invoked; this happens after selecting the entry.

CASCADE ENTRIES

A cascade entry is one with an associated menu (determined by the **−menu** option). Cascade entries allow the construction of cascading menus. The **postcascade** method can be used to post and unpost the associated menu just next to of the cascade entry. The associated menu must be a child of the menu containing the cascade entry (this is needed in order for menu traversal to work correctly).

A cascade entry posts its associated menu by invoking

$menu−>**post**(*x,y*)

where *menu* is the path name of the associated menu, and *x* and *y* are the root−window coordinates of the upper−right corner of the cascade entry. On Unix, the lower−level menu is unposted by executing

$menu−>**unpost**

where *menu* is the name of the associated menu. On other platforms, the platform's native code takes care of unposting the menu.

If a **−command** option is specified for a cascade entry then it is evaluated whenever the entry is invoked. This is not supported on Windows.

TEAR−OFF ENTRIES

A tear−off entry appears at the top of the menu if enabled with the **tearOff** option. It is not like other menu entries in that it cannot be created with the **add** method and cannot be deleted with the **delete** method. When a tear−off entry is created it appears as a dashed line at the top of the menu. Under the default bindings, invoking the tear−off entry causes a torn−off copy to be made of the menu and all of its submenus.

MENUBARS

Any menu can be set as a menubar for a toplevel window (see the *Toplevel\|Tk::Toplevel* constructor for syntax). On the Macintosh, whenever the toplevel is in front, this menu's cascade items will appear in the menubar across the top of the main monitor. On Windows and Unix, this menu's items will be displayed in a menubar accross the top of the window. These menus will behave according to the interface guidelines of their platforms. For every menu set as a menubar, a clone menu is made. See *"CLONES"* for more information.

SPECIAL MENUS IN MENUBARS

Certain menus in a menubar will be treated specially. On the Macintosh, access to the special Apple and Help menus is provided. On Windows, access to the Windows System menu in each window is provided. On X Windows, a special right−justified help menu is provided. In all cases, these menus must be created with the command name of the menubar menu concatenated with the special name. So for a menubar named .menubar, on the Macintosh, the special menus would be .menubar.apple and .menubar.help; on Windows, the special menu would be .menubar.system; on X Windows, the help menu would be .menubar.help.

When Tk sees an Apple menu on the Macintosh, that menu's contents make up the first items of the Apple menu on the screen whenever the window containing the menubar is in front. The menu is the first one that the user sees and has a title which is an Apple logo. After all of the Tk−defined items, the menu will have a

separator, followed by all of the items in the user‘s Apple Menu Items folder. Since the System uses a different menu definition procedure for the Apple menu than Tk uses for its menus, and the system APIs do not fully support everything Tk tries to do, the menu item will only have its text displayed. No font attributes, images, bitmaps, or colors will be displayed. In addition, a menu with a tearoff item will have the tearoff item displayed as "(TearOff)".

When Tk see a Help menu on the Macintosh, the menu‘s contents are appended to the standard help menu on the right of the user‘s menubar whenever the user‘s menubar is in front. The first items in the menu are provided by Apple. Similar to the Apple Menu, cusomization in this menu is limited to what the system provides.

When Tk sees a System menu on Windows, its items are appended to the system menu that the menubar is attached to. This menu has an icon representing a spacebar, and can be invoked with the mouse or by typing Alt+Spacebar. Due to limitations in the Windows API, any font changes, colors, images, bitmaps, or tearoff images will not appear in the system menu.

When Tk see a Help menu on X Windows, the menu is moved to be last in the menubar and is right justified.

SEPARATORS IN MENUBARS

Separator entries are not displayed in menubars. The *last* separator entry causes remaining entries to be right justified.

CLONES

When a menu is set as a menubar for a toplevel window, or when a menu is torn off, a clone of the menu is made. This clone is a menu widget in its own right, but it is a child of the original. Changes in the configuration of the original are reflected in the clone. Additionally, any cascades that are pointed to are also cloned so that menu traversal will work right. Clones are destroyed when either the tearoff or menubar goes away, or when the original menu is destroyed.

WIDGET METHODS

The **Menu** method creates a widget object. This object supports the **configure** and **cget** methods described in *Tk::options* which can be used to enquire and modify the options described above. The widget also inherits all the methods provided by the generic *Tk::Widget|Tk::Widget* class, and the *Tk::Wm|Tk::Wm* class.

Many of the methods for a menu take as one argument an indicator of which entry of the menu to operate on. These indicators are called *index*es and may be specified in any of the following forms:

number

Specifies the entry numerically, where 0 corresponds to the top−most entry of the menu, 1 to the entry below it, and so on.

active

Indicates the entry that is currently active. If no entry is active then this form is equivalent to **none**. This form may not be abbreviated.

end Indicates the bottommost entry in the menu. If there are no entries in the menu then this form is equivalent to **none**. This form may not be abbreviated.

last Same as **end**.

none

Indicates ‘‘no entry at all’’; this is used most commonly with the **activate** option to deactivate all the entries in the menu. In most cases the specification of **none** causes nothing to happen in the method. This form may not be abbreviated.

@*number*

In this form, *number* is treated as a y−coordinate in the menu‘s window; the entry closest to that y−coordinate is used. For example, ‘‘**@0**’’ indicates the top−most entry in the window.

pattern

If the index doesn't satisfy one of the above forms then this form is used. *Pattern* is pattern−matched against the label of each entry in the menu, in order from the top down, until a matching entry is found. (In perl/Tk the matching is under review, but exact match should work.)

The following methods are possible for menu widgets:

$menu−>**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.

$menu−>**add***(type, ?option, value, option, value, ...?)*

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

−activebackground => *value*

Specifies a background color to use for displaying this entry when it is active. If this option is specified as an empty string (the default), then the **activeBackground** option for the overall menu is used. If the **$Tk::strictMotif** variable has been set to request strict Motif compliance, then this option is ignored and the **−background** option is used in its place. This option is not available for separator or tear−off entries.

−activeforeground => *value*

Specifies a foreground color to use for displaying this entry when it is active. If this option is specified as an empty string (the default), then the **activeForeground** option for the overall menu is used. This option is not available for separator or tear−off entries.

−accelerator => *value*

Specifies a string to display at the right side of the menu entry. Normally describes an accelerator keystroke sequence that may be typed to invoke the same function as the menu entry. This option is not available for separator or tear−off entries.

−background => *value*

Specifies a background color to use for displaying this entry when it is in the normal state (neither active nor disabled). If this option is specified as an empty string (the default), then the **background** option for the overall menu is used. This option is not available for separator or tear−off entries.

−bitmap => *value*

Specifies a bitmap to display in the menu instead of a textual label, in any of the forms accepted by **Tk_GetBitmap**. This option overrides the **−label** option but may be reset to an empty string to enable a textual label to be displayed. If a **−image** option has been specified, it overrides **−bitmap**. This option is not available for separator or tear−off entries.

−columnbreak => *value*

When this option is zero, the appears below the previous entry. When this option is one, the menu appears at the top of a new column in the menu.

−command => *value*

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

–font => *value*

Specifies the font to use when drawing the label or accelerator string in this entry. If this option is specified as an empty string (the default) then the **font** option for the overall menu is used. This option is not available for separator or tear–off entries.

–foreground => *value*

Specifies a foreground color to use for displaying this entry when it is in the normal state (neither active nor disabled). If this option is specified as an empty string (the default), then the **foreground** option for the overall menu is used. This option is not available for separator or tear–off entries.

–hidemargin => *value*

Specifies whether the standard margins should be drawn for this menu entry. This is useful when creating palette with images in them, i.e., color palettes, pattern palettes, etc. 1 indicates that the margin for the entry is hidden; 0 means that the margin is used.

–image => *value*

Specifies an image to display in the menu instead of a text string or bitmap The image must have been created by some previous invocation of **image create**. This option overrides the **–label** and **–bitmap** options but may be reset to an empty string to enable a textual or bitmap label to be displayed. This option is not available for separator or tear–off entries.

–indicatoron => *value*

Available only for checkbutton and radiobutton entries. *Value* is a boolean that determines whether or not the indicator should be displayed.

–label => *value*

Specifies a string to display as an identifying label in the menu entry. Not available for separator or tear–off entries.

–menu => *value*

Available only for cascade entries. Specifies the path name of the submenu associated with this entry. The submenu must be a child of the menu.

–offvalue => *value*

Available only for checkbutton entries. Specifies the value to store in the entry‘s associated variable when the entry is deselected.

–onvalue => *value*

Available only for checkbutton entries. Specifies the value to store in the entry‘s associated variable when the entry is selected.

–selectcolor => *value*

Available only for checkbutton and radiobutton entries. Specifies the color to display in the indicator when the entry is selected. If the value is an empty string (the default) then the **selectColor** option for the menu determines the indicator color.

–selectimage => *value*

Available only for checkbutton and radiobutton entries. Specifies an image to display in the entry (in place of the **–image** option) when it is selected. *Value* is the name of an image, which must have been created by some previous invocation of **image create**. This option is ignored unless the **–image** option has been specified.

–state => *value*

Specifies one of three states for the entry: **normal**, **active**, or **disabled**. In normal state the entry is displayed using the **foreground** option for the menu and the **background**

option from the entry or the menu. The active state is typically used when the pointer is over the entry. In active state the entry is displayed using the **activeForeground** option for the menu along with the **activebackground** option from the entry. Disabled state means that the entry should be insensitive: the default bindings will refuse to activate or invoke the entry. In this state the entry is displayed according to the **disabledForeground** option for the menu and the **background** option from the entry. This option is not available for separator entries.

–underline => *value*

Specifies the integer index of a character to underline in the entry. This option is also queried by the default bindings and used to implement keyboard traversal. 0 corresponds to the first character of the text displayed in the entry, 1 to the next character, and so on. If a bitmap or image is displayed in the entry then this option is ignored. This option is not available for separator or tear–off entries.

–value => *value*

Available only for radiobutton entries. Specifies the value to store in the entry's associated variable when the entry is selected. If an empty string is specified, then the **–label** option for the entry as the value to store in the variable.

–variable => *value*

Available only for checkbutton and radiobutton entries. Specifies the name of a global value to set when the entry is selected. For checkbutton entries the variable is also set when the entry is deselected. For radiobutton entries, changing the variable causes the currently–selected entry to deselect itself.

The **add** method returns an empty string.

$menu*–>**clone**(*$parent?, *cloneType?*)

Makes a clone of the current menu as a child of *$parent*. This clone is a menu in its own right, but any changes to the clone are propogated to the original menu and vice versa. *cloneType* can be **normal**, **menubar**, or **tearoff**. Should not normally be called outside of the Tk library. See *"CLONES"* for more information.

$menu–>**delete**(*index1?, index2*?)

Delete all of the menu entries between *index1* and *index2* inclusive. If *index2* is omitted then it defaults to *index1*. Attempts to delete a tear–off menu entry are ignored (instead, you should change the **tearOff** option to remove the tear–off entry).

$menu–>**entrycget**(*index, option*)

Returns the current value of a configuration option for the entry given by *index*. *Option* may have any of the values accepted by the **add** method.

$menu–>**entryconfigure**(*index* ?,*options*?)

This method is similar to the **configure** method, 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** method. If *options* are specified, options are modified as indicated in the method call and the method returns an empty string. If no *options* are specified, returns a list describing the current options for entry *index* (see *Tk::options* for information on the format of this list).

$menu–>**index**(*index)*

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

$menu–>**insert**(*index, type*?, *–option=>value*, ...?)

Same as the **add** method except that it inserts the new entry just before the entry given by *index*, instead of appending to the end of the menu. The *type*, *–option*, and *value* arguments have the same interpretation as for the **add** widget method. It is not possible to insert new menu entries before the

tear−off entry, if the menu has one.

$menu−>**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 callback associated with it then the result of that callback is returned as the result of the **invoke** widget method. Otherwise the result is an empty string. Note: invoking a menu entry does not automatically unpost the menu; the default bindings normally take care of this before invoking the **invoke** method.

$menu−>**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 method normally returns an empty string. If the **postCommand** option has been specified, then its value is executed before posting the menu and the result of that callback is returned as the result of the **post** widget method. If an error returns while executing the method, then the error is returned without posting the menu.

$menu−>**postcascade***(index)*

Posts the submenu associated with the cascade entry given by *index*, and unposts any previously posted submenu. If *index* doesn't correspond to a cascade entry, or if *$menu* isn't posted, the method has no effect except to unpost any currently posted submenu.

$menu−>**type***(index)*

Returns the type of the menu entry given by *index*. This is the *type* argument passed to the **add** widget method when the entry was created, such as **command** or **separator**, or **tearoff** for a tear−off entry.

$menu−>**unpost**

Unmap the window so that it is no longer displayed. If a lower−level cascaded menu is posted, unpost that menu. Returns an empty string. This method does not work on Windows and the Macintosh, as those platforms have their own way of unposting menus.

$menu−>**yposition***(index)*

Returns a decimal string giving the y−coordinate within the menu window of the topmost pixel in the entry specified by *index*.

MENU CONFIGURATIONS

The default bindings support four different ways of using menus:

Pulldown Menus in Menubar

This is the most command case. You create a menu widget that will become the menu bar. You then add cascade entries to this menu, specifying the pull down menus you wish to use in your menu bar. You then create all of the pulldowns. Once you have done this, specify the menu using the **−menu** option of the toplevel's method. See the **toplevel** manual entry for details.

Pulldown Menus in Menu Buttons

This is the compatable way to do menu bars. You create one menubutton widget for each top−level menu, and typically you arrange a series of menubuttons in a row in a menubar window. You also create the top−level menus and any cascaded submenus, and tie them together with **−menu** options in menubuttons and cascade menu entries. The top−level menu must be a child of the menubutton, and each submenu must be a child of the menu that refers to it. Once you have done this, the default bindings will allow users to traverse and invoke the tree of menus via its menubutton; see the **menubutton** documentation for details.

Popup Menus

Popup menus typically post in response to a mouse button press or keystroke. You create the popup menus and any cascaded submenus, then you call the **Post** method at the appropriate time to post the top−level menu.

$menu*->**Post**(*$x,$y?,$entry?)

$x and *$y* are the root window coordinates at which the *$menu* will be displayed. If *$entry* is specified then that entry is centred on that point, otherwise the top–left corner of the *$menu* is placed at that point.

Menu also inherits methods from *Tk::Wm* and so the method **Popup** can be used to position menu relative to other windows, the mouse cursor or the screen.

Option Menus

An option menu consists of a menubutton with an associated menu that allows you to select one of several values. The current value is displayed in the menubutton and is also stored in a global variable. Use the *Tk::Optionmenu* class to create option menubuttons and their menus.

Torn–off Menus

You create a torn–off menu by invoking the tear–off entry at the top of an existing menu. The default bindings will create a new menu that is a copy of the original menu and leave it permanently posted as a top–level window. The torn–off menu behaves just the same as the original menu.

DEFAULT BINDINGS

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

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

[2] When the mouse leaves a menu all of the entries in the menu deactivate, except in the special case where the mouse moves from a menu to a cascaded submenu.

[3] When a button is released over a menu, the active entry (if any) is invoked. The menu also unposts unless it is a torn–off menu.

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

[5] If any of the entries in a menu have letters underlined with with **–underline** option, then pressing one of the underlined letters (or its upper–case or lower–case equivalent) invokes that entry and unposts the menu.

[6] The Escape key aborts a menu selection in progress without invoking any entry. It also unposts the menu unless it is a torn–off menu.

[7] The Up and Down keys activate the next higher or lower entry in the menu. When one end of the menu is reached, the active entry wraps around to the other end.

[8] The Left key moves to the next menu to the left. If the current menu is a cascaded submenu, then the submenu is unposted and the current menu entry becomes the cascade entry in the parent. If the current menu is a top–level menu posted from a menubutton, then the current menubutton is unposted and the next menubutton to the left is posted. Otherwise the key has no effect. The left–right order of menubuttons is determined by their stacking order: Tk assumes that the lowest menubutton (which by default is the first one created) is on the left.

[9] The Right key moves to the next menu to the right. If the current entry is a cascade entry, then the submenu is posted and the current menu entry becomes the first entry in the submenu. Otherwise, if the current menu was posted from a menubutton, then the current menubutton is unposted and the next menubutton to the right is posted.

Disabled menu entries are non–responsive: they don‘t activate and they ignore mouse button presses and releases.

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

BUGS

At present it isn't possible to use the option database to specify values for the options to individual entries.

SEE ALSO

Tk::callbacks\Tk::callbacks

KEYWORDS

menu, widget

NAME

Tk::Menubutton – Create and manipulate Menubutton widgets

=for category Tk Widget Classes

SYNOPSIS

$menubutton = *$parent*->**Menubutton**(*?options?*);

STANDARD OPTIONS

–activebackground –cursor –highlightthickness –takefocus –activeforeground –disabledforeground –image –text –anchor –font –justify –textvariable –background –foreground –padx –underline –bitmap –highlightbackground–pady –wraplength –borderwidth –highlightcolor –relief

See *Tk::options* for details of the standard options.

WIDGET–SPECIFIC OPTIONS

Name: **direction**
Class: **Height**
Switch: **–direction**

Specifies where the menu is going to be popup up. **above** tries to pop the menu above the menubutton. **below** tries to pop the menu below the menubutton. **left** tries to pop the menu to the left of the menubutton. **right** tries to pop the menu to the right of the menu button. **flush** pops the menu directly over the menubutton.

Name: **height**
Class: **Height**
Switch: **–height**

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

Name: **indicatorOn**
Class: **IndicatorOn**
Switch: **–indicatoron**

The value must be a proper boolean value. If it is true then a small indicator rectangle will be displayed on the right side of the menubutton and the default menu bindings will treat this as an option menubutton. If false then no indicator will be displayed.

Name: **menu**
Class: **MenuName**
Switch: **–menu**

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

Name: **state**
Class: **State**
Switch: **–state**

Specifies one of three states for the menubutton: **normal**, **active**, or **disabled**. In normal state the menubutton is displayed using the **foreground** and **background** options. The active state is typically used when the pointer is over the menubutton. In active state the menubutton is displayed using the **activeForeground** and **activeBackground** options. Disabled state means that the menubutton should be insensitive: the default bindings will refuse to activate the widget and will ignore mouse button presses. In this state the **disabledForeground** and **background** options determine how the button is

displayed.

Name: **width**
Class: **Width**
Switch: **–width**

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

DESCRIPTION

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

A menubutton is a widget that displays a textual string, bitmap, or image and is associated with a menu widget. If text is displayed, it must all be in a single font, but it can occupy multiple lines on the screen (if it contains newlines or if wrapping occurs because of the **wrapLength** option) and one of the characters may optionally be underlined using the **underline** option. In normal usage, pressing mouse button 1 over the menubutton causes the associated menu to be posted just underneath the menubutton. If the mouse is moved over the menu before releasing the mouse button, the button release causes the underlying menu entry to be invoked. When the button is released, the menu is unposted.

Menubuttons are typically organized into groups called menu bars that allow scanning: if the mouse button is pressed over one menubutton (causing it to post its menu) and the mouse is moved over another menubutton in the same menu bar without releasing the mouse button, then the menu of the first menubutton is unposted and the menu of the new menubutton is posted instead.

There are several interactions between menubuttons and menus; see the **menu** manual entry for information on various menu configurations, such as pulldown menus and option menus.

WIDGET METHODS

The **Menubutton** method creates a widget object. This object supports the **configure** and **cget** methods described in *Tk::options* which can be used to enquire and modify the options described above. The **menu** method returns the menu associated with the widget. The widget also inherits all the methods provided by the generic *Tk::Widget\Tk::Widget* class.

DEFAULT BINDINGS

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

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

[2] Pressing mouse button 1 over a menubutton posts the menubutton: its relief changes to raised and its associated menu is posted under the menubutton. If the mouse is dragged down into the menu with the button still down, and if the mouse button is then released over an entry in the menu, the menubutton is unposted and the menu entry is invoked.

[3] If button 1 is pressed over a menubutton and then released over that menubutton, the menubutton stays posted: you can still move the mouse over the menu and click button 1 on an entry to invoke it. Once a menu entry has been invoked, the menubutton unposts itself.

[4] If button 1 is pressed over a menubutton and then dragged over some other menubutton, the original menubutton unposts itself and the new menubutton posts.

[5] If button 1 is pressed over a menubutton and released outside any menubutton or menu, the menubutton unposts without invoking any menu entry.

[6] When a menubutton is posted, its associated menu claims the input focus to allow keyboard traversal of the menu and its submenus. See the **menu** documentation for details on these bindings.

[7] If the **underline** option has been specified for a menubutton then keyboard traversal may be used to post the menubutton: Alt+*x*, where *x* is the underlined character (or its lower−case or upper−case equivalent), may be typed in any window under the menubutton's toplevel to post the menubutton.

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

[9] If a menubutton has the input focus, the space and return keys post the menubutton.

If the menubutton's state is **disabled** then none of the above actions occur: the menubutton is completely non−responsive.

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

KEYWORDS

menubutton, widget

NAME

Tk::Message – Create and manipulate Message widgets

=for category Tk Widget Classes

SYNOPSIS

$message = *$parent*->**Message***(?options?);*

STANDARD OPTIONS

–anchor –font –highlightthickness –takefocus –background –foreground –padx –text –borderwidth –highlightbackground–pady –textvariable –cursor –highlightcolor –relief –width

See *Tk::options* for details of the standard options.

WIDGET–SPECIFIC OPTIONS

Name: **aspect**
Class: **Aspect**
Switch: **–aspect**

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

Name: **justify**
Class: **Justify**
Switch: **–justify**

Specifies how to justify lines of text. Must be one of **left**, **center**, or **right**. Defaults to **left**. This option works together with the **anchor**, **aspect**, **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.

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

A message is a widget that displays a textual string. A message widget has three special features. First, it breaks up its string into lines in order to produce a given aspect ratio for the window. The line breaks are chosen at word boundaries wherever possible (if not even a single word would fit on a line, then the word will be split across lines). Newline characters in the string will force line breaks; they can be used, for example, to leave blank lines in the display.

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

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

WIDGET METHODS

The **Message** method creates a widget object. This object supports the **configure** and **cget** methods described in *Tk::options* which can be used to enquire and modify the options described above. The widget also inherits all the methods provided by the generic *Tk::Widget|Tk::Widget* class.

DEFAULT BINDINGS

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

BUGS

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

KEYWORDS

message, widget

NAME

messageBox – pop up a message window and wait for user response.

=for category Popups and Dialogs

SYNOPSIS

$response = *$widget*->**messageBox**(−*option* => *value, ...);*

DESCRIPTION

This method uses *Tk::Dialog|Tk::Dialog* to quickly create several common dialog boxes. A dialog widget consists of a message, an icon and a set of buttons (see the −*type* option). After the message window is popped up, **messageBox** waits for the user to select one of the buttons and return the button text. NOTE: unlike **Tk::Dialog** which creates its widget once and can be used many times, the **messageBox** window is created every time it's used.

The following option/value pairs are supported:

–default

The case−sensitive symbolic name of the default button for this message window ('OK', 'Cancel' and so on). See **–type** for a list of the symbolic names. If the message box has just one button it will automatically be made the default, otherwise if this option is not specified, there won't be any default button.

–icon

Specifies an icon to display. Any of the builtin Tk bitmaps can specified.

–message

Specifies the message to display.

–title

Specifies a string to display as the title.

–type

Specifies a predefined set of buttons to be displayed. The following values are possible: 'AbortRetryIgnore', 'OK', 'OKCancel', 'RetryCancel', 'YesNo' or 'YesNoCancel'.

EXAMPLE

$repsonse = *$mw*->**messageBox**(−*icon* => 'questhead', –message => 'Hello World!', –title => 'My title', –type => 'AbortRetryIgnore', –default => 'Retry');

AUTHOR

Stephen.O.Lidie@Lehigh.EDU. 98/05/25

NAME

Tk::Mwm – Communicate with the Motif(tm) window manager.

=for category Tix Extensions

SYNOPSIS

use Tk::Mwm;

$toplevel->**mwm***Option?(args)?*

$toplevel->**mwm***(option ?,args?)*

DESCRIPTION

Interface to special extentions supported by mwm.

METHODS

$toplevel->**mwmDecoration***?(?option??=>value?* ?,...?)?

When no options are given, this method returns the values of all the decorations options for the toplevel window with the *$toplevel*. When only one option is given without specifying the value, the current value of that option is returned. When more than one "option–value" pairs are passed to this method, the specified values will be assigned to the corresponding options. As a result, the appearance of the Motif decorations around the toplevel window will be changed. Possible options are: **–border**, **–menu**, **–maximize**, **–minimize**, **–resizeh** and **–title**. The value must be a Boolean value. The values returned by this command are undefined when the window is not managed by mwm.

$toplevel->**mwmIsmwmrunning**

This returns value is true if mwm is running on the screen where the specified window is located, false otherwise.

$toplevel->**mwmProtocol**

When no additional options are given, this method returns all protocols associated with this toplevel window.

$toplevel->**mwmProtocol***(***activate** => *protocol_name*)

Activate the mwm protocol message in mwm‘s menu.

$toplevel->**MwmProtocol***(***add** => *protocol_name*, *menu_message*)

Add a new mwm protocol message for this toplevel window. The message is identified by the string name specified in *protocol_name*. A menu item will be added into mwm‘s menu as specified by *menu_message*. Once a new mwm protocol message is added to a toplevel, it can be caught by the TK **protocol** method. Here is an example:

$toplevel->**mwmProtocol***(***‘add’** => *‘MY_PRINT_HELLO‘, ‘"Print Hello" _H Ctrl<Key*

$toplevel->**protocol***(***‘MY_PRINT_HELLO’** => *sub {print "Hello"});*

$toplevel->**mwmProtocol***(***‘deactivate’** => *protocol_name*)

Deactivate the mwm protocol message in mwm‘s menu.

$toplevel->**mwmProtocol***(***‘delete’** => *protocol_name*)

Delete the mwm protocol message from mwm‘s menu. Please note that the window manager protocol handler associated with this protocol (by the **protocol** method) is not deleted automatically. You have to delete the protocol handle explicitly. E.g.:

$mw->**mwmProtocol***(***‘delete’** => *‘MY_PRINT_HELLO’);*

$mw->**protocol***(***‘MY_PRINT_HELLO’** => *‘’);*

BUGS

This is a Tix extension which perl/Tk has adopted. It has not been tested as perl/Tk's author has not got round to installing a Motif Window Manager.

On some versions of mwm, the **–border** will not disappear unless **–resizeh** is turned off. Also, the **–title** will not disappear unless all of **–title**, **–menu**, **–maximize** and **–minimize** are turned off.

SEE ALSO

Tk::Wm|Tk::Wm Tk::tixWm|Tk::tixWm Tk::Toplevel|Tk::Toplevel

KEYWORDS

window manager, mwm, TIX

AUTHOR

Ioi Kim Lam – ioi@graphics.cis.upenn.edu

NAME

Tk::NoteBook – display several windows in limited space with notebook metaphor.

=for pm Tixish/NoteBook.pm

=for category Tix Extensions

SYNOPSIS

```
use Tk::NoteBook;
...
$w = $frame->NoteBook();
$page1 = $w->add("page1", options);
$page2 = $w->add("page2", options);
...
$page2 = $w->add("page2", options);
```

DESCRIPTION

The NoteBook widget provides a notebook metaphor to display several windows in limited space. The notebook is divided into a stack of pages of which only one is displayed at any time. The other pages can be selected by means of choosing the visual "tabs" at the top of the widget. Additionally, the <Tab key may be used to traverse the pages. If **–underline** is used, Alt– bindings will also work.

The widget takes all the options that a Frame does. In addition, it supports the following options:

–dynamicgeometry

If set to false (default and recommended), the size of the NoteBook will match the size of the largest page. Otherwise the size will match the size of the current page causing the NoteBook to change size when different pages of different sizes are selected.

–ipadx

The amount of internal horizontal padding around the pages.

–ipady

The amount of internal vertical padding around the pages.

METHODS

The following methods may be used with a NoteBook object in addition to standard methods.

add(*pageName*, *options*)

Adds a page with name *pageName* to the notebook. Returns an object of type **Frame**. The recognized *options* are:

–anchor

Specifies how the information in a tab is to be displayed. Must be one of **n**, **ne**, **e**, **se**, **s**, **sw**, **w**, **nw** or **center**.

–bitmap

Specifies a bitmap to display on the tab of this page. The bitmap is displayed only if none of the **–label** or **–image** options are specified.

–image

Specifies an image to display on the tab of this page. The image is displayed only if the **–label** option is not specified.

–label

Specifies the text string to display on the tab of this page.

–justify

When there are multiple lines of text displayed in a tab, this option determines the justification of the lines.

–createcmd

Specifies a *callback\|Tk::callbacks* to be called the first time the page is shown on the screen. This option can be used to delay the creation of the contents of a page until necessary. It can be useful in situations where there are a large number of pages in a NoteBook widget; with **–createcmd** you do not have to make the user wait until all pages are constructed before displaying the first page.

–raisecmd

Specifies a *callback\|Tk::callbacks* to be called whenever this page is raised by the user.

–state

Specifies whether this page can be raised by the user. Must be either **normal** or **disabled**.

–underline

Specifies the integer index of a character to underline in the tab. This option is used by the default bindings to implement keyboard traversal for menu buttons and menu entries. 0 corresponds to the first character of text displayed on the widget, 1 to the next character and so on.

–wraplength

This option specifies the maximum line length of the label string on this tab. If the line length of the label string exceeds this length, then it is wrapped onto the next line so that no line is longer than the specified length. The value may be specified in any standard forms for screen distances. If this value is less than or equal to 0, then no wrapping is done: lines will break only at newline characters in the text.

delete(*pageName*)

Deletes the page identified by *pageName*.

pagecget(*pageName*, *–option*)

Returns the current value of the configuration option given by *–option* in the page given by *pageName*. *Option* may have any of the values accepted in the **add** method.

pageconfigure(*pageName*, *options*)

Like configure for the page indicated by *pageName*. *Options* may be any of the options accepted by the **add** method.

raise(*pageName*)

Raise the page identified by *pageName*.

`raised()`

Returns the name of the currently raised page.

AUTHORS

Rajappa Iyer <rsi@earthling.net Nick Ing–Simmons <nick@ni–s.u–net.com

NAME

option − Using the option database in Perl/Tk

=for category Creating and Configuring Widgets

SYNOPSIS

$widget−>**widgetClass***(***Name**=>*name,* **−class**=>*class);*

$widget−>**PathName;**

$widget−>**optionAdd***(pattern=>value ?,priority?);*

$widget−>**optionClear;**

$widget−>**optionGet***(name, class);*

$widget−>**optionReadfile***(fileName ?,priority?);*

DESCRIPTION

The option database (also known as the *resource database* or the *application defaults database*) is a set of rules for applying default options to widgets. Users and system administrators can set up these rules to customize the appearance of applications without changing any application code; for example, a user might set up personal foreground and background colors, or a site might use fonts associated with visual or language preferences. Different window managers (and implementations of them) have implemented the database differently, but most Xt−based window managers use the *.Xdefaults* file or the *xrdb* utility to manage user preferences; some use both, and/or implement a more complex set of site, user and application databases. Check your site documentation for these topics or your window manager‘s **RESOURCE_MANAGER** property.

Being a good citizen

For most applications, the option database "just works." The **option...** methods are for applications that need to do something unusual, such as add new rules or test an option‘s default. Even in such cases, the application should provide for user preferences. Do not hardcode widget options without a **very** good reason. All users have their own tastes and they are all different. They choose a special font in a special size and have often spend a lot of time working out a color scheme that they will love until death. When you respect their choices they will enjoy working with your applications much more. Don‘t destroy the common look and feel of a personal desktop.

Option rules and widget identification

All widgets in an application are identified hierarchically by *pathname*, starting from the **MainWindow** and passing through each widget used to create the endpoint. The path elements are *widget names*, much like the elements of a file path from the root directory to a file. The rules in the option database are patterns that are matched against a widget‘s *pathname* to determine which defaults apply. When a widget is created, the **Name** option can be used to assign the widget‘s name and thus create a distinctive path for widgets in an application. If the **Name** option isn‘t given, Perl/Tk assigns a default name based on the type of widget; a **MainWindow**‘s default name is the **appname**. These defaults are fine for most widgets, so don‘t feel you need to find a meaningful name for every widget you create. A widget must have a distinctive name to allow users to tailor its options independently of other widgets in an application. For instance, to create a **Text** widget that will have special options assigned to it, give it a name such as:

```
$text = $mw->Text(Name => 'importantText');
```

You can then tailor the widget‘s attributes with a rule in the option database such as:

```
*importantText*foreground: red
```

The *class* attribute identifies groups of widgets, usually within an application but also to group similar widgets among different applications. One typically assigns a class to a **TopLevel** or **Frame** so that the class will apply to all of that widget‘s children. To extend the example, we could be more specific about the importantText widget by giving its frame a class:

```
$frame = $mw->Frame(-class => 'Urgent');
$text = $frame->Text(Name => 'importantText');
```

Then the resource pattern can be specified as so:

```
*Urgent*importantText*foreground: red
```

Similarly, the pattern `*Urgent*background: cyan` would apply to all widgets in the frame.

METHODS

$widget->**widgetClass**(**Name**=>*name*, **-class**=>*class*);

Identify a new widget with *name* and/or *class*. **Name** specifies the path element for the widget; names generally begin with a lowercase letter. **-class** specifies the class for the widget and its children; classes generally begin with an uppercase letter. If not specified, Perl/Tk will assign a unique default name to each widget. Only **MainWindow** widgets have a default class, made by uppercasing the first letter of the application name.

$widget->**PathName**;

The **PathName** method returns the widget's *pathname*, which uniquely identifies the widget within the application.

$widget->**optionAdd**(*pattern*=>*value* ?, *priority*?);

The **optionAdd** method 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 the **optionGet** method. If *priority* is specified, it indicates the priority level for this option (see below for legal values); it defaults to **interactive**. This method always returns an empty string.

$widget->**optionClear**;

The **optionClear** method clears the option database. Default options (from the **RESOURCE_MANAGER** property or the **.Xdefaults** file) will be reloaded automatically the next time an option is added to the database or removed from it. This method always returns an empty string.

$widget->**optionGet**(*name,class*);

The **optionGet** method returns the value of the option specified for *$widget* under *name* and *class*. To look up the option, **optionGet** matches the patterns in the resource database against *$widget*'s *pathname* along with the class of *$widget* (or its parent if *$widget* has no class specified). The widget's class and name are options set when the widget is created (not related to class in the sense of *bless*); the **MainWindow**'s name is the **appname** and its class is (by default) derived from the name of the script.

If several entries in the option database match *$widget*'s *pathname*, *name*, and *class*, then the method 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.

$widget->**optionReadfile**(*fileName*?,*priority*?);

The **optionReadfile** method 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** methods 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.

BUGS

The priority scheme used by core Tk is not the same as used by normal Xlib routines. In particular is assumes that the order of the entries is defined, but user commands like **xrdb −merge** can change the order.

SEE ALSO

Tk::Xrm|*Tk::Xrm*

KEYWORDS

database, option, priority, retrieve

NAME

Tk::Optionmenu – Let the user select one of some predefined options values

=for pm Tk/Optionmenu.pm

=for category Tk Widget Classes

SYNOPSIS

```
use Optionmenu;

$opt = $w->Optionmenu(
                    -options => REFERENCE_to_OPTIONLIST,
                    -command => CALLBACK,
                    -variable => SCALAR_REF,
                    );

$opt->addOptions( OPTIONLIST );

# OPTION LIST is
#   a)  $val1, $val2, $val3,...
#   b)  [ $lab1=>$val1], [$lab2=>val2], ... ]
#   c)  combination of a) and b), e.g.,
#       val1, [$lab2=>val2], val3, val4, [...], ...
```

DESCRIPTION

The **Optionmenu** widget allows the user chose between a given set of options.

If the user should be able to change the available option have a look at *Tk::BrowseEntry*.

OPTIONS

–options

(Re)sets the list of options presented.

–command

Defines the *callback\|Tk::callbacks* that is invokes when a new option is selected.

–variable

Reference to a scalar that contains the current value of the selected option.

METHODS

addOptions

Adds OPTION_LIST to the already available options.

EXAMPLE

```
use Tk;
my $mw = MainWindow->new();

my $var;
my $opt = $mw->Optionmenu(
            -options => [qw(jan feb mar apr)],
            -command => sub { print "got: ", shift, "\n" },
            -variable => \$var,
            )->pack;

$opt->addOptions([may=>5],[jun=>6],[jul=>7],[aug=>8]);

$mw->Label(-textvariable=>\$var, -relief=>'groove')->pack;
$mw->Button(-text=>'Exit', -command=>sub{$mw->destroy})->pack;
```

```
MainLoop;
```

SEE ALSO

Tk::Menubutton, *Tk::BrowseEntry*

NAME

Tk::options – Standard options supported by widgets and their manipulation

=for category Creating and Configuring Widgets

SYNOPSIS

$value = *$widget*->**cget**(*'–option'*);

$widget->**configure**(*–option=>value ?,–option=>value ...?*);

@*list* = *$widget*->**configure**(*'–option'*);

@*lol* = *$widget*->**configure;**

DESCRIPTION

All widgets, and images have a standard mechanism for setting and querying attibutes or options. The mechanism is based on two methods **configure** and **cget**. The behaviour of these methods is as follows:

$widget->**configure**(*–option=>value* ?,*–option=>value* ...?);

Sets the values of *–option* to *value* for each *–option=>value* pair. The internal **new** method does an implicit **configure** in this form with options passed in at widget create time.

$widget->**configure**(*'–option'*)

In array context returns a list of five or two elements. If *–option* is an alias for another options it return a list consisting of the alias option and the name for the option is is an alias for, e.g., `('-bg', 'background')`. If *–option* is not an alias the returned list has the following five elements:

Option Name

The value of *–option*, e.g., **–background**.

Name The option's name in the option database, e.g., `background`.

Class The option's class value in the option database, e.g., `Background`.

Default The default value for the option if not specified or in the option database, e.g., `grey`.

Value The current value (as returned by **cget**), e.g., `white`.

$widget->**configure**

Returns a list of lists for all the options supported by *$widget*. Each sub–list is in the form returned by **configure**('*–option*'). (This mechanism is used by the **Tk::Derived** class to determine the options available from base class.)

$widget->**cget**(*'–option'*)

Returns the current value of *–option* for *$widget*.

cget('*–option*') is clumsy with the need for **''** due to perl's parsing rules. Something more subtle using *tie* might look better.

The following paragraphs describe the common configuration options supported by widgets in the Tk toolkit. Every widget does not necessarily support every option (see the the documentation entries for individual widgets for a list of the standard options supported by that widget), but if a widget does support an option with one of the names listed below, then the option has exactly the effect described below.

In the descriptions below, ''Name'' refers to the option's name in the option database. ''Class'' refers to the option's class value in the option database. ''Switch'' refers to the switch used in widget–creation and **configure** widget methods to set this value. For example, if an option's configure option is **–foreground** and there exists a widget *$widget,* then the call:

$widget->**configure**(**–foreground=>'black')**

may be used to specify the value **black** for the option in the widget *$widget*. Configure options may be

abbreviated, as long as the abbreviation is unambiguous (abbreviation is deprecated in perl/Tk).

Creation options: Widget Name and Class

The **Name** and **−class** options can only be specified when a widget is created, and cannot be changed with **configure**. These options determine the widget's identity and how Tk applies resource values from the option database (see *Tk::option*) and so they cannot be assigned by the options database.

Name: *name*
Switch: **Name**

Specifies the path element for the widget. Names generally begin with a lowercase letter.

Each widget has a unique *pathname* that follows the hierarchy from the **MainWindow** to the widget itself. Since the widget's **PathName** is used to assign options from the options database, it is important to specify a distinctive **Name** for any widget that will have non−default options. See *Tk::option* for details.

Name: *class*
Switch: **−class**

Specifies a class for the window. Classes generally begin with an uppercase letter.

This class will be used when querying the option database for the window's other options (see Tk::options), and it will also be used later for other purposes such as bindings. One typically assigns a class to a **TopLevel** or **Frame** so that the class will apply to all of that widget's children.

Reconfigurable options

These options can be set at widget creation or changed later via **configure**.

Name: **activeBackground**
Class: **Foreground**
Switch: **−activebackground**

Specifies background color to use when drawing active elements. An element (a widget or portion of a widget) is active if the mouse cursor is positioned over the element and pressing a mouse button will cause some action to occur. If strict Motif compliance has been requested by setting the **`$Tk::strictMotif`** variable, this option will normally be ignored; the normal background color will be used instead. For some elements on Windows and Macintosh systems, the active color will only be used while mouse button 1 is pressed over the element.

Name: **activeBorderWidth**
Class: **BorderWidth**
Switch: **−activeborderwidth**

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).

Name: **activeForeground**
Class: **Background**
Switch: **−activeforeground**

Specifies foreground color to use when drawing active elements. See above for definition of active elements.

Name: **activetile**
Class: **Tile**
Switch: **−activetile**

Specifies image used to display inside active elements of the widget. See above for definition of active elements.

Name: **anchor**
Class: **Anchor**
Switch: **−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.

Name: **background**
Class: **Background**
Switch: **−background**
Alias: **−bg**

Specifies the normal background color to use when displaying the widget.

Name: **bitmap**
Class: **Bitmap**
Switch: **−bitmap**

Specifies a bitmap to display in the widget, in any of the forms acceptable to **Tk_GetBitmap**. The exact way in which the bitmap is displayed may be affected by other options such as **−anchor** or **−justify**. Typically, if this option is specified then it overrides other options that specify a textual value to display in the widget; the **−bitmap** option may be reset to an empty string to re−enable a text display. In widgets that support both **−bitmap** and **−image** options, **−image** will usually override **−bitmap**.

Name: **borderWidth**
Class: **BorderWidth**
Switch: **−borderwidth**
Alias: **−bd**

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**.

Name: **cursor**
Class: **Cursor**
Switch: **−cursor**

Specifies the mouse cursor to be used for the widget. The value may have any of the forms acceptable to **Tk_GetCursor**.

Name: **dash**
Class: **Dash**
Switch: **−dash**

The value may have any of the forms accepted by **Tk_GetDash**, such as **4**, **[6,4]**, **.**, **−**, **−.**, or **−..**.

Name: **dashoffset**
Class: **Dashoffset**
Switch: **−dashoffset**

Specifies the offset in the dash list where the drawing starts.

Name: **disabledForeground**
Class: **DisabledForeground**
Switch: **−disabledforeground**

Specifies foreground color to use when drawing a disabled element. If the option is specified as an empty string (which is typically the case on monochrome displays), disabled elements are drawn with the normal foreground color but they are dimmed by drawing them with a stippled fill pattern.

Name: **disabledtile**
Class: **Tile**

Switch: **–disabledtile**

Specifies image to use when drawing a disabled element.

Name: **exportSelection**
Class: **ExportSelection**
Switch: **–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**, , **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.

Name: **font**
Class: **Font**
Switch: **–font**

Specifies the font to use when drawing text inside the widget.

Name: **foreground**
Class: **Foreground**
Switch: **–foreground**
Alias: **–fg**

Specifies the normal foreground color to use when displaying the widget.

Name: **highlightBackground**
Class: **HighlightBackground**
Switch: **–highlightbackground**

Specifies the color to display in the traversal highlight region when the widget does not have the input focus.

Name: **highlightColor**
Class: **HighlightColor**
Switch: **–highlightcolor**

Specifies the color to use for the traversal highlight rectangle that is drawn around the widget when it has the input focus.

Name: **highlightThickness**
Class: **HighlightThickness**
Switch: **–highlightthickness**

Specifies a non–negative value indicating the width of the highlight rectangle to draw around the outside of the widget when it has the input focus. The value may have any of the forms acceptable to **Tk_GetPixels**. If the value is zero, no focus highlight is drawn around the widget.

Name: **image**
Class: **Image**
Switch: **–image**

Specifies an image to display in the widget, which must have been created with an image create. (See *Tk::Image* for details of image creation.) Typically, if the **–image** option is specified then it overrides other options that specify a bitmap or textual value to display in the widget; the **–image** option may be reset to an empty string to re–enable a bitmap or text display.

Name: **insertBackground**
Class: **Foreground**
Switch: **–insertbackground**

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).

Name: **insertBorderWidth**
Class: **BorderWidth**
Switch: **–insertborderwidth**

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**.

Name: **insertOffTime**
Class: **OffTime**
Switch: **–insertofftime**

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.

Name: **insertOnTime**
Class: **OnTime**
Switch: **–insertontime**

Specifies a non–negative integer value indicating the number of milliseconds the insertion cursor should remain ‘‘on’’ in each blink cycle.

Name: **insertWidth**
Class: **InsertWidth**
Switch: **–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.

Name: **jump**
Class: **Jump**
Switch: **–jump**

For widgets with a slider that can be dragged to adjust a value, such as scrollbars, this option determines when notifications are made about changes in the value. The option‘s value must be a boolean of the form accepted by **Tcl_GetBoolean**. If the value is false, updates are made continuously as the slider is dragged. If the value is true, updates are delayed until the mouse button is released to end the drag; at that point a single notification is made (the value ‘‘jumps’’ rather than changing smoothly).

Name: **justify**
Class: **Justify**
Switch: **–justify**

When there are multiple lines of text displayed in a widget, this option determines how the lines line up with each other. Must be one of **left**, **center**, or **right**. **Left** means that the lines’ left edges all line up, **center** means that the lines’ centers are aligned, and **right** means that the lines’ right edges line up.

Name: **offset**
Class: **Offset**
Switch: **–offset**

Specifies the offset of tiles (see also **–tile** option). It can have two different formats **–offset x,y** or **–offset side**, where side can be **n**, **ne**, **e**, **se**, **s**, **sw**, **w**, **nw**, or **center**. In the first case the origin is the origin of the toplevel of the current window. For the canvas itself and canvas objects the origin is the canvas origin, but putting **#** in front of the coordinate pair indicates using the toplevel origin in stead. For canvas objects, the **–offset** option is used for stippling as well. For the line and polygon canvas items you can also specify an index as argument, which connects the stipple or tile origin to one of the coordinate points of the line/polygon.

Name: **orient**
Class: **Orient**
Switch: **–orient**

For widgets that can lay themselves out with either a horizontal or vertical orientation, such as scrollbars, this option specifies which orientation should be used. Must be either **horizontal** or **vertical** or an abbreviation of one of these.

Name: **padX**
Class: **Pad**
Switch: **–padx**

Specifies a non–negative value indicating how much extra space to request for the widget in the X–direction. The value may have any of the forms acceptable to **Tk_GetPixels**. When computing how large a window it needs, the widget will add this amount to the width it would normally need (as determined by the width of the things displayed in the widget); if the geometry manager can satisfy this request, the widget will end up with extra internal space to the left and/or right of what it displays inside. Most widgets only use this option for padding text: if they are displaying a bitmap or image, then they usually ignore padding options.

Name: **padY**
Class: **Pad**
Switch: **–pady**

Specifies a non–negative value indicating how much extra space to request for the widget in the Y–direction. The value may have any of the forms acceptable to **Tk_GetPixels**. When computing how large a window it needs, the widget will add this amount to the height it would normally need (as determined by the height of the things displayed in the widget); if the geometry manager can satisfy this request, the widget will end up with extra internal space above and/or below what it displays inside. Most widgets only use this option for padding text: if they are displaying a bitmap or image, then they usually ignore padding options.

Name: **relief**
Class: **Relief**
Switch: **–relief**

Specifies the 3–D effect desired for the widget. Acceptable values are **raised**, **sunken**, **flat**, **ridge**, **solid**, and **groove**. The value indicates how the interior of the widget should appear relative to its exterior; for example, **raised** means the interior of the widget should appear to protrude from the screen, relative to the exterior of the widget.

Name: **repeatDelay**
Class: **RepeatDelay**
Switch: **–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.

Name: **repeatInterval**
Class: **RepeatInterval**
Switch: **–repeatinterval**

Used in conjunction with **repeatDelay**: once auto–repeat begins, this option determines the number of milliseconds between auto–repeats.

Name: **selectBackground**
Class: **Foreground**
Switch: **–selectbackground**

Specifies the background color to use when displaying selected items.

Name: **selectBorderWidth**
Class: **BorderWidth**
Switch: **–selectborderwidth**

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**.

Name: **selectForeground**
Class: **Background**
Switch: **–selectforeground**

Specifies the foreground color to use when displaying selected items.

Name: **setGrid**
Class: **SetGrid**
Switch: **–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 *Tk::Wm/"GRIDDED GEOMETRY MANAGEMENT"* for more details.

Name: **takeFocus**
Class: **TakeFocus**
Switch: **–takefocus**

Determines whether the window accepts the focus during keyboard traversal (e.g., Tab and Shift–Tab). Before setting the focus to a window, the traversal scripts consult the value of the **takeFocus** option. A value of means that the window should be skipped entirely during keyboard traversal. **1** means that the window should receive the input focus as long as it is viewable (it and all of its ancestors are mapped). An empty value for the option means that the traversal scripts make the decision about whether or not to focus on the window: the current algorithm is to skip the window if it is disabled, if it has no key bindings, or if it is not viewable. If the value has any other form, then the traversal scripts take the value, append the name of the window to it (with a separator space), and evaluate the resulting string as a Callback. The script must return , **1**, or an empty string: a or **1** value specifies whether the window will receive the input focus, and an empty string results in the default decision described above. Note: this interpretation of the option is defined entirely by the Callbacks that implement traversal: the widget implementations ignore the option entirely, so you can change its meaning if you redefine the keyboard traversal scripts.

Name: **text**
Class: **Text**
Switch: **–text**

Specifies a string to be displayed inside the widget. The way in which the string is displayed depends on the particular widget and may be determined by other options, such as **anchor** or **justify**.

Name: **textVariable**
Class: **Variable**
Switch: **–textvariable**

Specifies the name of a variable. The value of the variable is a text string to be displayed inside the widget; if the variable value changes then the widget will automatically update itself to reflect the new value. The way in which the string is displayed in the widget depends on the particular widget and may be determined by other options, such as **anchor** or **justify**.

Name: **troughColor**
Class: **Background**
Switch: **–troughcolor**

Specifies the color to use for the rectangular trough areas in widgets such as scrollbars and scales.

Name: **troughTile**
Class: **Tile**
Switch: **–troughtile**

Specifies image used to display in the rectangular trough areas in widgets such as scrollbars and scales.

Name: **underline**
Class: **Underline**
Switch: **–underline**

Specifies the integer index of a character to underline in the widget. This option is used by the default bindings to implement keyboard traversal for menu buttons and menu entries. 0 corresponds to the first character of the text displayed in the widget, 1 to the next character, and so on.

Name: **wrapLength**
Class: **WrapLength**
Switch: **–wraplength**

For widgets that can perform word–wrapping, this option specifies the maximum line length. Lines that would exceed this length are wrapped onto the next line, so that no line is longer than the specified length. The value may be specified in any of the standard forms for screen distances. If this value is less than or equal to 0 then no wrapping is done: lines will break only at newline characters in the text.

Name: **xScrollCommand**
Class: **ScrollCommand**
Switch: **–xscrollcommand**

Specifies a callback used to communicate with horizontal scrollbars. When the view in the widget's window changes (or whenever anything else occurs that could change the display in a scrollbar, such as a change in the total size of the widget's contents), the widget will make a callback passing two numeric arguments in addition to any specified in the callback. Each of the numbers is a fraction between 0 and 1, which indicates a position in the document. 0 indicates the beginning of the document, 1 indicates the end, .333 indicates a position one third the way through the document, and so on. The first fraction indicates the first information in the document that is visible in the window, and the second fraction indicates the information just after the last portion that is visible. Typically the **xScrollCommand** option consists of the scrollbar widget object and the method ''set'' i.e. [**set** => *`$sb`]:* this will cause the scrollbar to be updated whenever the view in the window changes. If this option is not specified, then no command will be executed.

Name: **yScrollCommand**
Class: **ScrollCommand**
Switch: **–yscrollcommand**

Specifies a calback used to communicate with vertical scrollbars. This option is treated in the same way as the **xScrollCommand** option, except that it is used for vertical scrollbars and is provided by widgets that support vertical scrolling. See the description of **xScrollCommand** for details on how this option is used.

SEE ALSO

Tk::option\Tk::option Tk::callbacks\Tk::callbacks Tk::configspec\Tk::configspec Tk_GetPixels\Tk::pTk::GetPixels

KEYWORDS

class, name, standard option, switch

NAME

Tk – An overview of an Object Oriented Tk8.0 extension for perl5

=for category Introduction

SYNOPSIS

use Tk;

```
$main = MainWindow->new();
$widget = $main->Widget(...);
$widget->pack(...);
...
MainLoop;
```

DESCRIPTION

In writing the perl Tk extension, the goals were to provide a complete interface to the latest production version of John Ousterhout's Tk, while providing an Object Oriented interface to perl code.

CONTENTS

The package is composed of three loosely connected parts:

pTk – Converted Tk source

The *pTk* sub–directory is a copy of the C code of Tk4.0, modified to allow use by languages other than the original Tcl. (The pTk can be read as 'perl' Tk or 'portable' Tk, depending on your sensibilities.)

Tk to Perl 'Glue'

The top level directory provides *Tk.xs* and *tkGlue.c* which provide the perl–callable interfaces to pTk

Perl code for 'Widget' Classes

The *Tk/Tk* sub–directory contains the various perl modules that comprise the "Classes" that are visible to Tk applications.

The "major" widgets such as **Tk::Text** are actually in separate directories at the top level (e.g. *Text/** for **Tk::Text**) and are dynamically loaded as needed on platforms which support perl5's **DynaLoader**.

CLASS HIERARCHY

package Tk; – the 'base class'

All the "command names" documented in Tcl/Tk are made to look like perl sub's and reside in the Tk package. Their names are all lower case. Typically there are very few commands at this level which are called directly by applications.

package Tk::Widget; – the 'Widget class'

There are no actual objects of the **Tk::Widget** class; however all the various Tk window "widgets" inherit from it, and it in turn inherits all the core Tk functions from Tk.

Tk::Widget provides various functions and interfaces which are common to all Widgets.

A widget is represented to perl as a blessed reference to a hash. There are some members of the hash which are private to Tk and its tkGlue code. Keys starting with **'.'** and of the form **/_[A–Z][A–Za–z_]+_/** (i.e. starting and ending in _ and with first char after _ being upper case) should be considered reserved to **Tk**.

Tk::Button, **Tk::Entry**, **Tk::Text** ...

There is one class for each of the "Tk" widget item types. Some of them like **Tk::Frame** do very little indeed, and really only exist so that they can be derived from or so that focus or menu traversal can discover the "kind" of window being processed.

Other classes, **Tk::Text** for example, provide a lot of methods used with Tk's "bind" to provide a rich keyboard/mouse interface to the widgets' data.

These widget classes also include conversions of the Tcl code for event bindings, keyboard focus traversal, menu bars, and menu keyboard traversal. All the Tcl functions have been converted, but the names have changed (systematically) and they have been split up between the various classes in what I hope is an appropriate manner. Name changes are normally: dropping initial tk_ as the Tk−ness is implicit in the **Tk::** prefix, and similarly dropping say Menu from the name if it has been moved the Tk::Menu class. Thus 'proc tkMenuNextEntry' becomes 'sub NextEntry' in the Tk::Menu package.

Tk::Image

This does for Tk4.0's "images" what **Tk::Widget** does for widgets. Images are new to Tk4.0 and the class structure is not mature either.

There are three sub−classes **Tk::Bitmap**, **Tk::Pixmap** and **Tk::Photo**.

It is expected that **Tk::Image** hierarchy will evolve during the "beta" phase of Tk to allow dynamic or auto−loaded image types or photo formats (e.g. support for JPEG format for photos).

Composite Widgets

A composite is some kind of 'frame' with subwidgets which give it useful behaviour. **Tk::Dialog** is an example of a composite widget classes built from the basic **Tk** ones. It is intended that user code should not need to be aware that a particular class is a composite, and create and configure such widgets in the same manner as any other kind. The **configure** mechanism and the methods of the class manipulate the subwidgets as required.

Composite widgets are implemented via **Tk::Frame** and multiple inheritance. The two 'frame' base classes **Tk::Frame** and **Tk::Toplevel** include the additional class **Tk::Derived** in their inheritance. **Tk::Derived** provides methods to allow additional **configure** options to be defined for a widget.

A Composite widget is typically defined as derived from **Tk::Frame** or **Tk::Toplevel** (e.g. **Tk::Dialog**).

NAME

Tk::pack – Geometry manager that packs around edges of cavity

=for category Tk Geometry Management

SYNOPSIS

$widget->**pack***?(args)?*

$widget->**pack***Option?(args)?*

DESCRIPTION

The **pack** method 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.

In this **perl** port of Tk it is normal to pack widgets one–at–a–time using the widget object to be packed to invoke a method call. This is a slight distortion of underlying Tk interface (which can handle lists of windows to one pack method call) but has proven effective in practice.

The **pack** method can have any of several forms, depending on *Option*:

$slave->**pack***?(options)?*

The options consist of pairs of arguments that specify how to manage the slave. See *"THE PACKER ALGORITHM"* below for details on how the options are used by the packer. The following options are supported:

–after => *$other*

$other must be another window. Use its master as the master for the slave, and insert the slave just after *$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*

$other must be another window. Use its master as the master for the slave, and insert the slave just before *$other* in the packing order.

–expand => *boolean*

Specifies whether the slave 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 => *$master*

Insert the slave(s) at the end of the packing order for the master window given by *$master*.

–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 slave 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 slave will be inserted at the specified point. If the slave are already managed by the geometry manager then any unspecified options for them retain their previous values rather than receiving default values.

`$slave`**–>packForget**

Removes *slave* from the packing order for its master and unmaps its window. The slave will no longer be managed by the packer.

`$slave`**–>packInfo**

Returns a list whose elements are the current configuration state of the slave given by `$slave` in the same option–value form that might be specified to **packConfigure**. The first two elements of the list are ‘‘**–in**=>`$master`’’ where `$master` is the slave‘s master.

`$master`**–>packPropagate***?(boolean)?*

If *boolean* has a true boolean value such as **1** or **on** then propagation is enabled for `$master`, (see *"GEOMETRY PROPAGATION"* below). If *boolean* has a false boolean value then propagation is disabled for `$master`. In either of these cases an empty string is returned. If *boolean* is omitted then the method returns or **1** to indicate whether propagation is currently enabled for `$master`. Propagation is enabled by default.

`$master`**–>packSlaves**

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 list/string is returned in array/scalar context, respectively

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 **packPropagate** method 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** methods to change the stacking order of either the master or the slave.

SEE ALSO

Tk::form\Tk::form Tk::grid\Tk::grid Tk::place\Tk::place

KEYWORDS

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

NAME

setPalette, bisque – Modify the Tk color palette

=for category Creating and Configuring Widgets

SYNOPSIS

$widget->**setPalette**(*background*)

$widget->**setPalette**(*name*=>*value*?,*name*=>*value* ...?)

$widget->**bisque**

DESCRIPTION

The **setPalette** method changes the color scheme for Tk. It does this by modifying the colors of existing widgets and by changing the option database so that future widgets will use the new color scheme. If **setPalette** is invoked with a single argument, the argument is the name of a color to use as the normal background color; **setPalette** will compute a complete color palette from this background color. Alternatively, the arguments to **setPalette** may consist of any number of *name*–*value* pairs, where the first argument of the pair is the name of an option in the Tk option database and the second argument is the new value to use for that option. The following database names are currently supported:

```
activeBackground        foreground      selectColor
activeForeground        highlightBackground     selectBackground
background      highlightColor  selectForeground
disabledForeground      insertBackground        troughColor
```

setPalette tries to compute reasonable defaults for any options that you don't specify. You can specify options other than the above ones and Tk will change those options on widgets as well. This feature may be useful if you are using custom widgets with additional color options.

Once it has computed the new value to use for each of the color options, **setPalette** scans the widget hierarchy to modify the options of all existing widgets. For each widget, it checks to see if any of the above options is defined for the widget. If so, and if the option's current value is the default, then the value is changed; if the option has a value other than the default, **setPalette** will not change it. The default for an option is the one provided by the widget ((**`$w->configure('option'))[3]`**) unless **setPalette** has been run previously, in which case it is the value specified in the previous invocation of **setPalette**.

After modifying all the widgets in the application, **setPalette** adds options to the option database to change the defaults for widgets created in the future. The new options are added at priority **widgetDefault**, so they will be overridden by options from the .Xdefaults file or options specified on the command–line that creates a widget.

The method **bisque** is provided for backward compatibility: it restores the application's colors to the light brown (''bisque'') color scheme used in Tk 3.6 and earlier versions.

BUGS

The use of option database names rather than the configure names is understandable given the mechanism (copied from Tcl/Tk), but is potentially confusing.

The interpolation of different 'shades' of color used for 3D effects in 'RGB' space can lead to undesirable changes in 'hue'. Interpolation in 'HSV' (as used in **Tk::ColorEditor**) would be more robust and X11R5's color support probably even more so.

SEE ALSO

Tk::options|Tk::options

KEYWORDS

bisque, color, palette

NAME

Tk::Photo – Full–color images

=for category Tk Image Classes

SYNOPSIS

$widget–>**Photo**(*?name??, options?*)

DESCRIPTION

A photo is an *image*|*Tk::Image* whose pixels can display any color or be transparent. A photo image is stored internally in full color (32 bits per pixel), and is displayed using dithering if necessary. Image data for a photo image can be obtained from a file or a string, or it can be supplied from C code through a procedural interface. At present, only GIF and PPM/PGM formats are supported, but an interface exists to allow additional image file formats to be added easily. A photo image is transparent in regions where no image data has been supplied.

CREATING PHOTOS

Photos are created using the **Photo** method. **Photo** supports the following *options*:

–data => *string*

Specifies the contents of the image as a string. The format of the string must be one of those for which there is an image file format handler that will accept string data. If both the **–data** and **–file** options are specified, the **–file** option takes precedence.

–format => *format–name*

Specifies the name of the file format for the data specified with the **–data** or **–file** option.

–file => *name*

name gives the name of a file that is to be read to supply data for the photo image. The file format must be one of those for which there is an image file format handler that can read data.

–gamma => *value*

Specifies that the colors allocated for displaying this image in a window should be corrected for a non–linear display with the specified gamma exponent value. (The intensity produced by most CRT displays is a power function of the input value, to a good approximation; gamma is the exponent and is typically around 2). The value specified must be greater than zero. The default value is one (no correction). In general, values greater than one will make the image lighter, and values less than one will make it darker.

–height => *number*

Specifies the height of the image, in pixels. This option is useful primarily in situations where the user wishes to build up the contents of the image piece by piece. A value of zero (the default) allows the image to expand or shrink vertically to fit the data stored in it.

–palette => *palette–spec*

Specifies the resolution of the color cube to be allocated for displaying this image, and thus the number of colors used from the colormaps of the windows where it is displayed. The *palette–spec* string may be either a single decimal number, specifying the number of shades of gray to use, or three decimal numbers separated by slashes (/), specifying the number of shades of red, green and blue to use, respectively. If the first form (a single number) is used, the image will be displayed in monochrome (i.e., grayscale).

–width => *number*

Specifies the width of the image, in pixels. This option is useful primarily in situations where the user wishes to build up the contents of the image piece by piece. A value of zero (the default) allows the image to expand or shrink horizontally to fit the data stored in it.

IMAGE METHODS

When a photo image is created, Tk also creates a new object. This object supports the **configure** and **cget** methods described in *Tk::options* which can be used to enquire and modify the options described above.

Those options that write data to the image generally expand the size of the image, if necessary, to accommodate the data written to the image, unless the user has specified non−zero values for the **−width** and/or **−height** configuration options, in which case the width and/or height, respectively, of the image will not be changed.

The following addition methods are available for photo images:

$image−>**blank**

Blank the image; that is, set the entire image to have no data, so it will be displayed as transparent, and the background of whatever window it is displayed in will show through.

$image−>**copy***(sourceImage* ?,*option value(s)* ...?)

Copies a region from the image called *sourceImage* (which must be a photo image) to the image called *$image*, possibly with pixel zooming and/or subsampling. If no options are specified, this method copies the whole of *sourceImage* into *$image*, starting at coordinates (0,0) in *$image*. The following options may be specified:

−from => *x1 y1 ?x2 y2?*

Specifies a rectangular sub−region of the source image to be copied. (*x1,y1*) and (*x2,y2*) specify diagonally opposite corners of the rectangle. If *x2* and *y2* are not specified, the default value is the bottom−right corner of the source image. The pixels copied will include the left and top edges of the specified rectangle but not the bottom or right edges. If the **−from** option is not given, the default is the whole source image.

−to => *x1 y1 ?x2 y2?*

Specifies a rectangular sub−region of the destination image to be affected. (*x1,y1*) and (*x2,y2*) specify diagonally opposite corners of the rectangle. If *x2* and *y2* are not specified, the default value is (*x1,y1*) plus the size of the source region (after subsampling and zooming, if specified). If *x2* and *y2* are specified, the source region will be replicated if necessary to fill the destination region in a tiled fashion.

−shrink Specifies that the size of the destination image should be reduced, if necessary, so that the region being copied into is at the bottom−right corner of the image. This option will not affect the width or height of the image if the user has specified a non−zero value for the **−width** or **−height** configuration option, respectively.

−zoom => *x y*

Specifies that the source region should be magnified by a factor of *x* in the X direction and *y* in the Y direction. If *y* is not given, the default value is the same as *x*. With this option, each pixel in the source image will be expanded into a block of *x* x *y* pixels in the destination image, all the same color. *x* and *y* must be greater than 0.

−subsample => *x y*

Specifies that the source image should be reduced in size by using only every *x*th pixel in the X direction and *y*th pixel in the Y direction. Negative values will cause the image to be flipped about the Y or X axes, respectively. If *y* is not given, the default value is the same as *x*.

$image−>**data***(?option value(s), ...*?)

returns image data in the form of a string. The following options may be specified:

−background => *color*

If the color is specified, the data will not contain any transparency information. In all transparent pixels the color will be replaced by the specified color.

–format => *format–name*

Specifies the name of the image file format handler to be used to convert the data. Specifically, this method searches for the first handler whose name matches a initial substring of *format–name* and which has the capability to write an string. If this option is not given, the data is returned in the default format as accepted by *$image*–>**put**.

–from => *x1 y1 ?x2 y2?*

Specifies a rectangular region of *$image* to be written to the string. If only *x1* and *y1* are specified, the region extends from *(x1,y1)* to the bottom–right corner of *$image*. If all four coordinates are given, they specify diagonally opposite corners of the rectangular region. The default, if this option is not given, is the whole image.

–grayscale

If this options is specified, the data will not contain color information. All pixel data will be transformed into grayscale.

$image–>**get***(x,y)*

Returns the color of the pixel at coordinates (*x*,*y*) in the image as a list of three integers between 0 and 255, representing the red, green and blue components respectively.

$image–>**put***(data* ?,**–format**=>*format–name*? ?,**–to**=> *x1 y1 ?x2 y2?*?)

Sets pixels in *imageName* to the data specified in *data*. This command first searches the list of image file format handlers for a handler that can interpret the data in *data*, and then reads the image in *filename* into *imageName* (the destination image). The following options may be specified:

–format *format–name*

Specifies the format of the image data in *data*. Specifically, only image file format handlers whose names begin with *format–name* will be used while searching for an image data format handler to read the data. Otherwise *data* is used to form a two–dimensional array of pixels that are then copied into the *$image*. *data* is structured then as a list of horizontal rows, from top to bottom, each of which is a list of colors, listed from left to right. Each color may be specified by name (e.g., blue) or in hexadecimal form (e.g., #2376af).

–from *x1 y1 x2 y2*

Specifies a rectangular sub–region of the image file data to be returned. If only *x1* and *y1* are specified, the region extends from (*x1,y1*) to the bottom–right corner of the image in the image file. If all four coordinates are specified, they specify diagonally opposite corners or the region. The default, if this option is not specified, is the whole of the image.

–shrink

If this option, the size of *imageName* will be reduced, if necessary, so that the region into which the image file data are read is at the bottom–right corner of the *imageName*. This option will not affect the width or height of the image if the user has specified a non–zero value for the **–width** or **–height** configuration option, respectively.

–to *x y*

Specifies the coordinates of the top–left corner of the region of *imageName* into which data from *filename* are to be read. The default is (0,0).

$image–>**read***(filename* ?,*option value(s), ...*?)

Reads image data from the file named *filename* into the image. This method first searches the list of image file format handlers for a handler that can interpret the data in *filename*, and then reads the image in *filename* into *$image* (the destination image). The following options may be specified:

–format => *format–name*

Specifies the format of the image data in *filename*. Specifically, only image file format handlers whose names begin with *format–name* will be used while searching for an image

data format handler to read the data.

–from => *x1 y1 ?x2 y2?*

Specifies a rectangular sub–region of the image file data to be copied to the destination image. If only *x1* and *y1* are specified, the region extends from (*x1,y1*) to the bottom–right corner of the image in the image file. If all four coordinates are specified, they specify diagonally opposite corners or the region. The default, if this option is not specified, is the whole of the image in the image file.

–shrink If this option, the size of *$image* will be reduced, if necessary, so that the region into which the image file data are read is at the bottom–right corner of the *$image*. This option will not affect the width or height of the image if the user has specified a non–zero value for the **–width** or **–height** configuration option, respectively.

–to => *x y*

Specifies the coordinates of the top–left corner of the region of *$image* into which data from *filename* are to be read. The default is (0,0).

$image–>**redither**

The dithering algorithm used in displaying photo images propagates quantization errors from one pixel to its neighbors. If the image data for *$image* is supplied in pieces, the dithered image may not be exactly correct. Normally the difference is not noticeable, but if it is a problem, this method can be used to recalculate the dithered image in each window where the image is displayed.

$image–>**write***(filename* ?,*option value(s), ...*?)

Writes image data from *$image* to a file named *filename*. The following options may be specified:

–background *color*

If the color is specified, the data will not contain any transparency information. In all transparent pixels the color will be replaced by the specified color.

–format => *format–name*

Specifies the name of the image file format handler to be used to write the data to the file. Specifically, this subcommand searches for the first handler whose name matches a initial substring of *format–name* and which has the capability to write an image file. If this option is not given, this subcommand uses the first handler that has the capability to write an image file.

–from => *x1 y1 ?x2 y2?*

Specifies a rectangular region of *$image* to be written to the image file. If only *x1* and *y1* are specified, the region extends from *(x1,y1)* to the bottom–right corner of *$image*. If all four coordinates are given, they specify diagonally opposite corners of the rectangular region. The default, if this option is not given, is the whole image.

–grayscale

If this options is specified, the data will not contain color information. All pixel data will be transformed into grayscale.

IMAGE FORMATS

The photo image code is structured to allow handlers for additional image file formats to be added easily. The photo image code maintains a list of these handlers. Handlers are added to the list by registering them with a call to **Tk_CreatePhotoImageFormat**. The standard Tk distribution comes with handlers for PPM/PGM and GIF formats, which are automatically registered on initialization.

When reading an image file or processing string data specified with the **–data** configuration option, the photo image code invokes each handler in turn until one is found that claims to be able to read the data in the file or string. Usually this will find the correct handler, but if it doesn‘t, the user may give a format name with the **–format** option to specify which handler to use. In fact the photo image code will try those

handlers whose names begin with the string specified for the **–format** option (the comparison is case–insensitive). For example, if the user specifies **–format gif**, then a handler named GIF87 or GIF89 may be invoked, but a handler named JPEG may not (assuming that such handlers had been registered).

When writing image data to a file, the processing of the **–format** option is slightly different: the string value given for the **–format** option must begin with the complete name of the requested handler, and may contain additional information following that, which the handler can use, for example, to specify which variant to use of the formats supported by the handler.

COLOR ALLOCATION

When a photo image is displayed in a window, the photo image code allocates colors to use to display the image and dithers the image, if necessary, to display a reasonable approximation to the image using the colors that are available. The colors are allocated as a color cube, that is, the number of colors allocated is the product of the number of shades of red, green and blue.

Normally, the number of colors allocated is chosen based on the depth of the window. For example, in an 8–bit PseudoColor window, the photo image code will attempt to allocate seven shades of red, seven shades of green and four shades of blue, for a total of 198 colors. In a 1–bit StaticGray (monochrome) window, it will allocate two colors, black and white. In a 24–bit DirectColor or TrueColor window, it will allocate 256 shades each of red, green and blue. Fortunately, because of the way that pixel values can be combined in DirectColor and TrueColor windows, this only requires 256 colors to be allocated. If not all of the colors can be allocated, the photo image code reduces the number of shades of each primary color and tries again.

The user can exercise some control over the number of colors that a photo image uses with the **–palette** configuration option. If this option is used, it specifies the maximum number of shades of each primary color to try to allocate. It can also be used to force the image to be displayed in shades of gray, even on a color display, by giving a single number rather than three numbers separated by slashes.

CREDITS

The photo image type was designed and implemented by Paul Mackerras, based on his earlier photo widget and some suggestions from John Ousterhout.

SEE ALSO

Tk::Bitmap|Tk::Bitmap Tk::Image|Tk::Image Tk::Pixmap|Tk::Pixmap

KEYWORDS

photo, image, color

NAME

Tk::Pixmap – Create color images from XPM files.

=for category Tk Image Classes

SYNOPSIS

$widget->**Pixmap***?(name?,options?)?*

DESCRIPTION

XPM is a popular X Window image file format for storing color icons. The **Pixmap** image type can be used to create color images using XPM files.

Pixmaps support the following *options*:

–data => *string*

Specifies the contents of the source pixmap as a string. The string must adhere to the XPM file format (e.g., as generated by the **pixmap(1)** program). If both the **–data** and **–file** options are specified, the **–data** option takes precedence. Please note that the XPM file parsing code in the xpm library is somewhat fragile. The first line of the string must be "**/* XPM** */" or otherwise a segmatation fault will be caused.

–file => *name*

name gives the name of a file whose contents define the source pixmap. The file must adhere to the XPM file format (e.g., as generated by the **pixmap(1)** program).

IMAGE METHODS

When a pixmap image is created, Tk also creates a new object. This object supports the **configure** and **cget** methods described in *Tk::options* which can be used to enquire and modify the options described above.

SEE ALSO

Tk::Image|Tk::Image

KEYWORDS

pixmap, image, tix

NAME

Tk::place – Geometry manager for fixed or rubber–sheet placement

=for category Tk Geometry Management

SYNOPSIS

$widget–**place**?(*–option=value*?, *–option=value, ...*)?

$widget–**placeForget**

$widget–**placeInfo**

$master–**placeSlaves**

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.

$slave–**place**?(*–option=value*?, *–option=value*, ...?)?

The **place** method arranges for the placer to manage the geometry of *$slave*. The remaining arguments consist of one or more *–option=value* pairs that specify the way in which *$slave's* geometry is managed. If the placer is already managing *$slave,* then the *–option=value* pairs modify the configuration for *$slave*. The **place** method returns an empty string as result. The following *–option=value* pairs are supported:

–in = *$master*

$master is the reference to the window relative to which *$slave* is to be placed. *$master* must either be *$slave's* parent or a descendant of *$slave's* parent. In addition, *$master* and *$slave* must both be descendants of the same top–level window. These restrictions are necessary to guarantee that *$slave* is visible whenever *$master* is visible. If this option isn't specified then the master defaults to *$slave's* parent.

–x = *location*

Location specifies the x–coordinate within the master window of the anchor point for *$slave* widget. 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 *$slave* widget. In this case the location is specified in a relative fashion as a floating–point number: 0.0 corresponds to the left edge of the master and 1.0 corresponds to the right edge of the master. *Location* need not be in the range 0.0–1.0. If both **–x** and **–relx** are specified for a slave then their values are summed. For example, "**–relx**=0.5, **–x**=–2" positions the left edge of the slave 2 pixels to the left of the center of its master.

–y = *location*

Location specifies the y–coordinate within the master window of the anchor point for *$slave* widget. 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 *$slave* widget. In this case the value is specified in a relative fashion as a floating–point

number: 0.0 corresponds to the top edge of the master and 1.0 corresponds to the bottom edge of the master. *Location* need not be in the range 0.0–1.0. If both **–y** and **–rely** are specified for a slave then their values are summed. For example, **–rely**=0.5, **–x**=3 positions the top edge of the slave 3 pixels below the center of its master.

–anchor = *where*

Where specifies which point of *$slave* is to be positioned at the (x,y) location selected by the **–x**, **–y**, **–relx**, and **–rely** options. The anchor point is in terms of the outer area of *$slave* including its border, if any. Thus if *where* is **se** then the lower–right corner of *$slave*'*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 *$slave* in screen units (i.e. any of the forms accepted by **Tk_GetPixels**). The width will be the outer width of *$slave* 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 *$slave*. In this case the width is specified as a floating–point number relative to the width of the master: 0.5 means *$slave* will be half as wide as the master, 1.0 means *$slave* will have the same width as the master, and so on. If both **–width** and **–relwidth** are specified for a slave, their values are summed. For example, **–relwidth**=1.0, **–width**=5 makes the slave 5 pixels wider than the master.

–height = *size*

Size specifies the height for *$slave* in screen units (i.e. any of the forms accepted by **Tk_GetPixels**). The height will be the outer dimension of *$slave* 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 *$slave*. In this case the height is specified as a floating–point number relative to the height of the master: 0.5 means *$slave* will be half as high as the master, 1.0 means *$slave* will have the same height as the master, and so on. If both **–height** and **–relheight** are specified for a slave, their values are summed. For example, **–relheight**=1.0, **–height**=–2 makes the slave 2 pixels shorter than the master.

–bordermode = *mode*

Mode determines the degree to which borders within the master are used in determining the placement of the slave. The default and most common value is **inside**. In this case the placer considers the area of the master to be the innermost area of the master, inside any border: an option of **–x=** corresponds to an x–coordinate just inside the border and an option of **–relwidth**=1.0 means *$slave* will fill the area inside the master's border. If *mode* is **outside** then the placer considers the area of the master to include its border; this mode is typically used when placing *$slave* outside its master, as with the options **–x=**, **–y=**, **–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.

$slave–**placeSlaves**

The **placeSlaves** method returns a list of all the slave windows for which *$master* is the master. If there are no slaves for *$master* then an empty list is returned.

$slave–**placeForget**

The **placeForget** method causes the placer to stop managing the geometry of *$slave*. As a side effect of this method call *$slave* will be unmapped so that it doesn‘t appear on the screen. If *$slave* isn‘t currently managed by the placer then the method call has no effect. **placeForget** returns an empty string as result.

$slave–**placeInfo**

The **placeInfo** method returns a list giving the current configuration of *$slave*. The list consists of *–option=value* pairs in exactly the same form as might be specified to the **place** method. If the configuration of a window has been retrieved with **placeInfo**, that configuration can be restored later by first using **placeForget** to erase any existing information for the window and then invoking **place** 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* widget 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.

SEE ALSO

Tk::form|Tk::form Tk::grid|Tk::grid Tk::pack|Tk::pack

KEYWORDS

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

NAME

Tk2portableTk – how to make your **Tk** source portable to other interpreted languages.

=for category C Programming

Author

Ilya Zakharevich <ilya@math.ohio–state.edu has contributed most of this document. Many thanks.

DESCRIPTION

PortableTk is an attempt to make **Tk** useful from other languages. Currently tk4.0 runs under Perl using this approach. Below, *Lang* is the notation for an external language to which **PortableTk** glues **Tk** code.

The main problem with using the code developed for **TCL** with different languages is the absence of data types: almost anything is `char*`. It makes automatic translation hopeless. However, if you `typedef` several new symbols to be `char*`, you can still use your code in **TCL**, *and* it will make the automatic translation possible.

Another problem with the approach that "everything is a string" is impossibility to have a result that says "NotApplicable" without setting an error. Thus different **Tk** command return different string values that mean "error happened", like `""`, `" "` or `"??"`. Other languages can be more flexible, so in **portableTk** you should inform the compiler that what you want to return means "error" (see *Setting variables*).

Currently **PortableTk** uses several different approachs to simplify translation: several **TCL** functions that are especially dangerous to use are undefined, so you can easily find places that need to be updated to use Language–independent functions based on compiler warnings. Eventually a way to use these Language–independent functions under proper **TCL** will be also provided. The end of this document provides a starting point for such a project.

Structure of pTk, porting your code

pTk, that is a port of **Tk**, is very special with respect to porting of other code to **portableTk**. The problem is that currently there is very little hope to merge the modifications back into **Tk**, so a special strategy is needed to maintain this port. Do not use this strategy to port your own code.

pTk is produced from **Tk** via a two–step process: first, some manual editing (the result is in the subdirectory `mTk`), and second, automatic conversion by the `munge` script (written in Perl). Thus the subdirectory `pTk/mTk` contains code with minimal possible difference from the virgin **Tk** code, so it is easier to merge(1) the differences between **Tk** versions into modified code.

It looks like the strategy for a portable code should be exactly opposite: starting from **TCL**–based code, apply `munge`, and then hand–edit the resulting code. Probably it is also possible to target your code to **portableTk** from scratch, since this will make it possible to run it under a lot of *Lang*uages.

The only reason anyone would like to look into contents of `pTk/mTk` directory is to find out which constructs are not supported by `munge`. On the other hand, `pTk` directory contains code that is conformant to **portableTk**, so you can look there to find example code.

`munge` is the script that converts most common **Tk** constructs to their `portableTk` equivalent. For your code to qualify, you should follow **Tk** conventions on indentation and names of variables, in particular, the array of arguments for the `...CmdProc` should be called `argv`.

For details on what `munge` can do, see *Translation of some TCL functions*.

PortableTk API

Checking what you are running under

PortableTk provides a symbol `????`. If this symbol is defined, your source is compiled with it.

New types of configuration options

PortableTk defines several new types of configuration options:

```
TK_CONFIG_CALLBACK
TK_CONFIG_LANGARG
TK_CONFIG_SCALARVAR
TK_CONFIG_HASHVAR
TK_CONFIG_ARRAYVAR
TK_CONFIG_IMAGE
```

You should use them instead of TK_CONFIG_STRING whenever appropriate. This allows your application to receive a direct representation of the corresponding resource instead of the string representation, if this is possible under given language.

???? It looks like `TK_CONFIG_IMAGE` and `TK_CONFIG_SCALARVAR` set variables of type `char*`.

Language data

The following data types are defined:

`Arg` is the main datatype of the language. This is a type that your C function gets pointers to for arguments when the corresponding *Lang* function is called. The corresponding config type is `TK_CONFIG_LANGARG`.

This is also a type that keeps information about contents of *Lang* variable.

`Var` Is a substitute for a `char *` that contains name of variable. In *Lang* it is an object that contains reference to another *Lang* variable.

`LangResultSave`

????

`LangCallback`

`LangCallback*` a substitute for a `char *` that contains command to call. The corresponding config type is `TK_CONFIG_CALLBACK`.

`LangFreeProc`

It is the type that the `Lang_SplitList` sets. Before you call it, declare

```
Args *args;
LangFreeProc *freeProc = NULL;
...
code = Lang_SplitList(interp, value,
    &argc, &args, &freeProc);
```

After you use the split values, call

```
if (args != NULL && freeProc) (*freeProc)(argc,args);
```

It is not guaranteed that the `args` can survive deletion of `value`.

Conversion

The following macros and functions are used for conversion between strings and the additional types:

```
LangCallback * LangMakeCallback(Arg)
Arg LangCallbackArg(LangCallback *)
char * LangString(Arg)
```

After you use the result of `LangCallbackArg()`, you should free it with `freeProc` `LANG_DYNAMIC` (it is not guaranteed that any change of `Arg` will not be reflected in <LangCallback, so you cannot do `LangSet...()` in between, and you should reset it to `NULL` if you want to do any further assignments to this `Arg`).

The following function returns the `Arg` that is a reference to `Var`:

```
Arg LangVarArg(Var)
```

???? It is very anti–intuitive, I hope the name is changed.

```
int LangCmpCallback(LangCallback *a,Arg b)
```

(currently only a stub), and, at last,

```
LangCallback * LangCopyCallback(LangCallback *)
```

Callbacks

Above we have seen the new datatype `LangCallback` and the corresponding *Config option* `TK_CONFIG_CALLBACK`. The following functions are provided for manipulation of `LangCallbacks`:

```
void LangFreeCallback(LangCallback *)
int LangDoCallback(Tcl_Interp *,LangCallback *,
       int result,int argc, char *format,...)
```

The argument `format` of `LangDoCallback` should contain a string that is suitable for `sprintf` with optional arguments of `LangDoCallback`. `result` should be false if result of callback is not needed.

```
int LangMethodCall(Tcl_Interp *,Arg,char *method,
       int result,int argc,...)
```

????

Conceptually, `LangCallback*` is a substitute for ubiquitous `char *` in **TCL**. So you should use `LangFreeCallback` instead of `ckfree` or `free` if appropriate.

Setting variables

```
void LangFreeArg (Arg, Tcl_FreeProc *freeProc)
Arg  LangCopyArg (Arg);
void Tcl_AppendArg (Tcl_Interp *interp, Arg)
void LangSetString(Arg *, char *s)
void LangSetDefault(Arg *, char *s)
```

These two are equivalent unless s is an empty string. In this case `LangSetDefault` behaves like `LangSetString` with `s==NULL`, i.e., it sets the current value of the *Lang* variable to be false.

```
void LangSetInt(Arg *,int)
void LangSetDouble(Arg *,double)
```

The *Lang* functions separate uninitialized and initialized data comparing data with `NULL`. So the declaration for an `Arg` should look like

```
Arg arg = NULL;
```

if you want to use this `arg` with the above functions. After you are done, you should use `LangFreeArg` with `TCL_DYNAMIC` as `freeProc`.

Language functions

Use

`int  LangNull(Arg)`

to check that an object is false;

`int  LangStringMatch(char *string, Arg match)`

????

`void LangExit(int)`

to make a proper shutdown;

`int LangEval(Tcl_Interp *interp, char *cmd, int global)`

to call *Lang* `eval`;

```
void Lang_SetErrorCode(Tcl_Interp *interp,char *code)
char *Lang_GetErrorCode(Tcl_Interp *interp)
char *Lang_GetErrorInfo(Tcl_Interp *interp)
void LangCloseHandler(Tcl_Interp *interp,Arg arg,FILE
    *f,Lang_FileCloseProc *proc)
```

currently stubs only;

```
int LangSaveVar(Tcl_Interp *,Arg arg,Var *varPtr,int type)
```

to save the structure `arg` into *Lang* variable `*varPtr`;

```
void LangFreeVar(Var var)
```

to free the result;

```
int LangEventCallback(Tcl_Interp *,LangCallback *,XEvent *,KeySym)
```

????

```
int LangEventHook(int flags)
void LangBadFile(int fd)
int LangCmpConfig(char *spec, char *arg, size_t length)
```

unsupported????;

```
void Tcl_AppendArg (Tcl_Interp *interp, Arg)
```

Another useful construction is

```
Arg variable = LangFindVar(interp, Tk_Window tkwin, char *name);
```

After using the above function, you should call

```
LangFreeVar(Var variable);
```

???? Note discrepancy in types!

If you want to find the value of a variable (of type `Arg`) given the variable name, use `Tcl_GetVar(interp, varName, flags)`. If you are interested in the string value of this variable, use `LangString(Tcl_GetVar(...))`.

To get a **C** array of `Arg` of length `n`, use

```
Arg *args = LangAllocVec(n);
...
LangFreeVec(n,args);
```

You can set the values of the `Args` using `LangSet...` functions, and get string value using `LangString`.

If you want to merge an array of `Args` into one `Arg` (that will be an array variable), use

```
result = Tcl_Merge(listLength, list);
```

Translation of some TCL functions

We mark items that can be dealt with by `munge` by *Autoconverted*.

`Tcl_AppendResult`

does not take `(char*)NULL`, but `NULL` as delimiter. *Autoconverted*.

`Tcl_CreateCommand`, `Tcl_DeleteCommand`

`Tk_CreateWidget`, `Tk_DeleteWidget`, the second argument is the window itself, not the pathname. *Autoconverted*.

`sprintf(interp->result, "%d %d %d %d",...)`

`Tcl_IntResults(interp,4,0,...)`. *Autoconverted.*

`interp->result = "1";`

`Tcl_SetResult(interp,"1", TCL_STATIC)`. *Autoconverted.*

Reading `interp->result`

`Tcl_GetResult(interp)`. *Autoconverted.*

`interp->result = Tk_PathName(textPtr->tkwin);`

`Tk_WidgetResult(interp,textPtr->tkwin)`. *Autoconverted.*

Sequence `Tcl_PrintDouble, Tcl_PrintDouble, ..., Tcl_AppendResult`

Use a single command

```
void Tcl_DoubleResults(Tcl_Interp *interp, int append,
       int argc,...);
```

`append` governs whether it is required to clear the result first.

A similar command for `int` arguments is `Tcl_IntResults`.

`Tcl_SplitList`

Use `Lang_SplitList` (see the description above).

Translation back to TCL

To use your **portableTk** program with **TCL**, put

```
#include "ptcl.h"
```

before inclusion of `tk.h`, and link the resulting code with `ptclGlue.c`.

These files currently implement the following:

Additional config types:

```
TK_CONFIG_CALLBACK
TK_CONFIG_LANGARG
TK_CONFIG_SCALARVAR
TK_CONFIG_HASHVAR
TK_CONFIG_ARRAYVAR
TK_CONFIG_IMAGE
```

Types:

```
Var, Arg, LangCallback, LangFreeProc.
```

Functions and macros:

```
Lang_SplitList, LangString, LangSetString, LangSetDefault,
LangSetInt, LangSetDouble Tcl_ArgResult, LangCallbackArg,
LangSaveVar, LangFreeVar,
LangFreeSplitProc, LangFreeArg, Tcl_DoubleResults, Tcl_IntResults,
LangDoCallback, Tk_WidgetResult, Tcl_CreateCommand,
Tcl_DeleteCommand, Tcl_GetResult.
```

Current implementation contains enough to make it possible to compile `mTk/tkText*.[ch]` with the virgin **Tk**.

New types of events ????

PortableTk defines following new types of events:

```
TK_EVENTTYPE_NONE
TK_EVENTTYPE_STRING
```

```
TK_EVENTTYPE_NUMBER
TK_EVENTTYPE_WINDOW
TK_EVENTTYPE_ATOM
TK_EVENTTYPE_DISPLAY
TK_EVENTTYPE_DATA
```

and a function

```
char * Tk_EventInfo(int letter,
           Tk_Window tkwin, XEvent *eventPtr,
           KeySym keySym, int *numPtr, int *isNum, int *type,
           int num_size, char *numStorage)
```

Checking for trouble

If you start with working TCL code, you can start convertion using the above hints. Good indication that you are doing is OK is absence of `sprintf` and `sscanf` in your code (at least in the part that is working with interpreter).

Additional API

What is described here is not included into base **portableTk** distribution. Currently it is coded in **TCL** and as Perl macros (core is coded as functions, so theoretically you can use the same object files with different interpreted languages).

`ListFactory`

Dynamic arrays in **TCL** are used for two different purposes: to construct strings, and to construct lists. These two usages will have separate interfaces in other languages (since list is a different type from a string), so you should use a different interface in your code.

The type for construction of dynamic lists is `ListFactory`. The API below is a counterpart of the API for construction of dynamic lists in **TCL**:

```
void ListFactoryInit(ListFactory *)
void ListFactoryFinish(ListFactory *)
void ListFactoryFree(ListFactory *)
Arg * ListFactoryArg(ListFactory *)
void ListFactoryAppend(ListFactory *, Arg *arg)
void ListFactoryAppendCopy(ListFactory *, Arg *arg)
ListFactory * ListFactoryNewLevel(ListFactory *)
ListFactory * ListFactoryEndLevel(ListFactory *)
void ListFactoryResult(Tcl_Interp *, ListFactory *)
```

The difference is that a call to `ListFactoryFinish` should precede the actual usage of the value of `ListFactory`, and there are two different ways to append an `Arg` to a `ListFactory`:
`ListFactoryAppendCopy()` guarantees that the value of `arg` is copied to the list, but
`ListFactoryAppend()` may append to the list a reference to the current value of `arg`. If you are not going to change the value of `arg` after appending, the call to ListFactoryAppend may be quicker.

As in **TCL**, the call to `ListFactoryFree()` does not free the `ListFactory`, only the objects it references.

The functions `ListFactoryNewLevel()` and `ListFactoryEndLevel()` return a pointer to a `ListFactory` to fill. The argument of `ListFactoryEndLevel()` cannot be used after a call to this function.

DStrings

Production of strings are still supported in **portableTk**.

Accessing Args

The following functions for getting a value of an Arg *may* be provided:

```
double LangDouble(Arg)
int LangInt(Arg)
long LangLong(Arg)
int LangIsList(Arg arg)
```

The function LangIsList() is supported only partially under **TCL**, since there is no data types. It checks whether there is a space inside the string arg.

Assigning numbers to Args

While LangSetDouble() and LangSetInt() are supported ways to assign numbers to assign an integer value to a variable, for the sake of efficiency under **TCL** it is supposed that the destination of these commands was massaged before the call so it contains a long enough string to sprintf() the numbers inside it. If you are going to immediately use the resulting Arg, the best way to do this is to declare a buffer in the beginning of a block by

```
dArgBuffer;
```

and assign this buffer to the Arg by

```
void LangSetDefaultBuffer(Arg *)
```

You can also create the buffer(s) manually and assign them using

```
void LangSetBuffer(Arg *, char *)
```

This is the only choice if you need to assign numeric values to several Args simultaneously. The advantage of the first approach is that the above declarations can be made nops in different languages.

Note that if you apply LangSetDefaultBuffer to an Arg that contains some value, you can create a leak if you do not free that Arg first. This is a non–problem in real languages, but can be a trouble in TCL, unless you use only the above API.

Creating new Args

The API for creating a new Arg is

```
void LangNewArg(Arg *, LangFreeProc *)
```

The API for creating a new Arg is absent. Just initialize Arg to be NULL, and apply one of LangSet... methods.

After you use this Arg, it should be freed thusly:

```
LangFreeArg(arg, freeProc).
```

Evaluating a list

Use

```
int LangArgEval(Tcl_Interp *, Arg arg)
```

Here arg should be a list to evaluate, in particular, the first element should be a LangCallback massaged to be an Arg. The arguments can be send to the subroutine by reference or by value in different languages.

Getting result as Arg

Use Tcl_ArgResult. It is not guaranteed that result survives this operation, so the Arg you get should be the only mean to access the data from this moment on. After you use this Arg, you should free it with freeProc LANG_DYNAMIC (you can do LangSet...() in between).

NAME

Tk::Radiobutton – Create and manipulate Radiobutton widgets

=for category Tk Widget Classes

SYNOPSIS

$radiobutton = *$parent*->**Radiobutton***(?options?);*

STANDARD OPTIONS

–activebackground –cursor –highlightthickness –takefocus –activeforeground –disabledforeground –image –text –anchor –font –justify –textvariable –background –foreground –padx –underline –bitmap –highlightbackground–pady –wraplength –borderwidth –highlightcolor –relief

See *Tk::options* for details of the standard options.

WIDGET–SPECIFIC OPTIONS

Name: **command**
Class: **Command**
Switch: **–command**

Specifies a *Tk callback\Tk::callbacks* 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.

Name: **height**
Class: **Height**
Switch: **–height**

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

Name: **indicatorOn**
Class: **IndicatorOn**
Switch: **–indicatoron**

Specifies whether or not the indicator should be drawn. Must be a proper boolean value. If false, the **relief** option is ignored and the widget's relief is always sunken if the widget is selected and raised otherwise.

Name: **selectColor**
Class: **Background**
Switch: **–selectcolor**

Specifies a background color to use when the button is selected. If **indicatorOn** is true then the color applies to the indicator. Under Windows, this color is used as the background for the indicator regardless of the select state. If **indicatorOn** is false, this color is used as the background for the entire widget, in place of **background** or **activeBackground**, whenever the widget is selected. If specified as an empty string then no special color is used for displaying when the widget is selected.

Name: **selectImage**
Class: **SelectImage**
Switch: **–selectimage**

Specifies an image to display (in place of the **image** option) when the radiobutton is selected. This option is ignored unless the **image** option has been specified.

Name: **state**
Class: **State**
Switch: **–state**

Specifies one of three states for the radiobutton: **normal**, **active**, or **disabled**. In normal state the radiobutton is displayed using the **foreground** and **background** options. The active state is typically used when the pointer is over the radiobutton. In active state the radiobutton is displayed using the **activeForeground** and **activeBackground** options. Disabled state means that the radiobutton should be insensitive: the default bindings will refuse to activate the widget and will ignore mouse button presses. In this state the **disabledForeground** and **background** options determine how the radiobutton is displayed.

Name: **value**
Class: **Value**
Switch: **–value**

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

Name: **variable**
Class: **Variable**
Switch: **–variable**

Specifies reference to a 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 `\$Tk::selectedButton`.

Name: **width**
Class: **Width**
Switch: **–width**

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

DESCRIPTION

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

A radiobutton is a widget that displays a textual string, bitmap or image and a diamond or circle called an *indicator*. If text is displayed, it must all be in a single font, but it can occupy multiple lines on the screen (if it contains newlines or if wrapping occurs because of the **wrapLength** option) and one of the characters may optionally be underlined using the **underline** option. A radiobutton has all of the behavior of a simple button: it can display itself in either of three different ways, according to the **state** option; it can be made to appear raised, sunken, or flat; it can be made to flash; and it invokes a *Tk callback\Tk::callbacks* whenever mouse button 1 is clicked over the check button.

In addition, radiobuttons can be *selected*. If a radiobutton is selected, the indicator is normally drawn with a selected appearance, and a Tcl variable associated with the radiobutton is set to a particular value (normally 1). Under Unix, the indicator is drawn with a sunken relief and a special color. Under Windows, the indicator is drawn with a round mark inside. If the radiobutton is not selected, then the indicator is drawn with a deselected appearance, and the associated variable is set to a different value (typically 0). Under Unix, the indicator is drawn with a raised relief and no special color. Under Windows, the indicator is drawn without a round mark inside. Typically, several radiobuttons share a single variable and the value of the variable indicates which radiobutton is to be selected. When a radiobutton is selected it sets the value of the variable to indicate that fact; each radiobutton also monitors the value of the variable and automatically selects and deselects itself when the variable's value changes. By default the variable **selectedButton** is used; its contents give the name of the button that is selected, or the empty string if no button associated

with that variable is selected. The name of the variable for a radiobutton, plus the variable to be stored into it, may be modified with options on the command line or in the option database. Configuration options may also be used to modify the way the indicator is displayed (or whether it is displayed at all). By default a radiobutton is configured to select itself on button clicks.

WIDGET METHODS

The **Radiobutton** method creates a widget object. This object supports the **configure** and **cget** methods described in *Tk::options* which can be used to enquire and modify the options described above. The widget also inherits all the methods provided by the generic *Tk::Widget|Tk::Widget* class.

The following additional methods are available for radiobutton widgets:

$radiobutton`->`**deselect**

Deselects the radiobutton and sets the associated variable to an empty string. If this radiobutton was not currently selected, the command has no effect.

$radiobutton`->`**flash**

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

$radiobutton`->`**invoke**

Does just what would have happened if the user invoked the radiobutton with the mouse: selects the button and invokes its associated Tcl command, if there is one. The return value is the return value from the Tcl command, or an empty string if there is no command associated with the radiobutton. This command is ignored if the radiobutton's state is **disabled**.

$radiobutton`->`**select**

Selects the radiobutton and sets the associated variable to the value corresponding to this widget.

BINDINGS

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

[1] On Unix systems, a radiobutton activates whenever the mouse passes over it and deactivates whenever the mouse leaves the radiobutton. On Mac and Windows systems, when mouse button 1 is pressed over a radiobutton, the button activates whenever the mouse pointer is inside the button, and deactivates whenever the mouse pointer leaves the button.

[2] When mouse button 1 is pressed over a radiobutton it is invoked (it becomes selected and the command associated with the button is invoked, if there is one).

[3] When a radiobutton has the input focus, the space key causes the radiobutton to be invoked.

If the radiobutton's state is **disabled** then none of the above actions occur: the radiobutton is completely non–responsive.

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

KEYWORDS

radiobutton, widget

NAME

Tk::ROText – ‘readonly’ perl/Tk Text widget

=for pm Tk/ROText.pm

=for category Derived Widgets

SYNOPSIS

```
use Tk::ROText;
...
$ro = $mw->ROText(?options,...?);
```

DESCRIPTION

This "*IS A*" text widget with all bindings removed that would alter the contents of the text widget. The contents can still be modified via method calls.

KEYS

widget, text, readonly

SEE ALSO

Tk::Text

NAME

Tk::Scale – Create and manipulate Scale widgets

=for category Tk Widget Classes

SYNOPSIS

$scale = *$parent*->**Scale**(*?options?*);

STANDARD OPTIONS

–activebackground –font –highlightthickness –repeatinterval –background –foreground –orient –takefocus –borderwidth –highlightbackground –relief –troughcolor –cursor –highlightcolor –repeatdelay

See *Tk::options* for details of the standard options.

WIDGET–SPECIFIC OPTIONS

Name: **bigIncrement**
Class: **BigIncrement**
Switch: **–bigincrement**

Some interactions with the scale cause its value to change by ‘‘large‘‘ increments; this option specifies the size of the large increments. If specified as 0, the large increments default to 1/10 the range of the scale.

Name: **command**
Class: **Command**
Switch: **–command**

Specifies the prefix of a *Tk callback\Tk::callbacks* to invoke whenever the scale‘s value is changed via a method. The actual command consists of this option followed by a space and a real number indicating the new value of the scale.

Name: **digits**
Class: **Digits**
Switch: **–digits**

An integer specifying how many significant digits should be retained when converting the value of the scale to a string. If the number is less than or equal to zero, then the scale picks the smallest value that guarantees that every possible slider position prints as a different string.

Name: **from**
Class: **From**
Switch: **–from**

A real value corresponding to the left or top end of the scale.

Name: **label**
Class: **Label**
Switch: **–label**

A string to display as a label for the scale. For vertical scales the label is displayed just to the right of the top end of the scale. For horizontal scales the label is displayed just above the left end of the scale. If the option is specified as an empty string, no label is displayed.

Name: **length**
Class: **Length**
Switch: **–length**

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

Name: **resolution**
Class: **Resolution**
Switch: **–resolution**

A real value specifying the resolution for the scale. If this value is greater than zero then the scale's value will always be rounded to an even multiple of this value, as will tick marks and the endpoints of the scale. If the value is less than zero then no rounding occurs. Defaults to 1 (i.e., the value will be integral).

Name: **showValue**
Class: **ShowValue**
Switch: **–showvalue**

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

Name: **sliderLength**
Class: **SliderLength**
Switch: **–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**.

Name: **sliderRelief**
Class: **SliderRelief**
Switch: **–sliderrelief**

Specifies the relief to use when drawing the slider, such as **raised** or **sunken**.

Name: **state**
Class: **State**
Switch: **–state**

Specifies one of three states for the scale: **normal**, **active**, or **disabled**. If the scale is disabled then the value may not be changed and the scale won't activate. If the scale is active, the slider is displayed using the color specified by the **activeBackground** option.

Name: **tickInterval**
Class: **TickInterval**
Switch: **–tickinterval**

Must be a real value. Determines the spacing between numerical tick marks displayed below or to the left of the slider. If 0, no tick marks will be displayed.

Name: **to**
Class: **To**
Switch: **–to**

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

Name: **variable**
Class: **Variable**
Switch: **–variable**

Specifies the name of a global variable to link to the scale. Whenever the value of the variable changes, the scale will update to reflect this value. Whenever the scale is manipulated interactively, the variable will be modified to reflect the scale's new value.

Name: **width**
Class: **Width**
Switch: **–width**

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

DESCRIPTION

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

A scale is a widget that displays a rectangular *trough* and a small *slider*. The trough corresponds to a range of real values (determined by the **from**, **to**, and **resolution** options), and the position of the slider selects a particular real value. The slider's position (and hence the scale's value) may be adjusted with the mouse or keyboard as described in the *"BINDINGS"* section below. Whenever the scale's value is changed, a Tcl command is invoked (using the **command** option) to notify other interested widgets of the change. In addition, the value of the scale can be linked to a Tcl variable (using the **variable** option), so that changes in either are reflected in the other.

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

WIDGET METHODS

The **Scale** method creates a widget object. This object supports the **configure** and **cget** methods described in *Tk::options* which can be used to enquire and modify the options described above. The widget also inherits all the methods provided by the generic *Tk::Widget|Tk::Widget* class.

The following additional methods are available for scale widgets:

$scale->**coords***(?value?)*

Returns a list whose elements are the x and y coordinates of the point along the centerline of the trough that corresponds to *value*. If *value* is omitted then the scale's current value is used.

$scale->**get***(?x, y?)*

If *x* and *y* are omitted, returns the current value of the scale. If *x* and *y* are specified, they give pixel coordinates within the widget; the command returns the scale value corresponding to the given pixel. Only one of *x* or *y* is used: for horizontal scales *y* is ignored, and for vertical scales *x* is ignored.

$scale->**identify***(x, y)*

Returns a string indicating what part of the scale lies under the coordinates given by *x* and *y*. A return value of **slider** means that the point is over the slider; **trough1** means that the point is over the portion of the slider above or to the left of the slider; and **trough2** means that the point is over the portion of the slider below or to the right of the slider. If the point isn't over one of these elements, an empty string is returned.

$scale->**set***(value)*

This command is invoked to change the current value of the scale, and hence the position at which the slider is displayed. *Value* gives the new value for the scale. The command has no effect if the scale is disabled.

BINDINGS

Tk automatically creates class bindings for scales that give them the following default behavior. Where the behavior is different for vertical and horizontal scales, the horizontal behavior is described in parentheses.

[1] If button 1 is pressed in the trough, the scale's value will be incremented or decremented by the value of the **resolution** option so that the slider moves in the direction of the cursor. If the button is held down, the action auto–repeats.

[2] If button 1 is pressed over the slider, the slider can be dragged with the mouse.

[3] If button 1 is pressed in the trough with the Control key down, the slider moves all the way to the end of its range, in the direction towards the mouse cursor.

[4] If button 2 is pressed, the scale‘s value is set to the mouse position. If the mouse is dragged with button 2 down, the scale‘s value changes with the drag.

[5] The Up and Left keys move the slider up (left) by the value of the **resolution** option.

[6] The Down and Right keys move the slider down (right) by the value of the **resolution** option.

[7] Control−Up and Control−Left move the slider up (left) by the value of the **bigIncrement** option.

[8] Control−Down and Control−Right move the slider down (right) by the value of the **bigIncrement** option.

[9] Home moves the slider to the top (left) end of its range.

[10] End moves the slider to the bottom (right) end of its range.

If the scale is disabled using the **state** option then none of the above bindings have any effect.

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

KEYWORDS

scale, slider, trough, widget

NAME

Tk::Scrollbar – Create and manipulate Scrollbar widgets

=for category Tk Widget Classes

SYNOPSIS

$scrollbar = *$parent*->**Scrollbar***(?options?);*

STANDARD OPTIONS

–activebackground –highlightbackground–orient –takefocus –background
–highlightcolor –relief –troughcolor –borderwidth –highlightthickness
–repeatdelay –cursor –jump –repeatinterval

See *Tk::options* for details of the standard options.

WIDGET–SPECIFIC OPTIONS

Name: **activeRelief**
Class: **ActiveRelief**
Switch: **–activerelief**

Specifies the relief to use when displaying the element that is active, if any. Elements other than the active element are always displayed with a raised relief.

Name: **command**
Class: **Command**
Switch: **–command**

Specifies a callback to invoke to change the view in the widget associated with the scrollbar. When a user requests a view change by manipulating the scrollbar, the callback is invoked. The callback is passed additional arguments as described later. This option almost always has a value such as **[xview => $widget]** or **[yview => $widget],** consisting of the a widget object and either **xview** (if the scrollbar is for horizontal scrolling) or **yview** (for vertical scrolling). All scrollable widgets have **xview** and **yview** methods that take exactly the additional arguments appended by the scrollbar as described in *"SCROLLING COMMANDS"* below.

Name: **elementBorderWidth**
Class: **BorderWidth**
Switch: **–elementborderwidth**

Specifies the width of borders drawn around the internal elements of the scrollbar (the two arrows and the slider). The value may have any of the forms acceptable to **Tk_GetPixels**. If this value is less than zero, the value of the **borderWidth** option is used in its place.

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

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

indicate which portion of the document is visible in the associated window. For example, if the slider in a vertical scrollbar covers the top third of the area between the two arrows, it means that the associated window displays the top third of its document.

Scrollbars can be used to adjust the view in the associated window by clicking or dragging with the mouse. See *"BINDINGS"* below for details.

ELEMENTS

A scrollbar displays five elements, which are referred to in the methods for the scrollbar:

arrow1

The top or left arrow in the scrollbar.

trough1

The region between the slider and **arrow1**.

slider

The rectangle that indicates what is visible in the associated widget.

trough2

The region between the slider and **arrow2**.

arrow2

The bottom or right arrow in the scrollbar.

WIDGET METHODS

The **Scrollbar** method creates a widget object. This object supports the **configure** and **cget** methods described in *Tk::options* which can be used to enquire and modify the options described above. The widget also inherits all the methods provided by the generic *Tk::Widget|Tk::Widget* class.

The following additional methods are available for scrollbar widgets:

`$scrollbar->`**activate***(?element?)*

Marks the element indicated by *element* as active, which causes it to be displayed as specified by the **activeBackground** and **activeRelief** options. The only element values understood by this command are **arrow1**, **slider**, or **arrow2**. If any other value is specified then no element of the scrollbar will be active. If *element* is not specified, the command returns the name of the element that is currently active, or an empty string if no element is active.

`$scrollbar->`**delta***(deltaX, deltaY)*

Returns a real number indicating the fractional change in the scrollbar setting that corresponds to a given change in slider position. For example, if the scrollbar is horizontal, the result indicates how much the scrollbar setting must change to move the slider *deltaX* pixels to the right (*deltaY* is ignored in this case). If the scrollbar is vertical, the result indicates how much the scrollbar setting must change to move the slider *deltaY* pixels down. The arguments and the result may be zero or negative.

`$scrollbar->`**fraction***(x, y)*

Returns a real number between 0 and 1 indicating where the point given by *x* and *y* lies in the trough area of the scrollbar. The value 0 corresponds to the top or left of the trough, the value 1 corresponds to the bottom or right, 0.5 corresponds to the middle, and so on. *X* and *y* must be pixel coordinates relative to the scrollbar widget. If *x* and *y* refer to a point outside the trough, the closest point in the trough is used.

`$scrollbar->`**get**

Returns the scrollbar settings in the form of a list whose elements are the arguments to the most recent **set** method.

$scrollbar->**identify**(*x, y*)

Returns the name of the element under the point given by *x* and *y* (such as **arrow1**), or an empty string if the point does not lie in any element of the scrollbar. *X* and *y* must be pixel coordinates relative to the scrollbar widget.

$scrollbar->**set**(*first, last*)

This command is invoked by the scrollbar's associated widget to tell the scrollbar about the current view in the widget. The command takes two arguments, each of which is a real fraction between 0 and 1. The fractions describe the range of the document that is visible in the associated widget. For example, if *first* is 0.2 and *last* is 0.4, it means that the first part of the document visible in the window is 20% of the way through the document, and the last visible part is 40% of the way through.

SCROLLING COMMANDS

When the user interacts with the scrollbar, for example by dragging the slider, the scrollbar notifies the associated widget that it must change its view. The scrollbar makes the notification by evaluating a callback specified as the scrollbar's **–command** option. The callback may take several forms. In each case, the intial arguments passed are those specified in the **–command** callback itself, which usually has a form like [**yview** => *$widget]*. (Which will invoke *$widget*->**yview**(*...*) where the ... part is as below. See *Tk::callbacks* for details.) The callback is passed additional arguments as follows:

moveto,*fraction*

Fraction is a real number between 0 and 1. The widget should adjust its view so that the point given by *fraction* appears at the beginning of the widget. If *fraction* is 0 it refers to the beginning of the document. 1.0 refers to the end of the document, 0.333 refers to a point one–third of the way through the document, and so on.

scroll,*number,***units**

The widget should adjust its view by *number* units. The units are defined in whatever way makes sense for the widget, such as characters or lines in a text widget. *Number* is either 1, which means one unit should scroll off the top or left of the window, or –1, which means that one unit should scroll off the bottom or right of the window.

scroll,*number*,**page**

The widget should adjust its view by *number* pages. It is up to the widget to define the meaning of a page; typically it is slightly less than what fits in the window, so that there is a slight overlap between the old and new views. *Number* is either 1, which means the next page should become visible, or –1, which means that the previous page should become visible.

OLD COMMAND SYNTAX

In versions of Tk before 4.0, the **set** and **get** widget commands used a different form. This form is still supported for backward compatibility, but it is deprecated. In the old command syntax, the **set** method has the following form:

$scrollbar->**set**(*totalUnits, windowUnits, firstUnit, lastUnit*)

In this form the arguments are all integers. *TotalUnits* gives the total size of the object being displayed in the associated widget. The meaning of one unit depends on the associated widget; for example, in a text editor widget units might correspond to lines of text. *WindowUnits* indicates the total number of units that can fit in the associated window at one time. *FirstUnit* and *lastUnit* give the indices of the first and last units currently visible in the associated window (zero corresponds to the first unit of the object).

Under the old syntax the **get** method returns a list of four integers, consisting of the *totalUnits*, *windowUnits*, *firstUnit*, and *lastUnit* values from the last **set** method.

The callbacks generated by scrollbars also have a different form when the old syntax is being used, the callback is passed a single argument:

unit *Unit* is an integer that indicates what should appear at the top or left of the associated widget's window. It has the same meaning as the *firstUnit* and *lastUnit* arguments to the **set** method.

The most recent **set** method determines whether or not to use the old syntax. If it is given two real arguments then the new syntax will be used in the future, and if it is given four integer arguments then the old syntax will be used.

BINDINGS

Tk automatically creates class bindings for scrollbars that give them the following default behavior. If the behavior is different for vertical and horizontal scrollbars, the horizontal behavior is described in parentheses.

[1] Pressing button 1 over **arrow1** causes the view in the associated widget to shift up (left) by one unit so that the document appears to move down (right) one unit. If the button is held down, the action auto–repeats.

[2] Pressing button 1 over **trough1** causes the view in the associated widget to shift up (left) by one screenful so that the document appears to move down (right) one screenful. If the button is held down, the action auto–repeats.

[3] Pressing button 1 over the slider and dragging causes the view to drag with the slider. If the **jump** option is true, then the view doesn't drag along with the slider; it changes only when the mouse button is released.

[4] Pressing button 1 over **trough2** causes the view in the associated widget to shift down (right) by one screenful so that the document appears to move up (left) one screenful. If the button is held down, the action auto–repeats.

[5] Pressing button 1 over **arrow2** causes the view in the associated widget to shift down (right) by one unit so that the document appears to move up (left) one unit. If the button is held down, the action auto–repeats.

[6] If button 2 is pressed over the trough or the slider, it sets the view to correspond to the mouse position; dragging the mouse with button 2 down causes the view to drag with the mouse. If button 2 is pressed over one of the arrows, it causes the same behavior as pressing button 1.

[7] If button 1 is pressed with the Control key down, then if the mouse is over **arrow1** or **trough1** the view changes to the very top (left) of the document; if the mouse is over **arrow2** or **trough2** the view changes to the very bottom (right) of the document; if the mouse is anywhere else then the button press has no effect.

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

[9] In vertical scrollbars Control–Up and Control–Down have the same behavior as mouse clicks over **trough1** and **trough2**, respectively. In horizontal scrollbars these keys have no effect.

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

[11] In horizontal scrollbars Control–Up and Control–Down have the same behavior as mouse clicks over **trough1** and **trough2**, respectively. In vertical scrollbars these keys have no effect.

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

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

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

SEE ALSO

Tk::callbacks\Tk::callbacks Tk::Scrolled\Tk::Scrolled

KEYWORDS

scrollbar, widget

NAME

Tk::Scrolled – Create a widget with attached scrollbar(s)

=for category Derived Widgets

=for index_group Perl/Tk Constructs

SYNOPSIS

$whatever = *$parent*->**Scrolled**(*Whatever ?,*–**scrollbars**=>*where? ?,...?);*

DESCRIPTION

To stop a flood of **ScrlWhatever** widgets Perl/Tk introcuded the special constructor **Scrolled**. **Scrolled** creates a widget of the given Class *Whatever* with attached *scrollbar(s)\Tk::Scrollbar*.

OPTIONS

All options beside **–scrollbars** explained below are passed to the *Whatever* widget constructor.

–scrollbars

Expects as argument the position where the scrollbars should be created: **w**, **e** or **n**, **s** or a combination of them. If the one or both positions are prefixed with **o** the scrollbar will only show up if there is a 'real' need to scroll.

ADVERTISED SUBWIDGETS

See *Tk::mega/"Subwidget"* how to use advertised widgets.

scrolled

the scrolled widget

widget

same as **scrolled** above. *widget* is the kind of widget passed to scrolled as first argument in all lowercase.

xscrollbar

the **Scrollbar** widget used for horizontal scrolling (if it exists)

yscrollbar

the **Scrollbar** widget used for vertical scrolling (if it exists)

corner

a frame in the corner between the vertical and horizontal scrolbar

BUGS

If a widget does not support *–{x,y}scrollcommand* options, **Scrolled** does not complain if the specified widget class does not support them. E.g.,

$parent–**Scrolled**(**'Button',** ...)

One does not get an error message or warning when one tries to configure scrollbars after the widget construction:

$ascrolled–**configure**(**–scrollbars** = **'e');**

SEE ALSO

Tk::Scrollbar\Tk::Scrollbar

KEYWORDS

scrolled, scrollbar

NAME

Tk::Selection – Manipulate the X selection

=for category User Interaction

SYNOPSIS

$widget->**Selection***Option?(args)?*

DESCRIPTION

This command provides an interface to the X selection mechanism and implements the full selection functionality described in the X Inter–Client Communication Conventions Manual (ICCCM).

The widget object used to invoke the methods below determines which display is used to access the selection. In order to avoid conflicts with **selection** methods of widget classes (e.g. **Text**) this set of methods uses the prefix **Selection**. The following methods are currently supported:

$widget->**SelectionClear***?(***–selection***=>selection)?***

If *selection* exists anywhere on *$widget's* display, clear it so that no window owns the selection anymore. *Selection* specifies the X selection that should be cleared, and should be an atom name such as PRIMARY or CLIPBOARD; see the Inter–Client Communication Conventions Manual for complete details. *Selection* defaults to PRIMARY. Returns an empty string.

$widget->**SelectionGet***?(?***–selection***=>selection?,?***–type***=>type?)?*

Retrieves the value of *selection* from *$widget's* display and returns it as a result. *Selection* defaults to PRIMARY.

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 *format* is not STRING then things get messy, the following description is from the Tcl/Tk man page as yet incompetely translated for the perl version – it is misleading at best.

If the selection is returned in a non–string format, such as INTEGER or ATOM, the **SelectionGet** converts it to a list of perl values: atoms are converted to their textual names, and anything else is converted integers.

A goal of the perl port is to provide better handling of different formats than Tcl/Tk does, which should be possible given perl's wider range of ''types''. Although some thought went into this in very early days of perl/Tk what exactly happens is still "not quite right" and subject to change.

$widget->**SelectionHandle***(?***–selection***=>selection?,?***–type***=>type?,?***–format***=>format? callback*)

Creates a handler for selection requests, such that *callback* will be executed whenever *selection* is owned by *$widget* and someone attempts to retrieve it in the form given by *type* (e.g. *type* is specified in the **selection get** command). *Selection* defaults to PRIMARY, *type* defaults to STRING, and *format* defaults to STRING. If *callback* is an empty string then any existing handler for *$widget, type*, and *selection* is removed.

When *selection* is requested, *$widget* is the selection owner, and *type* is the requested type, *callback* will be executed with two additional arguments. The two additional arguments 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 *callback* with increasing *offset* values. If *callback* 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 *callback*‘s result is equal to *maxBytes* then *callback* 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 *callback* returns an error (e.g. via **die**) 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 not STRING then things get messy, the following description is from the Tcl/Tk man page as yet untranslated for the perl version − it is misleading at best.

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 many selection requesters, except Tcl/Tk. If Tcl/Tk is being used to retrieve the selection then the value is converted back to a string at the requesting end, so *format* is irrelevant.

A goal of the perl port is to provide better handling of different formats than Tcl/Tk does, which should be possible given perl‘s wider range of ‘‘types‘‘. Although some thought went into this in very early days of perl/Tk what exactly happens is still "not quite right" and subject to change.

$widget−>**SelectionOwner***?(*−**selection***=>selection)?*

SelectionOwner returns the window in this application that owns *selection* on the display containing *$widget*, or an empty string if no window in this application owns the selection. *Selection* defaults to PRIMARY.

$widget−>**SelectionOwn***?(?*−**command***=>callback?,?*−**selection***=>selection?)?*

SelectionOwn causes *$widget* to become the new owner of *selection* on *$widget*‘*s* display, returning an empty string as result. The existing owner, if any, is notified that it has lost the selection. If *callback* is specified, it will be executed when some other window claims ownership of the selection away from *$widget*. *Selection* defaults to PRIMARY.

KEYWORDS

clear, format, handler, ICCCM, own, selection, target, type

NAME

send – Execute a command in a different application

=for category Tk Generic Methods

SYNOPSIS

$result = *$widget*->**send**(*?options,?app=>cmd?arg arg ...?)*

DESCRIPTION

This method arranges for *cmd* (and *arg*s) to be 'sent' to the application named by *app*. It returns the result or an error (hence above should probably be 'wrapped' in **eval{}** and $@ tested). *App* may be the name of any application whose main window is on the display containing the sender's main window; it need not be within the same process. If no *arg* arguments are present, then the string to be sent is contained entirely within the *cmd* argument. If one or more *arg*s are present, they are concatenated separated by white space to form the string to be sent.

If the initial arguments of the call begin with ''–'' they are treated as options. The following options are currently defined:

–async

Requests asynchronous invocation. In this case the **send** command will complete immediately without waiting for *cmd* to complete in the target application; no result will be available and errors in the sent command will be ignored. If the target application is in the same process as the sending application then the **–async** option is ignored.

— Serves no purpose except to terminate the list of options. This option is needed only if *app* could contain a leading ''–'' character.

APPLICATION NAMES

The name of an application is set initially from the name of the program or script that created the application. You can query and change the name of an application with the **appname** method.

WHAT IS A SEND

The **send** mechanism was designed to allow Tcl/Tk applications to send Tcl Scripts to each other. This does not map very well onto perl/Tk. Perl/Tk "sends" a string to *app*, what happens as a result of this depends on the receiving application. If the other application is a Tcl/Tk4.* application it will be treated as a Tcl Script. If the "other" application is perl/Tk application (including sends to self) then the string is passed as an argument to a method call of the following form:

$mainwindow->**Receive**(*string);*

There is a default (AutoLoaded) **Tk::Receive** which returns an error to the sending application. A particular application may define its own **Receive** method in any class in **MainWindow**'s inheritance tree to do whatever it sees fit. For example it could **eval** the string, possibly in a **Safe** "compartment".

If a Tcl/Tk application "sends" anything to a perl/Tk application then the perl/Tk application would have to attempt to interpret the incoming string as a Tcl Script. Simple cases are should not be too hard to emulate (split on white space and treat first element as "command" and other elements as arguments).

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. Hence the default behaviour outlined above. (With the availability of **Safe** it may make sense to relax default behaviour a little.)

Unmonitored **eval**'ing of these incoming "scripts" can cause perl to read and write 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, core 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**.

SEE ALSO

Perl‘s **eval** perl‘s **Safe** Module system‘s administrator/corporate security guidelines etc.

KEYWORDS

application, name, remote execution, security, send

NAME

Tk::Submethods – add aliases for tk sub–commands

=for pm Tk/Submethods.pm

=for category Implementation

SYNOPSIS

```
use Tk::Submethods ( 'command1' => [qw(sub1 sub2 sub3)],
                     'command2' => [qw(sub1 sub2 sub3)]);
```

DESCRIPTION

Creates `->commandSub(...)` as an alias for `->command('sub',...)` e.g. `->grabRelease` for `->grab('release')`.

For each command/subcommand pair this creates a closure with command and subcommand as bound lexical variables and assigns a reference to this to a 'glob' in the callers package.

Someday the sub–commands may be created directly in the C code.

NAME

Tk::DragDrop::SunConst – Constants for Sun‘s Drag&Drop protocol

=for pm DragDrop/DragDrop/SunConst.pm

=for category Experimental Modules

DESCRIPTION

This module defines symbolic name subs for the numeric constants that make up Sun‘s Drag&Drop protocol. They are in this module with Exporter as they are shared between the two halves (Dropper and Receiver) of the protocol.

NAME

Tk::Table – Scrollable 2 dimensional table of Tk widgets

=for pm Tk/Table.pm

=for category Tk Geometry Management

SYNOPSIS

```
use Tk::Table;

$table = $parent->Table(-rows => number,
                        -columns => number,
                        -scrollbars => anchor,
                        -fixedrows => number,
                        -fixedcolumns => number,
                        -takefocus => boolean);

$widget = $table->Button(...);

$old = $table->put($row,$col,$widget);
$old = $table->put($row,$col,"Text");  # simple Label
$widget = $table->get($row,$col);

$cols = $table->totalColumns;
$rows = $table->totalRows;

$table->see($widget);
$table->see($row,$col);

($row,$col) = $table->Posn($widget);
```

DESCRIPTION

Tk::Table is an all–perl widget/geometry manager which allows a two dimensional table of arbitary perl/Tk widgets to be displayed.

Entries in the Table are simply ordinary perl/Tk widgets. They should be created with the Table as their parent. Widgets are positioned in the table using:

```
$table->put($row,$col,$widget)
```

All the widgets in each column are set to the same width – the requested width of the widest widget in the column. Likewise, all the widgets in each row are set to the same height – the requested height of the tallest widget in the column.

A number of rows and/or columns can be marked as 'fixed' – and so can serve as 'headings' for the remainder the rows which are scrollable.

The requested size of the table as a whole is such that the number of rows specified by –rows (default 10), and number of columns specified by –columns (default 10) can be displayed.

If the Table is told it can take the keyboard focus then cursor and scroll keys scroll the displayed widgets.

The Table will create and manage its own scrollbars if requested via –scrollbars.

The Tk::Table widget is derived from a Tk::Frame, so inherits all its configure options.

BUGS / Snags / Possible enhancements

- Very large Tables consume a lot of X windows
- No equivalent of pack's –anchor/–pad etc. options

NAME

Tcl vs perl − very old suspect documentation on porting.

=for category Other Modules and Languages

DESCRIPTION

This isn‘t really a .pod yet, nor is it Tcl vs perl it is a copy of John‘s comparison of Malcolm‘s original perl/Tk port with the current one. It is also out−of−date in places.

```
From: john@WPI.EDU (John Stoffel )

Here are some thoughts on the new Tk extension and how I think the
organization of the commands looks.  Mostly, I'm happy with it, it
makes some things more organized and more consistent with tcl/tk, but
since the overlying language is so different, I don't think we need to
follow exactly the tcl/tk model for how to call the language.

The basic structure of the Tk program is:

    require Tk;

    $top = MainWindow->new();

    #
    # create widgets
    #

    Tk::MainLoop;

    sub method1 {
    }

    sub methodN {
    }

This is pretty much the same as tkperl5a5, with some cosmetic naming
changes, and some more useful command name and usage changes.  A quick
comparison in no particular order follows:

tkperl5a5                              Tk
-------------------------------        -----------------------------------
$top=tkinit(name,display,sync);        $top=MainWindow->new();

tkpack $w, ... ;                 $w->pack(...)

$w = Class::new($top, ...);      $w = $top->Class(...);

tkmainloop;                      Tk::MainLoop;

tkbind($w,"<key>",sub);                  $w->bind("<key>",sub);

tkdelete($w, ...);               $w->delete(...);

$w->scanmark(...);               $w->scan("mark", ...);

$w->scandragto(...);             $w->scan("dragto", ...);

$w->tkselect();                          $w->Select();

$w->selectadjust(...);                   $w->selection("adjust", ...);

$w->selectto(...);               $w->selection("to", ...);

$w->selectfrom(...);             $w->selection("from", ...);

$w->tkindex(...);                $w->index(...);
```

```
tclcmd("xxx",...);              &Tk::xxx(...)    # all Tk commands, but no Tcl at a
tclcmd("winfo", xxx, $w, ...);  $w->xxx(...);
                                $w->mark(...);
                                $w->tag(...);
$w->grabstatus();               $w->grab("status");
$w->grabrelease(...);           $w->grab("release", ...);
focus($w);                      $w->focus;
update();                       Tk->update();
idletasks();                    Tk->update("idletasks");
wm("cmd",$w, ...);              $w->cmd(...);
destroy($w);                    $w->destroy();
                                Tk::option(...);
                                  $w->OptionGet(name,Class)
                                $w->place(...)
                                Tk::property(...);
$w = Entry::new($parent,...)
is now
$w = $parent->Entry(...)
As this allows new to be inherited from a Window class.
  -method=>x,-slave=>y
 is now
  -command => [x,y]
1st element of list is treated as "method" if y is an object reference.
(You can have -command => [a,b,c,d,e] too; b..e get passed as args).
Object references are now hashes rather than scalars and there
is only ever one such per window.  The Tcl_CmdInfo and PathName
are entries in the hash.
(This allows derived classes to
re-bless the hash and keep their on stuff in it too.)
Tk's "Tcl_Interp" is in fact a ref to "." window.
You can find all the Tk windows descended from it as their object
references get added (by PathName) into this hash.
$w->MainWindow returns this hash from any window.
I think that it should extend to multiple tkinits / Tk->news
with different Display's - if Tk code does.
Finally "bind" passes window as "extra" (or only)
argument. Thus
Tk::Button->bind(<Any-Enter>,"Enter");
Binds Enter events to Tk::Button::Enter by default
but gets called as $w->Enter so derived class of Button can just
```

define its own Enter method. &EvWref and associated globals and race conditions are no longer needed.

One thing to beware of : commands bound to events with $widget->bind follow same pattern, but get passed extra args :

```
$widget->bind(<Any-1>,[sub {print shift}, $one, $two ]);
```

When sub gets called it has :

```
   $widget $one $two
```

passed.

1st extra arg is reference to the per-widget hash that serves as the perl object for the widget.

Every time an XEvent a reference to a special class is placed in the widget hash. It can be retrieved by $w->XEvent method.

The methods of the XEvent class are the Tcl/Tk % special characters.

Thus:

```
$widget->bind(<Any-KeyPress>,
              sub {
               my $w = shift;
               my $e = $w->XEvent;
               print $w->PathName," ",$e->A," pressed ,$e->xy,"\n");
              });
```

XEvent->xy is a special case which returns "@" . $e->x . "," . $e->y which is common in Text package.

Because of passing a blessed widget hash to "bound" subs they can be bound to (possibly inherited) methods of the widget's class:

```
Class->bind(<Any-Down>,Down);

sub Class::Down
{
 my $w = shift;
 # handle down arrow
}
```

Also:

-command and friends can take a list the 1st element can be a ref to as sub or a method name. Remaining elements are passed as args to the sub at "invoke" time. Thus :

```
$b= $w->Button(blah blah, '-command' => [sub{print shift} , $fred ]);
```

Should do the trick, provided $fred is defined at time of button creation.

Thus 1st element of list is equivalent to Malcolm's -method and second would be his -slave. Any further elements are a bonus and avoid having to pass ref to an array/hash as a slave.

NAME

Tk::Text – Create and manipulate Text widgets

=for category Tk Widget Classes

SYNOPSIS

text `$text` ?*options*?

–background –highlightbackground –insertontime –selectborderwidth –borderwidth –highlightcolor –insertwidth –selectforeground –cursor –highlightthickness –padx –setgrid –exportselection –insertbackground –pady –takefocus –font –insertborderwidth –relief –xscrollcommand –foreground –insertofftime –selectbackground –yscrollcommand

WIDGET–SPECIFIC OPTIONS

Name: **height**
Class: **Height**
Switch: **–height**

Specifies the desired height for the window, in units of characters in the font given by the **–font** option. Must be at least one.

Name: **spacing1**
Class: **Spacing1**
Switch: **–spacing1**

Requests additional space above each text line in the widget, using any of the standard forms for screen distances. If a line wraps, this option only applies to the first line on the display. This option may be overriden with **–spacing1** options in tags.

Name: **spacing2**
Class: **Spacing2**
Switch: **–spacing2**

For lines that wrap (so that they cover more than one line on the display) this option specifies additional space to provide between the display lines that represent a single line of text. The value may have any of the standard forms for screen distances. This option may be overriden with **–spacing2** options in tags.

Name: **spacing3**
Class: **Spacing3**
Switch: **–spacing3**

Requests additional space below each text line in the widget, using any of the standard forms for screen distances. If a line wraps, this option only applies to the last line on the display. This option may be overriden with **–spacing3** options in tags.

Name: **state**
Class: **State**
Switch: **–state**

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

Name: **tabs**
Class: **Tabs**
Switch: **–tabs**

Specifies a set of tab stops for the window. The option‘s value consists of a list of screen distances giving the positions of the tab stops. Each position may optionally be followed in the next list element by one of the keywords **left**, **right**, **center**, or **numeric**, which specifies how to justify text relative to

the tab stop. **Left** is the default; it causes the text following the tab character to be positioned with its left edge at the tab position. **Right** means that the right edge of the text following the tab character is positioned at the tab position, and **center** means that the text is centered at the tab position. **Numeric** means that the decimal point in the text is positioned at the tab position; if there is no decimal point then the least significant digit of the number is positioned just to the left of the tab position; if there is no number in the text then the text is right–justified at the tab position. For example, **–tabs => [qw/2c left 4c 6c center/]** creates three tab stops at two–centimeter intervals; the first two use left justification and the third uses center justification. If the list of tab stops does not have enough elements to cover all of the tabs in a text line, then Tk extrapolates new tab stops using the spacing and alignment from the last tab stop in the list. The value of the **tabs** option may be overridden by **–tabs** options in tags. If no **–tabs** option is specified, or if it is specified as an empty list, then Tk uses default tabs spaced every eight (average size) characters.

Name: **width**
Class: **Width**
Switch: **–width**

Specifies the desired width for the window in units of characters in the font given by the **–font** option. If the font doesn‘t have a uniform width then the width of the character ‘‘0’’ is used in translating from character units to screen units.

Name: **wrap**
Class: **Wrap**
Switch: **–wrap**

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

DESCRIPTION

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

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

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

The third form of annotation allows arbitrary windows to be embedded in a text widget. See *"EMBEDDED WINDOWS"* below for more details.

The fourth form of annotation allows Tk images to be embedded in a text widget. See *"EMBEDDED IMAGES"* below for more details.

INDICES

Many of the methods for texts take one or more indices as arguments. An index is a string used to indicate a particular place within a text, such as a place to insert characters or one endpoint of a range of characters to delete. Indices have the syntax

```
base modifier modifier modifier ...
```

Where *base* gives a starting point and the *modifier*s adjust the index from the starting point (e.g. move forward or backward one character). Every index must contain a *base*, but the *modifier*s are optional.

The *base* for an index must have one of the following forms:

line.*char*

Indicates *char*'th character on line *line*. Lines are numbered from 1 for consistency with other UNIX programs that use this numbering scheme. Within a line, characters are numbered from 0. If *char* is **end** then it refers to the newline character that ends the line.

@*x*,*y*

Indicates the character that covers the pixel whose x and y coordinates within the text's window are *x* and *y*.

end Indicates the end of the text (the character just after the last newline).

mark

Indicates the character just after the mark whose name is *mark*.

tag.**first**

Indicates the first character in the text that has been tagged with *tag*. This form generates an error if no characters are currently tagged with *tag*.

tag.**last**

Indicates the character just after the last one in the text that has been tagged with *tag*. This form generates an error if no characters are currently tagged with *tag*.

`$widget`

Indicates the position of the embedded window referenced by `$widget`. This form generates an error if `$widget` does not reference to an embedded window.

imageName

Indicates the position of the embedded image whose name is *imageName*. This form generates an error if there is no embedded image by the given name.

If the *base* could match more than one of the above forms, such as a *mark* and *imageName* both having the same value, then the form earlier in the above list takes precedence. If modifiers follow the base index, each one of them must have one of the forms listed below. Keywords such as **chars** and **wordend** may be abbreviated as long as the abbreviation is unambiguous.

+ *count* **chars**

Adjust the index forward by *count* characters, moving to later lines in the text if necessary. If there are fewer than *count* characters in the text after the current index, then set the index to the last character in the text. Spaces on either side of *count* are optional.

– *count* **chars**

Adjust the index backward by *count* characters, moving to earlier lines in the text if necessary. If there are fewer than *count* characters in the text before the current index, then set the index to the first character in the text. Spaces on either side of *count* are optional.

+ *count* **lines**

Adjust the index forward by *count* lines, retaining the same character position within the line. If there are fewer than *count* lines after the line containing the current index, then set the index to refer to the same character position on the last line of the text. Then, if the line is not long enough to contain a character at the indicated character position, adjust the character position to refer to the last character of the line (the newline). Spaces on either side of *count* are optional.

– *count* lines

Adjust the index backward by *count* lines, retaining the same character position within the line. If there are fewer than *count* lines before the line containing the current index, then set the index to refer to the same character position on the first line of the text. Then, if the line is not long enough to contain a character at the indicated character position, adjust the character position to refer to the last character of the line (the newline). Spaces on either side of *count* are optional.

linestart

Adjust the index to refer to the first character on the line.

lineend

Adjust the index to refer to the last character on the line (the newline).

wordstart

Adjust the index to refer to the first character of the word containing the current index. A word consists of any number of adjacent characters that are letters, digits, or underscores, or a single character that is not one of these.

wordend

Adjust the index to refer to the character just after the last one of the word containing the current index. If the current index refers to the last character of the text then it is not modified.

If more than one modifier is present then they are applied in left–to–right order. For example, the index ‘‘**end – 1 chars**‘‘ refers to the next–to–last character in the text and ‘‘**insert wordstart – 1 c**‘‘ refers to the character just before the first one in the word containing the insertion cursor.

TAGS

The first form of annotation in text widgets is a tag. A tag is a textual string that is associated with some of the characters in a text. Tags may contain arbitrary characters, but it is probably best to avoid using the the characters ‘‘ ‘‘ (space), **+**, or **–**: these characters have special meaning in indices, so tags containing them can‘t be used as indices. There may be any number of tags associated with characters in a text. Each tag may refer to a single character, a range of characters, or several ranges of characters. An individual character may have any number of tags associated with it.

A priority order is defined among tags, and this order is used in implementing some of the tag–related functions described below. When a tag is defined (by associating it with characters or setting its display options or binding callbacks to it), it is given a priority higher than any existing tag. The priority order of tags may be redefined using the ‘‘`$text->`**tagRaise**‘‘ and ‘‘`$text->`**tagLower**‘‘ methods.

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 ‘‘`$text->`**tagConfigure**‘‘ method. If a character has been tagged, then the display options associated with the tag override the default display style. The following options are currently supported for tags:

–background => *color*

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

–bgstipple => *bitmap*

Bitmap specifies a bitmap that is used as a stipple pattern for the background. It may have any of the forms accepted by **Tk_GetBitmap**. If *bitmap* hasn‘t been specified, or if it is specified as an empty string, then a solid fill will be used for the background.

–borderwidth => *pixels*

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

been set for the tag.

–fgstipple => *bitmap*

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

–font => *fontName*

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

–foreground => *color*

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

–justify => *justify*

If the first character of a display line has a tag for which this option has been specified, then *justify* determines how to justify the line. It must be one of **left**, **right**, or **center**. If a line wraps, then the justification for each line on the display is determined by the first character of that display line.

–lmargin1 => *pixels*

If the first character of a text line has a tag for which this option has been specified, then *pixels* specifies how much the line should be indented from the left edge of the window. *Pixels* may have any of the standard forms for screen distances. If a line of text wraps, this option only applies to the first line on the display; the **–lmargin2** option controls the indentation for subsequent lines.

–lmargin2 => *pixels*

If the first character of a display line has a tag for which this option has been specified, and if the display line is not the first for its text line (i.e., the text line has wrapped), then *pixels* specifies how much the line should be indented from the left edge of the window. *Pixels* may have any of the standard forms for screen distances. This option is only used when wrapping is enabled, and it only applies to the second and later display lines for a text line.

–offset => *pixels*

Pixels specifies an amount by which the text's baseline should be offset vertically from the baseline of the overall line, in pixels. For example, a positive offset can be used for superscripts and a negative offset can be used for subscripts. *Pixels* may have any of the standard forms for screen distances.

–overstrike => *boolean*

Specifies whether or not to draw a horizontal rule through the middle of characters. *Boolean* may have any of the forms accepted by **Tk_GetBoolean**.

–relief => *relief*

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

–rmargin => *pixels*

If the first character of a display line has a tag for which this option has been specified, then *pixels* specifies how wide a margin to leave between the end of the line and the right edge of the window. *Pixels* may have any of the standard forms for screen distances. This option is only used when wrapping is enabled. If a text line wraps, the right margin for each line on the display is determined by the first character of that display line.

–spacing1 => *pixels*

Pixels specifies how much additional space should be left above each text line, using any of the standard forms for screen distances. If a line wraps, this option only applies to the first line on the

display.

–spacing2 => *pixels*

For lines that wrap, this option specifies how much additional space to leave between the display lines for a single text line. *Pixels* may have any of the standard forms for screen distances.

–spacing3 => *pixels*

Pixels specifies how much additional space should be left below each text line, using any of the standard forms for screen distances. If a line wraps, this option only applies to the last line on the display.

–state => *state*

State specifies if the text is *hidden* or *normal*. Hidden text is not displayed and takes no space on screen, but further on behaves just as normal text.

–tabs => *tabList*

TabList specifies a set of tab stops in the same form as for the **–tabs** option for the text widget. This option only applies to a display line if it applies to the first character on that display line. If this option is specified as an empty string, it cancels the option, leaving it unspecified for the tag (the default). If the option is specified as a non–empty string that is an empty list, such as **–tabs =** " ", then it requests default 8–character tabs as described for the **tabs** widget option.

–underline => *boolean*

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

–wrap => *mode*

Mode specifies how to handle lines that are wider than the text‘s window. It has the same legal values as the **–wrap** option for the text widget: **none**, **char**, or **word**. If this tag option is specified, it overrides the **–wrap** option for the text widget.

–elide => *value*

If value is true then text covered by the tag is not displayed.

–data => *value*

Allows an arbitrary perl scalar *value* to be associated with the tag.

If a character has several tags associated with it, and if their display options conflict, then the options of the highest priority tag are used. If a particular display option hasn‘t been specified for a particular tag, or if it is specified as an empty string, then that option will never be used; the next–highest–priority tag‘s option will used instead. If no tag specifies a particular display option, then the default style for the widget will be used.

The second purpose for tags is event bindings. You can associate bindings with a tag in much the same way you can associate bindings with a widget class: whenever particular X events occur on characters with the given tag, a given <perl/Tk callback|Tk::callbacks 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 **tagBind** widget method 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 ‘‘`$text->`**mark**’’ text widget method, and their current locations may be determined by using the mark name as an index in methods.

Each mark also has a *gravity*, which is either **left** or **right**. The gravity for a mark specifies what happens to the mark when text is inserted at the point of the mark. If a mark has left gravity, then the mark is treated as if it were attached to the character on its left, so the mark will remain to the left of any text inserted at the mark position. If the mark has right gravity, new text inserted at the mark position will appear to the right of the mark. The gravity for a mark defaults to **right**.

The name space for marks is different from that for tags: the same name may be used for both a mark and a tag, but they will refer to different things.

Two marks have special significance. First, the mark **insert** is associated with the insertion cursor, as described under *"THE INSERTION CURSOR"* below. Second, the mark **current** is associated with the character closest to the mouse and is adjusted automatically to track the mouse position and any changes to the text in the widget (one exception: **current** is not updated in response to mouse motions if a mouse button is down; the update will be deferred until all mouse buttons have been released). Neither of these special marks may be deleted.

EMBEDDED WINDOWS

The third form of annotation in text widgets is an embedded window. Each embedded window annotation causes a window to be displayed at a particular point in the text. There may be any number of embedded windows in a text widget, and any widget may be used as an embedded window (subject to the usual rules for geometry management, which require the text window to be the parent of the embedded window or a descendant of its parent). The embedded window's position on the screen will be updated as the text is modified or scrolled, and it will be mapped and unmapped as it moves into and out of the visible area of the text widget. Each embedded window occupies one character's worth of index space in the text widget, and it may be referred to either by the name of its embedded window or by its position in the widget's index space. If the range of text containing the embedded window is deleted then the window is destroyed.

When an embedded window is added to a text widget with the **widgetCreate** method, several configuration options may be associated with it. These options may be modified later with the **widgetConfigure** method. The following options are currently supported:

–align => *where*

If the window is not as tall as the line in which it is displayed, this option determines where the window is displayed in the line. *Where* must have one of the values **top** (align the top of the window with the top of the line), **center** (center the window within the range of the line), **bottom** (align the bottom of the window with the bottom of the line's area), or **baseline** (align the bottom of the window with the baseline of the line).

–create => *callback*

Specifies a *callback|Tk::callbacks* that may be evaluated to create the window for the annotation. If no **–window** option has been specified for the annotation this *callback* will be evaluated when the annotation is about to be displayed on the screen. *Callback* must create a window for the annotation and return the name of that window as its result. If the annotation's window should ever be deleted, *callback* will be evaluated again the next time the annotation is displayed.

–padx => *pixels*

Pixels specifies the amount of extra space to leave on each side of the embedded window. It may have any of the usual forms defined for a screen distance (see **Tk_GetPixels**).

–pady => *pixels*

Pixels specifies the amount of extra space to leave on the top and on the bottom of the embedded window. It may have any of the usual forms defined for a screen distance (see **Tk_GetPixels**).

–stretch => *boolean*

If the requested height of the embedded window is less than the height of the line in which it is displayed, this option can be used to specify whether the window should be stretched vertically to fill its line. If the **–pady** option has been specified as well, then the requested padding will be retained even if the window is stretched.

–window => *$widget*

Specifies the name of a window to display in the annotation.

EMBEDDED IMAGES

The final form of annotation in text widgets is an embedded image. Each embedded image annotation causes an image to be displayed at a particular point in the text. There may be any number of embedded images in a text widget, and a particular image may be embedded in multiple places in the same text widget. The embedded image's position on the screen will be updated as the text is modified or scrolled. Each embedded image occupies one character's worth of index space in the text widget, and it may be referred to either by its position in the widget's index space, or the name it is assigned when the image is inserted into the text widget with **imageCreate**. If the range of text containing the embedded image is deleted then that copy of the image is removed from the screen.

When an embedded image is added to a text widget with the **image** create method, a name unique to this instance of the image is returned. This name may then be used to refer to this image instance. The name is taken to be the value of the **–name** option (described below). If the **–name** option is not provided, the **–image** name is used instead. If the *imageName* is already in use in the text widget, then **#***nn* is added to the end of the *imageName*, where *nn* is an arbitrary integer. This insures the *imageName* is unique. Once this name is assigned to this instance of the image, it does not change, even though the **–image** or **–name** values can be changed with **image configure**.

When an embedded image is added to a text widget with the **imageCreate** method, several configuration options may be associated with it. These options may be modified later with the **image configure** method. The following options are currently supported:

–align => *where*

If the image is not as tall as the line in which it is displayed, this option determines where the image is displayed in the line. *Where* must have one of the values **top** (align the top of the image with the top of the line), **center** (center the image within the range of the line), **bottom** (align the bottom of the image with the bottom of the line's area), or **baseline** (align the bottom of the image with the baseline of the line).

–image => *image*

Specifies the name of the Tk image to display in the annotation. If *image* is not a valid Tk image, then an error is returned.

–name => *ImageName*

Specifies the name by which this image instance may be referenced in the text widget. If *ImageName* is not supplied, then the name of the Tk image is used instead. If the *imageName* is already in use, *#nn* is appended to the end of the name as described above.

–padx => *pixels*

Pixels specifies the amount of extra space to leave on each side of the embedded image. It may have any of the usual forms defined for a screen distance.

–pady => *pixels*

Pixels specifies the amount of extra space to leave on the top and on the bottom of the embedded image. It may have any of the usual forms defined for a screen distance.

THE SELECTION

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

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

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

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

The **sel** tag is automatically defined when a text widget is created, and it may not be deleted with the ''*$text*−>**tagDelete**'' method. 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 ''*$text*−>**markUnset**'' widget command. The **insert** mark represents the position of the insertion cursor, and the insertion cursor will automatically be drawn at this point whenever the text widget has the input focus.

WIDGET METHODS

The **Text** method creates a widget object. This object supports the **configure** and **cget** methods described in *Tk::options* which can be used to enquire and modify the options described above. The widget also inherits all the methods provided by the generic *Tk::Widget|Tk::Widget* class.

The following additional methods are available for text widgets:

$text−>**bbox***(index)*

Returns a list of four elements describing the screen area of the character given by *index*. The first two elements of the list give the x and y coordinates of the upper−left corner of the area occupied by the character, and the last two elements give the width and height of the area. If the character is only partially visible on the screen, then the return value reflects just the visible part. If the character is not visible on the screen then the return value is an empty list.

$text−>**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.

$text−>**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.

$text−>**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.

$text−>**dlineinfo***(index)*

Returns a list with five elements describing the area occupied by the display line containing *index*. The first two elements of the list give the x and y coordinates of the upper−left corner of the area occupied by the line, the third and fourth elements give the width and height of the area, and the fifth element gives the position of the baseline for the line, measured down from the top of the area. All of this information is measured in pixels. If the current wrap mode is **none** and the line extends beyond the

boundaries of the window, the area returned reflects the entire area of the line, including the portions that are out of the window. If the line is shorter than the full width of the window then the area returned reflects just the portion of the line that is occupied by characters and embedded windows. If the display line containing *index* is not visible on the screen then the return value is an empty list.

`$text->`**dump***(?switches?, index1,* ?*index2*?)

Return the contents of the text widget from *index1* up to, but not including *index2*, including the text and information about marks, tags, and embedded windows. If *index2* is not specified, then it defaults to one character past *index1*. The information is returned in the following format:

key1 value1 index1 key2 value2 index2 ...

The possible *key* values are **text**, **mark**, **tagon**, **tagoff**, and `$text`. The corresponding *value* is the text, mark name, tag name, or window name. The *index* information is the index of the start of the text, the mark, the tag transition, or the window. One or more of the following switches (or abbreviations thereof) may be specified to control the dump:

–all Return information about all elements: text, marks, tags, and windows. This is the default.

–command => *callback*

Instead of returning the information as the result of the dump operation, invoke the *callback* on each element of the text widget within the range. The callback has three arguments appended to it before it is evaluated: the *key*, *value*, and *index*.

–mark

Include information about marks in the dump results.

–tag

Include information about tag transitions in the dump results. Tag information is returned as **tagon** and **tagoff** elements that indicate the begin and end of each range of each tag, respectively.

–text

Include information about text in the dump results. The value is the text up to the next element or the end of range indicated by *index2*. A text element does not span newlines. A multi–line block of text that contains no marks or tag transitions will still be dumped as a set of text seqments that each end with a newline. The newline is part of the value.

–window

Include information about embedded windows in the dump results. The value of a window is its Tk pathname, unless the window has not been created yet. (It must have a create script.) In this case an empty string is returned, and you must query the window by its index position to get more information.

`$text->`**get***(index1,* ?*index2*?)

Return a range of characters from the text. The return value will be all the characters in the text starting with the one whose index is *index1* and ending just before the one whose index is *index2* (the character at *index2* will not be returned). If *index2* is omitted then the single character at *index1* is returned. If there are no characters in the specified range (e.g. *index1* is past the end of the file or *index2* is less than or equal to *index1*) then an empty string is returned. If the specified range contains embedded windows, no information about them is included in the returned string.

`$text->`**image***(option,* ?*arg, arg, ...*?)
`$text->`**image***Option(?arg, arg, ...*?)

This method is used to manipulate embedded images. The behavior of the method depends on the *option* argument that follows the **image** prefix. The following forms of the methods are currently supported:

$text->**imageCget**(*index, option*)

Returns the value of a configuration option for an embedded image. *Index* identifies the embedded image, and *option* specifies a particular configuration option, which must be one of the ones listed in *"EMBEDDED IMAGES"*.

$text->**imageConfigure**(*index, ?option, value, ...?*)

Query or modify the configuration options for an embedded image. If no *option* is specified, returns a list describing all of the available options for the embedded image at *index* (see *Tk::options* for information on the format of this list). If *option* is specified with no *value*, then the command returns a list describing the one named option (this list will be identical to the corresponding sublist of the value returned if no *option* is specified). If one or more *option*−*value* pairs are specified, then the command modifies the given option(s) to have the given value(s); in this case the command returns an empty string. See *"EMBEDDED IMAGES"* for information on the options that are supported.

$text->**imageCreate**(*index, ?option, value, ...?*)

This command creates a new image annotation, which will appear in the text at the position given by *index*. Any number of *option*−*value* pairs may be specified to configure the annotation. Returns a unique identifier that may be used as an index to refer to this image. See *"EMBEDDED IMAGES"* for information on the options that are supported, and a description of the identifier returned.

$text->**imageNames**

Returns a list whose elements are the names of all image instances currently embedded in `$text`.

$text->**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.

$text->**insert**(*index, chars, ?tagList, chars, tagList, ...?*)

Inserts all of the *chars* arguments just before the character at *index*. If *index* refers to the end of the text (the character after the last newline) then the new text is inserted just before the last newline instead. If there is a single *chars* argument and no *tagList*, then the new text will receive any tags that are present on both the character before and the character after the insertion point; if a tag is present on only one of these characters then it will not be applied to the new text. If *tagList* is specified then it consists of a list of tag names; the new characters will receive all of the tags in this list and no others, regardless of the tags present around the insertion point. If multiple *chars*−*tagList* argument pairs are present, they produce the same effect as if a separate **insert** widget command had been issued for each pair, in order. The last *tagList* argument may be omitted.

$text->**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:

$text->**markGravity**(*markName, ?direction?*)

If *direction* is not specified, returns **left** or **right** to indicate which of its adjacent characters *markName* is attached to. If *direction* is specified, it must be **left** or **right**; the gravity of *markName* is set to the given value.

$text->**markNames**

Returns a list whose elements are the names of all the marks that are currently set.

$text->**markNext**(*index*)

Returns the name of the next mark at or after *index*. If *index* is specified in numerical form, then the search for the next mark begins at that index. If *index* is the name of a mark, then

the search for the next mark begins immediately after that mark. This can still return a mark at the same position if there are multiple marks at the same index. These semantics mean that the **mark next** operation can be used to step through all the marks in a text widget in the same order as the mark information returned by the **dump** operation. If a mark has been set to the special **end** index, then it appears to be *after* **end** with respect to the **mark next** operation. An empty string is returned if there are no marks after *index*.

$text->**markPrevious***(index)*

Returns the name of the mark at or before *index*. If *index* is specified in numerical form, then the search for the previous mark begins with the character just before that index. If *index* is the name of a mark, then the search for the next mark begins immediately before that mark. This can still return a mark at the same position if there are multiple marks at the same index. These semantics mean that the **mark previous** operation can be used to step through all the marks in a text widget in the reverse order as the mark information returned by the **dump** operation. An empty string is returned if there are no marks before *index*.

$text->**markSet***(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.

$text->**markUnset***(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 ''*$text*->**markNames**''. This command returns an empty string.

$text->**scan***(option, args*) or
$text->**scan***option(args)*

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

$text->**scanMark***(x, y*)

Records *x* and *y* and the current view in the text window, for use in conjunction with later **scanDragto** method. Typically this method is associated with a mouse button press in the widget. It returns an empty string.

$text->**scanDragto***(x, y*)

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

$text->**search***(?switches,? pattern, index,* ?*stopIndex*?)

Searches the text in *$text* starting at *index* for a range of characters that matches *pattern*. If a match is found, the index of the first character in the match is returned as result; otherwise an empty string is returned. One or more of the following switches (or abbreviations thereof) may be specified to control the search:

–forwards

The search will proceed forward through the text, finding the first matching range starting at or after the position given by *index*. This is the default.

–backwards

The search will proceed backward through the text, finding the matching range closest to *index* whose first character is before *index*.

–exact Use exact matching: the characters in the matching range must be identical to those in *pattern*. This is the default.

–regexp Treat *pattern* as a regular expression and match it against the text using the rules for regular expressions (see the **regexp** command for details).

–nocase Ignore case differences between the pattern and the text.

–count *varName*

The argument following **–count** gives the name of a variable; if a match is found, the number of characters in the matching range will be stored in the variable.

–hidden Find hidden text as well. By default only displayed text is found.

— This switch has no effect except to terminate the list of switches: the next argument will be treated as *pattern* even if it starts with **–**.

The matching range must be entirely within a single line of text. For regular expression matching the newlines are removed from the ends of the lines before matching: use the **$** feature in regular expressions to match the end of a line. For exact matching the newlines are retained. If *stopIndex* is specified, the search stops at that index: for forward searches, no match at or after *stopIndex* will be considered; for backward searches, no match earlier in the text than *stopIndex* will be considered. If *stopIndex* is omitted, the entire text will be searched: when the beginning or end of the text is reached, the search continues at the other end until the starting location is reached again; if *stopIndex* is specified, no wrap–around will occur.

`$text->`**see***(index)*

Adjusts the view in the window so that the character given by *index* is completely visible. If *index* is already visible then the command does nothing. If *index* is a short distance out of view, the command adjusts the view just enough to make *index* visible at the edge of the window. If *index* is far out of view, then the command centers *index* in the window.

`$text->`**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:

`$text->`**tagAdd***(tagName, index1, ?index2, index1, index2, ...?)*

Associate the tag *tagName* with all of the characters starting with *index1* and ending just before *index2* (the character at *index2* isn't tagged). A single command may contain any number of *index1–index2* pairs. If the last *index2* is omitted then the single character at *index1* is tagged. If there are no characters in the specified range (e.g. *index1* is past the end of the file or *index2* is less than or equal to *index1*) then the command has no effect.

`$text->`**tagBind***(tagName, ?sequence?, ?script?)*

This command associates *script* with the tag given by *tagName*. Whenever the event sequence given by *sequence* occurs for a character that has been tagged with *tagName*, the script will be invoked. This method is similar to the **bind** command except that it operates on characters in a text rather than entire widgets. See the *Tk::bind* documentation for complete details on the syntax of *sequence* and the substitutions performed on *script* before invoking it. If all arguments are specified then a new binding is created, replacing any existing binding for the same *sequence* and *tagName* (if the first character of *script* is ''+'' then *script* augments an existing binding rather than replacing it). In this case the return value is an empty string. If *script* is omitted then the command returns the *script* associated with *tagName* and *sequence* (an error occurs if there is no such binding). If both *script* and *sequence* are omitted then the command returns a list of all the sequences for which bindings have been defined for *tagName*.

The only events for which bindings may be specified are those related to the mouse and keyboard (such as **Enter**, **Leave**, **ButtonPress**, **Motion**, and **KeyPress**) or virtual events.

Event bindings for a text widget use the **current** mark described under *"MARKS"* above. An **Enter** event triggers for a tag when the tag first becomes present on the current character, and a **Leave** event triggers for a tag when it ceases to be present on the current character. **Enter** and **Leave** events can happen either because the **current** mark moved or because the character at that position changed. Note that these events are different than **Enter** and **Leave** events for windows. Mouse and keyboard events are directed to the current character. If a virtual event is used in a binding, that binding can trigger only if the virtual event is defined by an underlying mouse–related or keyboard–related event.

It is possible for the current character to have multiple tags, and for each of them to have a binding for a particular event sequence. When this occurs, one binding is invoked for each tag, in order from lowest–priority to highest priority. If there are multiple matching bindings for a single tag, then the most specific binding is chosen (see the the documentation for the **bind** command for details). **continue** and **break** commands within binding scripts are processed in the same way as for bindings created with the **bind** command.

If bindings are created for the widget as a whole using the **bind** command, then those bindings will supplement the tag bindings. The tag bindings will be invoked first, followed by bindings for the window as a whole.

`$text->`**tagCget***(tagName, option)*

This command returns the current value of the option named *option* associated with the tag given by *tagName*. *Option* may have any of the values accepted by the **tag configure** method.

`$text->`**tagConfigure***(tagName, ?option?, ?value?, ?option, value, ...?)*

This command is similar to the **configure** method 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::options* 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.

`$text->`**tagDelete***(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.

`$text->`**tagLower***(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.

`$text->`**tagNames***(?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 ''`$text->`**tag**'' widget command but haven't been deleted by a ''`$text->`**tagDelete**'' method, even if no characters are currently marked with the tag). The list will be sorted in order from lowest priority to highest priority.

$text->**tagNextrange**(*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.

$text->**tagPrevrange**(*tagName, index1,* ?*index2*?)

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

$text->**tagRaise**(*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.

$text->**tagRanges**(*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.

$text->**tagRemove**(*tagName, index1,* ?*index2, index1, index2, ...*?)

Remove the tag *tagName* from all of the characters starting at *index1* and ending just before *index2* (the character at *index2* isn't affected). A single command may contain any number of *index1*–*index2* pairs. If the last *index2* is omitted then the single character at *index1* is tagged. If there are no characters in the specified range (e.g. *index1* is past the end of the file or *index2* is less than or equal to *index1*) then the command has no effect. This command returns an empty string.

$text–**widget**(*option?,* arg, arg, ...?)
$text–**widget***Option*(?*arg,* arg, ...?)

This method is used to manipulate embedded windows. The behavior of the method depends on the *option* argument that follows the **window** argument. The following forms of the method are currently supported:

$text–**windowCget**(*index,* option)

Returns the value of a configuration option for an embedded window. *Index* identifies the embedded window, and *option* specifies a particular configuration option, which must be one of the ones listed in *"EMBEDDED WINDOWS"* above.

$text–**windowConfigure**(*index*?, *option, value, ...*?)

Query or modify the configuration options for an embedded window. If no *option* is specified, returns a list describing all of the available options for the embedded window at *index* (see *Tk::options* for information on the format of this list). If *option* is specified with

no *value*, then the command returns a list describing the one named option (this list will be identical to the corresponding sublist of the value returned if no *option* is specified). If one or more *option−value* pairs are specified, then the command modifies the given option(s) to have the given value(s); in this case the command returns an empty string. See *"EMBEDDED WINDOWS"* above for information on the options that are supported.

$text−**windowCreate**(*index*?, *option, value, ...*?)

This command creates a new window annotation, which will appear in the text at the position given by *index*. Any number of *option−value* pairs may be specified to configure the annotation. See *"EMBEDDED WINDOWS"* above for information on the options that are supported. Returns an empty string.

$text−**windowNames**

Returns a list whose elements are the names of all windows currently embedded in `$text`.

$text−**xview**(*option,* args)

This command is used to query and change the horizontal position of the text in the widget's window. It can take any of the following forms:

$text−**xview**

Returns a list containing two elements. Each element is a real fraction between 0 and 1; together they describe the portion of the document's horizontal span that is visible in the window. For example, if the first element is .2 and the second element is .6, 20% of the text is off−screen to the left, the middle 40% is visible in the window, and 40% of the text is off−screen to the right. The fractions refer only to the lines that are actually visible in the window: if the lines in the window are all very short, so that they are entirely visible, the returned fractions will be 0 and 1, even if there are other lines in the text that are much wider than the window. These are the same values passed to scrollbars via the **−xscrollcommand** option.

$text−>**xviewMoveto***(fraction)*

Adjusts the view in the window so that *fraction* of the horizontal span of the text is off−screen to the left. *Fraction* is a fraction between 0 and 1.

$text−>**xviewScroll***(number, what*)

This command shifts the view in the window left or right according to *number* and *what*. *Number* must be an integer. *What* must be either **units** or **pages** or an abbreviation of one of these. If *what* is **units**, the view adjusts left or right by *number* average−width characters on the display; if it is **pages** then the view adjusts by *number* screenfuls. If *number* is negative then characters farther to the left become visible; if it is positive then characters farther to the right become visible.

$text−>**yview***(?args?)*

This command is used to query and change the vertical position of the text in the widget's window. It can take any of the following forms:

$text−>**yview**

Returns a list containing two elements, both of which are real fractions between 0 and 1. The first element gives the position of the first character in the top line in the window, relative to the text as a whole (0.5 means it is halfway through the text, for example). The second element gives the position of the character just after the last one in the bottom line of the window, relative to the text as a whole. These are the same values passed to scrollbars via the **−yscrollcommand** option.

$text->**yviewMoveto***(fraction)*

Adjusts the view in the window so that the character given by *fraction* appears on the top line of the window. *Fraction* is a fraction between 0 and 1; 0 indicates the first character in the text, 0.33 indicates the character one−third the way through the text, and so on.

$text->**yviewScroll***(number, what*)

This command adjust the view in the window up or down according to *number* and *what*. *Number* must be an integer. *What* must be either **units** or **pages**. If *what* is **units**, the view adjusts up or down by *number* lines on the display; if it is **pages** then the view adjusts by *number* screenfuls. If *number* is negative then earlier positions in the text become visible; if it is positive then later positions in the text become visible.

$text−**yview**(?**−pickplace**,? *index*)

Changes the view in the *$text*'*s* window to make *index* visible. If the **−pickplace** option isn't specified then *index* will appear at the top of the window. If **−pickplace** is specified then the widget chooses where *index* appears in the window:

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

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

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

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

The **−pickplace** option has been obsoleted by the **see** widget command (**see** handles both x− and y−motion to make a location visible, whereas **−pickplace** only handles motion in y).

$text->**yview***(number)*

This command makes the first character on the line after the one given by *number* visible at the top of the window. *Number* must be an integer. This command used to be used for scrolling, but now it is obsolete.

BINDINGS

Tk automatically creates class bindings for texts that give them the following default behavior. In the descriptions below, ''word'' refers to a contiguous group of letters, digits, or ''_'' characters, or any single character other than these.

[1] Clicking mouse button 1 positions the insertion cursor just before the character underneath the mouse cursor, sets the input focus to this widget, and clears any selection in the widget. Dragging with mouse button 1 strokes out a selection between the insertion cursor and the character under the mouse.

[2] Double−clicking with mouse button 1 selects the word under the mouse and positions the insertion cursor at the beginning of the word. Dragging after a double click will stroke out a selection consisting of whole words.

[3] Triple−clicking with mouse button 1 selects the line under the mouse and positions the insertion cursor at the beginning of the line. Dragging after a triple click will stroke out a selection consisting of whole lines.

[4] The ends of the selection can be adjusted by dragging with mouse button 1 while the Shift key is down; this will adjust the end of the selection that was nearest to the mouse cursor when button 1 was pressed. If the button is double−clicked before dragging then the selection will be adjusted in units of whole words; if it is triple−clicked then the selection will be adjusted in units of whole lines.

[5] Clicking mouse button 1 with the Control key down will reposition the insertion cursor without affecting the selection.

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

[7] The view in the widget can be adjusted by dragging with mouse button 2. If mouse button 2 is clicked without moving the mouse, the selection is copied into the text at the position of the mouse cursor. The Insert key also inserts the selection, but at the position of the insertion cursor.

[8] If the mouse is dragged out of the widget while button 1 is pressed, the entry will automatically scroll to make more text visible (if there is more text off−screen on the side where the mouse left the window).

[9] The Left and Right keys move the insertion cursor one character to the left or right; they also clear any selection in the text. If Left or Right is typed with the Shift key down, then the insertion cursor moves and the selection is extended to include the new character. Control−Left and Control−Right move the insertion cursor by words, and Control−Shift−Left and Control−Shift−Right move the insertion cursor by words and also extend the selection. Control−b and Control−f behave the same as Left and Right, respectively. Meta−b and Meta−f behave the same as Control−Left and Control−Right, respectively.

[10] The Up and Down keys move the insertion cursor one line up or down and clear any selection in the text. If Up or Right is typed with the Shift key down, then the insertion cursor moves and the selection is extended to include the new character. Control−Up and Control−Down move the insertion cursor by paragraphs (groups of lines separated by blank lines), and Control−Shift−Up and Control−Shift−Down move the insertion cursor by paragraphs and also extend the selection. Control−p and Control−n behave the same as Up and Down, respectively.

[11] The Next and Prior keys move the insertion cursor forward or backwards by one screenful and clear any selection in the text. If the Shift key is held down while Next or Prior is typed, then the selection is extended to include the new character. Control−v moves the view down one screenful without moving the insertion cursor or adjusting the selection.

[12] Control−Next and Control−Prior scroll the view right or left by one page without moving the insertion cursor or affecting the selection.

[13] Home and Control−a move the insertion cursor to the beginning of its line and clear any selection in the widget. Shift−Home moves the insertion cursor to the beginning of the line and also extends the selection to that point.

[14] End and Control−e move the insertion cursor to the end of the line and clear any selection in the widget. Shift−End moves the cursor to the end of the line and extends the selection to that point.

[15] Control−Home and Meta−< move the insertion cursor to the beginning of the text and clear any selection in the widget. Control−Shift−Home moves the insertion cursor to the beginning of the text and also extends the selection to that point.

[16] Control−End and Meta−> move the insertion cursor to the end of the text and clear any selection in the widget. Control−Shift−End moves the cursor to the end of the text and extends the selection to that point.

[17] The Select key and Control−Space set the selection anchor to the position of the insertion cursor. They don‘t affect the current selection. Shift−Select and Control−Shift−Space adjust the selection to the current position of the insertion cursor, selecting from the anchor to the insertion cursor if there was not any selection previously.

[18] Control−/ selects the entire contents of the widget.

[19] Control−\ clears any selection in the widget.

[20] The F16 key (labelled Copy on many Sun workstations) or Meta−w copies the selection in the widget to the clipboard, if there is a selection.

[21] The F20 key (labelled Cut on many Sun workstations) or Control−w copies the selection in the widget to the clipboard and deletes the selection. If there is no selection in the widget then these keys have no effect.

[22] The F18 key (labelled Paste on many Sun workstations) or Control−y inserts the contents of the clipboard at the position of the insertion cursor.

[23] The Delete key deletes the selection, if there is one in the widget. If there is no selection, it deletes the character to the right of the insertion cursor.

[24] Backspace and Control−h delete the selection, if there is one in the widget. If there is no selection, they delete the character to the left of the insertion cursor.

[25] Control−d deletes the character to the right of the insertion cursor.

[26] Meta−d deletes the word to the right of the insertion cursor.

[27] Control−k deletes from the insertion cursor to the end of its line; if the insertion cursor is already at the end of a line, then Control−k deletes the newline character.

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

[29] Meta−backspace and Meta−Delete delete the word to the left of the insertion cursor.

[30] Control−x deletes whatever is selected in the text widget.

[31] Control−t reverses the order of the two characters to the right of the insertion cursor.

If the widget is disabled using the **−state** option, then its view can still be adjusted and text can still be selected, but no insertion cursor will be displayed and no text modifications will take place.

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

PERFORMANCE ISSUES

Text widgets should run efficiently under a variety of conditions. The text widget uses about 2−3 bytes of main memory for each byte of text, so texts containing a megabyte or more should be practical on most workstations. Text is represented internally with a modified B−tree structure that makes operations relatively efficient even with large texts. Tags are included in the B−tree structure in a way that allows tags to span large ranges or have many disjoint smaller ranges without loss of efficiency. Marks are also implemented in a way that allows large numbers of marks. In most cases it is fine to have large numbers of unique tags, or a tag that has many distinct ranges.

One performance problem can arise if you have hundreds or thousands of different tags that all have the following characteristics: the first and last ranges of each tag are near the beginning and end of the text, respectively, or a single tag range covers most of the text widget. The cost of adding and deleting tags like this is proportional to the number of other tags with the same properties. In contrast, there is no problem with having thousands of distinct tags if their overall ranges are localized and spread uniformly throughout the text.

Very long text lines can be expensive, especially if they have many marks and tags within them.

The display line with the insert cursor is redrawn each time the cursor blinks, which causes a steady stream of graphics traffic. Set the **−insertofftime** option to 0 avoid this.

SEE ALSO

Tk::ROText|Tk::ROText Tk::TextUndo|Tk::TextUndo

KEYWORDS

text, widget

NAME

Tk::TextUndo – perl/tk text widget with bindings to undo changes.

=for pm Tk/TextUndo.pm

=for category Derived Widgets

SYNOPSIS

use Tk::TextUndo;

$testundo = *$parent*->**TextUndo**(*?–option=>value, ...?*);

DESCRIPTION

This IS–A text widget with an unlimited 'undo' history but without a re'undo' capability.

Bindings

The `TextUndo` widget has the same bindings as the *Text|Tk::Text* widget. In addition there are the following bindings:

Event <L4 <<Undo

undo the last change. Pressing <L4 several times undo step by step the changes made to the text widget.

Methods

The `TextUndo` widget has the same methods as `Text` widget. Additional methods for the `TextUndo` widget are:

`$text`–Load($filename);

Loads the contents of the `$filename` into the text widget. `Load()` delete the previous contents of the text widget as well as it's undo history of the previous file.

`$text`–Save(?$otherfilename?)

Save contents of the text widget to a file. If the `$otherfilename` is not specified, the text widget contents writes the file of `$filename` used in the last `Load()` call. If no file was previously `Load()`'ed an error message pops up. The default filename of the last `Load()` call is not overwriten by `$otherfilename`.

`$text`–FileName(?$otherfilename?)

If passed an argument sets the file name associated with the loaded document. Returns the current file name associated with the document.

KEYS

widget, text, undo

SEE ALSO

Tk::Text, *Tk::ROText*

NAME

Tk::Tiler – Scrollable frame with sub–widgets arranged into rows

=for pm Tk/Tiler.pm

=for category Tk Geometry Management

SYNOPSIS

```
use Tk::Tiler;

my $tl = $parent->Scrolled('Tiler', -columns => n, -rows => n);

my $a  = $tl->Xxxxx(...);
my $b  = $tl->Xxxxx(...);
my $c  = $tl->Xxxxx(...);

$tl->Manage($a,$b,$c);
```

DESCRIPTION

Tiler is derived form Tk::Frame. It is a geometry managing widget which accepts widgets to manage. It places the widgets in a grid with as many widgets as possible in a row. All the "slots" in the grid are the same size, which is determined by the largest managed widget.

The grid may be scrolled vertically.

NAME

Tk::TixGrid – Create and manipulate Tix Grid widgets

=for pm TixGrid/TixGrid.pm

=for category Tix Extensions

SYNOPSIS

$tixgrid = *$parent*->**TixGrid**?(*options*)?;

```
The port of C code and bindings is done but needs
debugging.  THERE ARE KNOWN BUGS. Work in progress ...
```

STANDARD OPTIONS

-background
-borderwidth
-cursor
-font
-foreground
-height
-highlightbackground
-highlightcolor
-highlightthickness
-padx
-pady
-relief
-selectbackground
-selectborderwidth
-selectforeground
-state
-takefocus
-width
-xscrollcommand
-yscrollcommand

See *Tk::options* for details of the standard options.

WIDGET-SPECIFIC OPTIONS

Name: **browseCmd**
Class: **BrowseCmd**
Switch: **-browsecmd**

?docu here? Not in configure output but used in bindings ??!!

If defined, gives a perl/Tk *callback\|Tk::callbacks* to be executed when the user browses a grid cell (This is normally the case when the user clicks on an entry). When this callback is called, it is passed with two additional parameters: *x y*, where (*x*,*y*) is the location of the cell that has just been clicked.

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

?docu here? Not in configure output but used in bindings ??!!

Name: **editDoneCmd**
Class: **EditDoneCmd**
Switch: **-editdonecmd**

If defined, gives a perl/Tk *callback\|Tk::callbacks* to be executed when the user has edited grid cell. When this callback is called, it is passed with two additional parameters: *x y*, where (*x*,*y*) is the location

of the cell that has just been edited.

Name: **editNotifyCmd**
Class: **EditNotifyCmd**
Switch: **–editnotifycmd**

If defined gives a perl/Tk *callback\Tk::callbacks* to be executed when the user tries to edit a grid cell. When this callback is called, it is passed with two additional parameters: *x y*, where (*x*,*y*,) is the location of the cell. This callback should return a boolean value: **true** indicates that the cell is editable and **false** otherwise.

Name: **FloatingCols**
Class: **floatingCols**
Switch: **–floatingcols**

Defines the number of columns that fixed when the widget is horizontally scrolled. These column(s) can be used as label(s) for the column(s). The floating column(s) can be configured in the **–formatcmd** callback with the **formatBorder** method. The default value is 0.

Name: **FloatingRows**
Class: **floatingRows**
Switch: **–floatingrows**

Defines the number of rows that are fixed when the widget is vertically scrolled. These row(s) can be used as label(s) for the row(s). The floating row(s) can be configured in the **–formatcmd** callback with the **formatBorder** method. The default value is 0.

Name: **formatCmd**
Class: **FormatCmd**
Switch: **–formatcmd**

If defined, gives a perl/Tk *callback\Tk::callbacks* to be executed when the grid cells need to be formatted on the screen. Normally, this callback calls the **format** method (see below). When this callback is called, it is passed with five additional parameters: *type x1 y1 x2 y2*. *type* gives the logical type of the region in the grid. It may be one of the following.

x–region the horizontal margin

y–region the vertical margin

s–region the area where the horizontal and vertical margins are joined

main all the cells that do not fall into the above three types

x1 y1 x2 y2 gives the extent of the region that needs formatting.

Name: **leftMargin**
Class: **LeftMargin**
Switch: **–leftmargin**

In the number of cells, gives the width of vertical margin. A zero indicates that no vertical should be drawn.

Name: **itemType**
Class: **ItemType**
Switch: **–itemtype**

?docu here?

Name: **selectMode**
Class: **SelectMode**
Switch: **–selectmode**

Specifies one of several styles for manipulating the selection. The value of the option may be arbitrary, but the default bindings expect it to be either **single**, **browse**, **multiple**, or **extended**; the default value is **single**.

Name: **selectUnit**
Class: **SelectUnit**
Switch: **–selectunit**

Specifies the selection unit. Valid values are **cell**, **column** or **row**.

Name: **sizeCmd**
Class: **SizeCmd**
Switch: **–sizecmd**

?docu here?

Name: **topMargin**
Class: **TopMargin**
Switch: **–topmargin**

In the number of cells, gives the height of horizontal margin. A zero indicates that no horizontal should be drawn.

DESCRIPTION

The **TixGrid** method creates a TixGrid new window and returns a blessed reference of this TixGrid widget. Additional options, described above, may be specified on the command line or in the option database to configure aspects of the **TixGrid** widget such as its cursor and relief.

A Grid widget displays its contents in a two dimensional grid of cells. Each cell may contain one Tix display item, which may be in text, graphics or other formats. See *Tk::DItem* for more information about Tix display items. Individual cells, or groups of cells, can be formatted with a wide range of attributes, such as its color, relief and border.

WIDGET METHODS

The **TixGrid** method creates a TixGrid widget and returns a blessed reference of this TixGrid widget. This reference may be used to invoke various operations on the widget. It has the following general form:

$tixgrid–**method**?(*arg, arg, ...)?*

*arg*s determine the exact behavior of the method. The following methods are possible for **TixGrid** widgets:

$tixgrid–**anchor**(*action*, *x*, *y*)
$tixgrid–**anchor***Action*(*x*, *y*)

Manipulates the **anchor cell** of the **TixGrid** widget. The anchor cell is the end of the selection that is fixed while the user is dragging out a selection with the mouse. *Action* can be **clear**, **get** or **set**. If *action* is **clear**, *x* and *y* args are not accepted.

$tixgrid–**bdtype**(*x*, *y* ?,*xbdWidth*, *ybdWidth*?)

????

$tixgrid–**cget**('–*option*')

Returns the current value of the configuration option given by –*option*. –*option* may have any of the values accepted by the **TixGrid** constructor method.

$tixgrid–**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 *$tixgrid* (see **Tk_ConfigureInfo** for information on the format of this list.) If –*option* is specified with no *value*, then the method 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 method modifies the given widget option(s) to have the given value(s); in this case the method returns an empty string. –*option* may have any of the values accepted by the **TixGrid** constructor method.

$tixgrid–**delete(***dim***,** *from***?,** *to***?)**
$tixgrid–**deleteColumn**(*from*?, *to*?)
$tixgrid–**deleteRow**(*from*?, *to*?)

Dim may be **row** or **column**. If *to* is not given, deletes a single row (or column) at the position *from*. If *to* is given, deletes the range of rows (or columns) from position *from* through *to*.

$tixgrid–**dragsite**(*option*, *x*, *y*)

?docu here? not implemented :–(

$tixgrid–**dropsite**(*option*, *x*, *y*)

?docu here? not implemented :–(

$tixgrid–**editApply**

If any cell is being edited, de–highlight the cell and applies the changes.

$tixgrid–**editSet**(*x*, *y*)

Highlights the cell at (*x*,*y*) for editing, if the **–editnotify** callback returns true for this cell.

$tixgrid–**entrycget**(*x*, *y*, '*–option*')

Returns the current value of the configuration option given by *–option* of the cell at (*x*,*y*). *–option* may have any of the values accepted by the **set** method.

$tixgrid–**entryconfigure**(*x*, *y*?, *–option*??=*value*, *–option*=*value*, ...?)

Query or modify the configuration options of the cell at (*x*,*y*). If no *–option* is specified, returns a list describing all of the available options for the cell (see **Tk_ConfigureInfo** for information on the format of this list.) If *–option* is specified with no *value*, then the method 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 method modifies the given widget option(s) to have the given value(s); in this case the method returns an empty string. *Option* may have any of the values accepted by the **set** method.

$tixgrid–**format**(*option*, ?*args*, ...?)
$tixgrid–**formatBorder**(*x1,y1*, *x2,y2*, *options*);
$tixgrid–**formatGrid**(*x1,y1*, *x2,y2*, *options*);

the **format** method can only be called by the **–formatcmd** callback of the tixGrid widget.

?docu complete?

$tixgrid–**geometryinfo**(?*width*, ?*height*, ...?)

?docu here? Return a list of 4 floats! Currently "{first1 last1} {first2,last2}" :–(

$tixgrid*–**index**(*$coordx, *$coordy)*

?docu here?

retuns *(nx, ny)* of entry at position (*$coordx, $coordy).* (??widget or screen offset??)

$tixgrid–**info**(*option*, ?*args*, ...?)

?docu here?

$tixgrid–**move**(*dim*, *from*, *to*, *offset*)
$tixgrid–**moveColumn**(*from*, *to*, *offset*)
$tixgrid–**moveRow**(*from*, *to*, *offset*)

Dim may be **row** or **column**. Moves the range of rows (or columns) from position *from* through *to* by the distance indicated by *offset*. For example, *$tixgrid*–**moveRow**(**2**, **4**, **1**) moves the rows 2,3,4 to rows 3,4,5.

$tixgrid–**nearest(***x, y***)**

?docu here? screen pos (pixels) to entry (nx,ny) translation.

$tixgrid–**selection**(*option*, *x1*, *y1* ?,*x2*, *y2*?)
$tixgrid–**selection***Option*(*x1*, *y1* ?,*x2*, *y2*?)

Option one of: **adjust**, **clear**, **includes**, **set**, and **toggle**.

x1 (y1) has not to be greater than x2 (y2), but only x2 and y2 can be 'max'.

BUG: *selection includes*: has no visible effect (as in Tix). Eh???

BUG: *selection clear*: only works for 0, 0, max, max (as in Tix). Eh???

When *x2, y2* are not given they default to *x1, y1*, respectively.

$tixgrid–**selectionAdjust**(*x1*, *y1* ?,*x2*, *y2*?)

?docu here?

$tixgrid–**selectionClear**(*x1*, *y1* ?,*x2*, *y2*?)

?docu here?

$tixgrid–**selectionIncludes**(*x1*, *y1* ?,*x2*, *y2*?)

?docu here?

$tixgrid–**selectionSet**(*x1*, *y1* ?,*x2*, *y2*?)

?docu here?

$tixgrid–**selectionToggle**(*x1*, *y1* ?,*x2*, *y2*?)

?docu here?

$tixgrid–**set**(*x*, *y*?, **–itemtype**=*type*??, *–option*=*value*, *...*?)

Creates a new display item at the cell at (*x*,*y*). The optional **–itemtype** parameter gives the type of the display item. An additional list of *option–value* pairs specify options of the display item. If a display item already exists at this cell, the old item will be deleted automatically.

$tixgrid–**size**(*dim*, *index*?, *–option*??=*value*, *...*?)
$tixgrid–**sizeColumn**(*index*?, *–option*??=*value*, *...*?)
$tixgrid–**sizeRow**(*index*?, *–option*??=*value*, *...*?)

Queries or sets the size of the row or column given by *dim* and *index*. *Dim* may be **row** or **column**. *Index* may be any non–negative integer that gives the position of a given row (or column). *Index* can also be the string **default**; in this case, this method queries or sets the default size of all rows (or columns). When no *option–value* pair is given, this method returns a list containing the current size setting of the given row (or column). When *option–value* pairs are given, the corresponding options of the size setting of the given row are changed. *–option* may be one of the following:

–pad0 = *pixels*

Specifies the paddings to the left of a column or the top of a row.

–pad1 = *pixels*

Specifies the paddings to the right of a column or the bottom of a row.

–size = *val*

Specifies the width of a column or the height of a row. *Val* may be: **auto** — the width of the column is set the widest cell in the column; a valid Tk screen distance unit (see **Tk_GetPixels**); or a real number following by the word **chars** (e.g. **3.4chars**) that sets the width of the column to the given number of characters.

$tixgrid–**sort(***dimension***,** *start***,** *end***, ?***args ...***?)**

?docu here? (not supported on Win* OSs up to now)

$tixgrid–**unset**(*x*, *y*)

Clears the cell at (*x*,*y*) by removing its display item.

$tixgrid–**xview**

?docu here?

$tixgrid–**yview**

?docu here?

BINDINGS

to be done.

SEE ALSO

Tk::DItem|Tk::DItem Tk::callbacks|Tk::callbacks Tk::FloatEntry|Tk::FloatEntry

BUGS

C code and bindings of TixGrid have some bugs.

KEYWORDS

tix, tixgrid, table, display item, spreadsheet

NAME

Tk::tixWm – Tix's addition to the standard TK wm command.

=for category Tix Extensions

SYNOPSIS

$widget->**wmCapture**

$widget->**wmRelease**

DESCRIPTION

The **wmCapture** and the **wmRelease** methods change the toplevel attribute of Tk widgets.

METHODS

$widget->**wmCapture**

Converts the toplevel window specified by *$widget* into a non–toplevel widget. Normally this command is called to convert a *Toplevel\Tk::Toplevel* widget into a *Frame\Tk::Frame* widget. The newly–converted frame widget is un–mapped from the screen. To make it appear inside its parent, you must call a geometry manager (e.g. grid or pack) explictly.

$widget->**wmRelease**

Makes the non–toplevel window specified by *$widget* into a toplevel widget. Normally this command is called to convert a *Frame\Tk::Frame* widget into a *Toplevel\Tk::Toplevel* widget, but it can also be used on any non–toplevel widget (e.g, label). The newly–converted toplevel window is in a **withdrawn** state. To make it appear on the screen, you must call **deiconify** after calling **wmRelease**.

Any data associated with *$widget* via **wm** methods (icon, protocol, command etc.) are released, and must be re–established if window is later re–captured.

BUGS

wmCapture does not exist in the Win32 window manager code.

How these methods interact with perl/Tk's class hierarchy is not yet clear. In particular a **wmRelease**d window will not automatically ''*inherit*'' the **Tk::Wm** methods, however a **wmCapture**d window still will. (A **release**d **Label** might make a good candidate for an **Icon**.)

AUTHORS

Ioi Kim Lam – ioi@graphics.cis.upenn.edu wrote original Tix version. Updated for tk8.0, Win32 and perl by Nick Ing–Simmons.

SEE ALSO

Tk::Wm\Tk::Wm Tk::Mwm\Tk::Mwm Tk::Frame\Tk::Frame Tk::Toplevel\Tk::Toplevel

KEYWORDS

window manager, wm, TIX

NAME

tkvars – Variables used or set by Tk

=for category Tk Generic Methods

DESCRIPTION

The following perl variables are either set or used by Tk at various times in its execution. (For a list of variables used by perl see *perlvar*.)

$Tk::library

This variable holds the file name for a directory containing the modules related to Tk. These modules include an initialization file that is normally processed whenever a Tk application starts up, plus other files containing procedures that implement default behaviors for widgets. The initial value of **$Tk::library** is set when Tk is added to an interpreter; this is done by searching searching for a directory named Tk in the directory where the file ***Tk.pm***, or the first directory ***Tk*** in `@INC`.

The **TK_LIBRARY** environment variable used by Tcl/Tk is not supported by perl/Tk. Please use *@INC* to change where modules are searched.

Note: This is Tcl remnant. With perl it makes more sense to use `@INC` and `%INC`).

$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.

Note: this is Tcl remnant. With perl it makes more sense to use **$Tk::VERSION** described below.

$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

The variable holds the current version number of the perl/Tk release in the form *major.minor*. *Major* and *minor* are integers.

The *major* version number shows on which Tcl/Tk release perl/Tk is based. E.g., **402** means based on Tcls Tk 4.2. (Patchlevel of Tcls Tk are not incorporated because perl/Tk tended to be ‘‘*ahead*‘‘ of them on some fixes and behind on others. The first digest of the major version number increases in any Tk release that includes changes that are not backward compatible (i.e. whenever existing perl/Tk applications and scripts may have to change to work with the new release).

The *minor* version depends on perl/Tk only. It uses the ‘even‘=‘stable‘, ‘odd‘=‘experimental’ scheme that linux uses:

```
.0xx  - inherently 'alpha'
.1xx  - experimental 'beta'
.2xx  - stable
.3xx  - experimental
.4xx  - stable
...
```

The minor version number increases with each new release of Tk, except that it resets to zero whenever the major version number changes.

$Tk::version

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.

Note: this is Tcl remnant. With perl it makes more sense to use **$Tk::VERSION** described above.

KEYWORDS

variables, version

NAME

Tk::TList − Create and manipulate Tix Tabular List widgets

=for category Tix Extensions

SYNOPSIS

$tlist = $parent−>**TList***(?options?);*

SUPER−CLASS

None.

STANDARD OPTIONS

−background **−borderwidth** **−class** **−cursor** **−foreground** **−font** **−height** **−highlightcolor** **−highlightthickness** **−relief** **−selectbackground** **−selectforeground** **−xscrollcommand** **−yscrollcommand** **−width**

See *Tk::options* for details of the standard options.

WIDGET−SPECIFIC OPTIONS

Name: **browsecmd**
Class: **BrowseCmd**
Switch: **−browsecmd**

Specifies a perl/Tk **callback** to be executed when the user browses through the entries in the TList widget.

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

Specifies the perl/Tk **callback** to be executed when the user invokes a list entry in the TList widget. Normally the user invokes a list entry by double−clicking it or pressing the Return key.

Name: **foreground**
Class: **Foreground**
Switch: **−foreground**
Alias: **−fg**

Specifies the default foreground color for the list entries.

Name: **height**
Class: **Height**
Switch: **−height**

Specifies the desired height for the window in number of characters.

Name: **itemType**
Class: **ItemType**
Switch: **−itemtype**

Specifies the default type of display item for this TList widget. When you call the **insert** methods, display items of this type will be created if the **−itemtype** option is not specified.

Name: **orient**
Class: **Orient**
Switch: **−orient**

Specifies the order of tabularizing the list entries. When set to "**vertical**", the entries are arranged in a column, from top to bottom. If the entries cannot be contained in one column, the remaining entries will go to the next column, and so on. When set to "**horizontal**", the entries are arranged in a row, from left to right. If the entries cannot be contained in one row, the remaining entries will go to the next row, and so on.

Name: **padX**
Class: **Pad**
Switch: **–padx**

The default horizontal padding for list entries.

Name: **padY**
Class: **Pad**
Switch: **–padx**

The default vertical padding for list entries.

Name: **selectBackground**
Class: **SelectBackground**
Switch: **–selectbackground**

Specifies the background color for the selected list entries.

Name: **selectBorderWidth**
Class: **BorderWidth**
Switch: **–selectborderwidth**

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**.

Name: **selectForeground**
Class: **SelectForeground**
Switch: **–selectforeground**

Specifies the foreground color for the selected list entries.

Name: **selectMode**
Class: **SelectMode**
Switch: **–selectmode**

Specifies one of several styles for manipulating the selection. The value of the option may be arbitrary, but the default bindings expect it to be either **single**, **browse**, **multiple**, or **extended**; the default value is **single**.

Name: **sizeCmd**
Class: **SizeCmd**
Switch: **–sizecmd**

Specifies a perl/Tk **callback** to be called whenever the TList widget changes its size. This command can be useful to implement "user scroll bars when needed" features.

Name: **state**
Class: **State**
Switch: **–state**

Specifies whether the TList command should react to user actions. When set to "**normal**", the TList reacts to user actions in the normal way. When set to "**disabled**", the TList can only be scrolled, but its entries cannot be selected or activated.

Name: **width**
Class: **Width**
Switch: **–width**

Specifies the desired width for the window in characters.

DESCRIPTION

The **TList** method creates a new window (given by the `$widget` argument) and makes it into a TList widget. Additional options, described above, may be specified on the command line or in the option database to configure aspects of the TList widget such as its cursor and relief.

The TList widget can be used to display data in a tabular format. The list entries of a TList widget are similar

to the entries in the Tk listbox widget. The main differences are (1) the TList widget can display the list entries in a two dimensional format and (2) you can use graphical images as well as multiple colors and fonts for the list entries.

Each list entry is identified by an **index**, which can be in the following forms:

number

An integer that indicates the position of the entry in the list. 0 means the first position, 1 means the second position, and so on.

end Indicates the end of the listbox. For some commands this means just after the last entry; for other commands it means the last entry.

@x,y

Indicates the element that covers the point in the listbox window specified by x and y (in pixel coordinates). If no element covers that point, then the closest element to that point is used.

DISPLAY ITEMS

Each list entry in an TList widget is associated with a **display** item. The display item determines what visual information should be displayed for this list entry. Please see *Tk::DItem* for a list of all display items.

When a list entry is created by the **insert** command, the type of its display item is determined by the **–itemtype** option passed to these commands. If the **–itemtype** is omitted, then by default the type specified by this TList widget's **–itemtype** option is used.

WIDGET METHODS

The **TList** method creates a widget object.

This object supports the **configure** and **cget** methods described in *Tk::options* which can be used to enquire and modify the options described above. The widget also inherits all the methods provided by the generic *Tk::Widget|Tk::Widget* class.

The following additional methods are available for TList widgets:

`$tlist->`**anchorSet***(index)*

Sets the anchor to the list entry identified by *index*. The anchor is the end of the selection that is fixed while dragging out a selection with the mouse.

`$tlist->`**anchorClear**

Removes the anchor, if any, from this TList widget. This only removes the surrounding highlights of the anchor entry and does not affect its selection status.

`$tlist->`**delete***(from, ?to?)*

Deletes one or more list entries between the two entries specified by the indices *from* and *to*. If *to* is not specified, deletes the single entry specified by *from*.

`$tlist->`**dragsiteSet***(index)*

Sets the dragsite to the list entry identified by *index*. The dragsite is used to indicate the source of a drag–and–drop action. Currently drag–and–drop functionality has not been implemented in Tix yet.

`$tlist->`**dragsiteClear**

Remove the dragsite, if any, from the this TList widget. This only removes the surrounding highlights of the dragsite entry and does not affect its selection status.

`$tlist->`**dropsiteSet***(index)*

Sets the dropsite to the list entry identified by *index*. The dropsite is used to indicate the target of a drag–and–drop action. Currently drag–and–drop functionality has not been implemented in Tix yet.

$tlist->**dropsiteClear**

Remove the dropsite, if any, from the this TList widget. This only removes the surrounding highlights of the dropsite entry and does not affect its selection status.

$tlist->**entrycget**(*index, option*)

Returns the current value of the configuration option given by *option* for the entry indentfied by *index*. *Option* may have any of the values accepted by the **insert** method.

$tlist->**entryconfigure**(*index, ?option?, ?value, option, value, ...?*)

Query or modify the configuration options of the list entry identified by *index*. If no *option* is specified, returns a list describing all of the available options for *index* (see **Tk_ConfigureInfo** for information on the format of this list). If *option* is specified with no *value*, then the method returns a list describing the one named option (this list will be identical to the corresponding sublist of the value returned if no *option* is specified). If one or more *option*–*value* pairs are specified, then the command modifies the given option(s) to have the given value(s); in this case the method returns an empty string. *Option* may have any of the values accepted by the **insert** method. The exact set of options depends on the value of the **–itemtype** option passed to the the **insert** method when this list entry is created.

$tlist->**insert**(*index, ?option, value, ...?*)

Creates a new list entry at the position indicated by *index*. The following configuration options can be given to configure the list entry:

–itemtype = *type*

Specifies the type of display item to be display for the new list entry. *type* must be a valid display item type. Currently the available display item types are **image**, **imagetext**, **text**, and *$widget*. If this option is not specified, then by default the type specified by this TList widget's **–itemtype** option is used.

–state = *state*

Specifies whether this entry can be selected or invoked by the user. Must be either **normal** or **disabled**.

–data = *data*

Arbitrary data to be associated with the entry (a perl scalar value).

The **insert** method accepts additional configuration options to configure the display item associated with this list entry. The set of additional configuration options depends on the type of the display item given by the **–itemtype** option. Please see *Tk::DItem* for a list of the configuration options for each of the display item types.

$tlist->**info**(*option, arg, ...*)

Query information about the TList widget. *option* can be one of the following:

$tlist->**info**(**anchor,** *index*)

Returns the index of the current anchor, if any, of the TList widget. If the anchor is not set, returns the empty string.

$tlist->**info**(**dragsite,** *index*)

Returns the index of the current dragsite, if any, of the TList widget. If the dragsite is not set, returns the empty string.

$tlist->**info**(**dropsite,** *index*)

Returns the index of the current dropsite, if any, of the TList widget. If the dropsite is not set, returns the empty string.

$tlist->**info**(**selection**)

Returns a list of selected elements in the TList widget. If no entries are selected, returns an empty string.

$tlist->**nearest***(x, y)*

Given an *(x,y)* coordinate within the TList window, this command returns the index of the TList element nearest to that coordinate.

$tlist->**see***(index)*

Adjust the view in the TList so that the entry given by *index* is visible. If the entry is already visible then the command has no effect; otherwise TList scrolls to bring the element into view at the edge to which it is nearest.

$tlist->**selection***(option, arg, ...)*

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

$tlist->**selectionClear***(?from?, ?to?)*

When no extra arguments are given, deselects all of the list entrie(s) in this TList widget. When only *from* is given, only the list entry identified by *from* is deselected. When both *from* and *to* are given, deselects all of the list entrie(s) between between *from* and *to*, inclusive, without affecting the selection state of entries outside that range.

$tlist->**selectionIncludes***(index)*

Returns 1 if the list entry indicated by *index* is currently selected; returns 0 otherwise.

$tlist->**selectionSet***(from, ?to?)*

Selects all of the list entrie(s) between between *from* and *to*, inclusive, without affecting the selection state of entries outside that range. When only *from* is given, only the list entry identified by *from* is selected.

$tlist->**xview***(args)*

This command is used to query and change the horizontal position of the information in the widget's window. It can take any of the following forms:

$tlist->**xview**

Returns a list containing two elements. Each element is a real fraction between 0 and 1; together they describe the horizontal span that is visible in the window. For example, if the first element is 0.2 and the second element is 0.6, 20% of the TList entry is off−screen to the left, the middle 40% is visible in the window, and 40% of the entry is off−screen to the right. These are the same values passed to scrollbars via the **−xscrollcommand** option.

$tlist->**xview***(index)*

Adjusts the view in the window so that the list entry identified by *index* is aligned to the left edge of the window.

$tlist->**xviewMoveto***(fraction)*

Adjusts the view in the window so that *fraction* of the total width of the TList is off−screen to the left. *fraction* must be a fraction between 0 and 1.

$tlist->**xviewScroll***(number, what)*

This command shifts the view in the window left or right according to *number* and *what*. *Number* must be an integer. *What* must be either **units** or **pages** or an abbreviation of one of these. If *what* is **units**, the view adjusts left or right by *number* character units (the width of the character) on the display; if it is **pages** then the view adjusts by *number* screenfuls. If *number* is negative then characters farther to the left become visible; if it is positive then characters farther to the right become visible.

$tlist->**yview***(?args?)*

This command is used to query and change the vertical position of the entries in the widget's window. It can take any of the following forms:

$tlist->**yview**

Returns a list containing two elements, both of which are real fractions between 0 and 1. The first element gives the position of the list element at the top of the window, relative to the TList as a whole (0.5 means it is halfway through the TList, for example). The second element gives the position of the list entry just after the last one in the window, relative to the TList as a whole. These are the same values passed to scrollbars via the **−yscrollcommand** option.

$tlist->**yview***(index)*

Adjusts the view in the window so that the list entry given by *index* is displayed at the top of the window.

$tlist->**yviewMoveto***(fraction)*

Adjusts the view in the window so that the list entry given by *fraction* appears at the top of the window. *Fraction* is a fraction between 0 and 1; 0 indicates the first entry in the TList, 0.33 indicates the entry one−third the way through the TList, and so on.

$tlist->**yviewScroll***(number, what*)

This command adjust the view in the window up or down according to *number* and *what*. *Number* must be an integer. *What* must be either **units** or **pages**. If *what* is **units**, the view adjusts up or down by *number* lines; if it is **pages** then the view adjusts by *number* screenfuls. If *number* is negative then earlier entries become visible; if it is positive then later entries become visible.

BINDINGS

[1] If the **−selectmode** is "browse", when the user drags the mouse pointer over the list entries, the entry under the pointer will be highlighted and the **−browsecmd** procedure will be called with one parameter, the index of the highlighted entry. Only one entry can be highlighted at a time. The **−command** procedure will be called when the user double−clicks on a list entry.

[2] If the **−selectmode** is "single", the entries will only be highlighted by mouse <ButtonRelease−1> events. When a new list entry is highlighted, the **−browsecmd** procedure will be called with one parameter indicating the highlighted list entry. The **−command** procedure will be called when the user double−clicks on a list entry.

[3] If the **−selectmode** is "multiple", when the user drags the mouse pointer over the list entries, all the entries under the pointer will be highlighted. However, only a contiguous region of list entries can be selected. When the highlighted area is changed, the **−browsecmd** procedure will be called with an undefined parameter. It is the responsibility of the **−browsecmd** procedure to find out the exact highlighted selection in the TList. The **−command** procedure will be called when the user double−clicks on a list entry.

[4] If the **−selectmode** is "extended", when the user drags the mouse pointer over the list entries, all the entries under the pointer will be highlighted. The user can also make disjointed selections using <Control−ButtonPress−1>. When the highlighted area is changed, the **−browsecmd** procedure will be called with an undefined parameter. It is the responsibility of the **−browsecmd** procedure to find out the exact highlighted selection in the TList. The **−command** procedure will be called when the user double−clicks on a list entry.

EXAMPLE

This example demonstrates how to use an TList to store a list of numbers:

```
use strict;
use Tk ();
use Tk::TList;

my $mw = Tk::MainWindow->new();
my $image = $mw->Getimage('folder');
```

```
my $tlist = $mw->TList(-orient => 'vertical');
for my $text ( qw/one two three four five six seven eight nine/ ) {
    $tlist->insert('end',
              -itemtype=>'imagetext', -image=>$image, -text=>$text);
}

$tlist->pack(-expand=>'yes', -fill=>'both');

Tk::MainLoop;
```

SEE ALSO

Tk::options|Tk::options Tk::Widget|Tk::Widget Tk::DItem|Tk::DItem Tk::HList|Tk::HList Tk::TixGrid|Tk::TixGrid

KEYWORDS

Tix(n), Tabular Listbox, Display Items

NAME

Tk::Toplevel – Create and manipulate Toplevel widgets

=for category Tk Widget Classes

SYNOPSIS

$toplevel = $parent->**Toplevel***(?options?);*

STANDARD OPTIONS

–borderwidth –highlightbackground–highlightthickness –takefocus –class –highlightcolor –relief –cursor

See *Tk::options* for details of the standard options.

WIDGET–SPECIFIC OPTIONS

Name: **background**
Class: **Background**
Switch: **–background**

This option is the same as the standard **background** option except that its value may also be specified as an undefined value. In this case, the widget will display no background or border, and no colors will be consumed from its colormap for its background and border.

Name: **colormap**
Class: **Colormap**
Switch: **–colormap**

Specifies a colormap to use for the window. The value may be either **new**, in which case a new colormap is created for the window and its children, or the name of another window (which must be on the same screen and have the same visual as `$widget)`, in which case the new window will use the colormap from the specified window. If the **colormap** option is not specified, the new window uses the default colormap of its screen. This option may not be changed with the **configure** method.

Name: **container**
Class: **Container**
Switch: **–container**

The value must be a boolean. If true, it means that this window will be used as a container in which some other application will be embedded (for example, a Tk toplevel can be embedded using the **–use** option). The window will support the appropriate window manager protocols for things like geometry requests. The window should not have any children of its own in this application. This option may not be changed with the **configure** method.

Name: **height**
Class: **Height**
Switch: **–height**

Specifies the desired height for the window in any of the forms acceptable to **Tk_GetPixels**. If this option is less than or equal to zero then the window will not request any size at all.

Name: **menu**
Class: **Menu**
Switch: **–menu**

Specifies a menu widget to be used as a menubar. On the Macintosh, the menubar will be displayed accross the top of the main monitor. On Microsoft Windows and all UNIX platforms, the menu will appear accross the toplevel window as part of the window dressing maintained by the window manager.

Name: **""**

Class: **""**
Switch: **–screen**

Specifies the screen on which to place the new window. Any valid screen name may be used, even one associated with a different display. Defaults to the same screen as its parent. This option is special in that it may not be specified via the option database, and it may not be modified with the **configure** method.

Name: **use**
Class: **Use**
Switch: **–use**

This option is used for embedding. If the value isn't an empty string, it must be the the window identifier of a container window, specified as a hexadecimal string like the ones returned by the **winfo id** command. The toplevel widget will be created as a child of the given container instead of the root window for the screen. If the container window is in a Tk application, it must be a frame or toplevel widget for which the **–container** option was specified. This option may not be changed with the **configure** method.

Name: **visual**
Class: **Visual**
Switch: **–visual**

Specifies visual information for the new window in any of the forms accepted by **Tk_GetVisual**. If this option is not specified, the new window will use the default visual for its screen. The **visual** option may not be modified with the **configure** method.

Name: **width**
Class: **Width**
Switch: **–width**

Specifies the desired width for the window in any of the forms acceptable to **Tk_GetPixels**. If this option is less than or equal to zero then the window will not request any size at all.

DESCRIPTION

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

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

WIDGET METHODS

The **Toplevel** method creates a widget object. This object supports the **configure** and **cget** methods described in *Tk::options* which can be used to enquire and modify the options described above. The widget also inherits all the methods provided by the generic *Tk::Widget|Tk::Widget* class, and the *Tk::Wm|Tk::Wm* class.

BINDINGS

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

SEE ALSO

Tk::Widget|Tk::Widget Tk::Wm|Tk::Wm

KEYWORDS

toplevel, widget

NAME

Tk::Tree – Create and manipulate Tree widgets

=for pm Tixish/Tree.pm

=for category Tix Extensions

SYNOPSIS

use Tk::Tree;

$tree = *$parent*->**Tree**(*?options?*);

SUPER–CLASS

The **Tree** class is derived from the *HList\|Tk::HList* class and inherits all the methods, options and subwidgets of its super–class. A **Tree** widget is not scrolled by default.

STANDARD OPTIONS

Tree supports all the standard options of an HList widget. See *Tk::options* for details on the standard options.

WIDGET–SPECIFIC OPTIONS

Name: **browseCmd**
Class: **BrowseCmd**
Switch: **–browsecmd**

Specifies a *callback\|Tk::callbacks* to call whenever the user browses on an entry (usually by single–clicking on the entry). The callback is called with one argument, the pathname of the entry.

Name: **closeCmd**
Class: **CloseCmd**
Switch: **–closecmd**

Specifies a *callback\|Tk::callbacks* to call whenever an entry needs to be closed (See *"BINDINGS"* below). This method is called with one argument, the pathname of the entry. This method should perform appropriate actions to close the specified entry. If the **–closecmd** option is not specified, the default closing action is to hide all child entries of the specified entry.

Name: **command**
Class: **Command**
Switch: **–command**

Specifies a *callback\|Tk::callbacks* to call whenever the user activates an entry (usually by double–clicking on the entry). The callback is called with one argument, the pathname of the entry.

Name: **ignoreInvoke**
Class: **IgnoreInvoke**
Switch: **–ignoreinvoke**

A Boolean value that specifies when a branch should be opened or closed. A branch will always be opened or closed when the user presses the (+) and (–) indicators. However, when the user invokes a branch (by doublc–clicking or pressing <Return>), the branch will be opened or closed only if **–ignoreinvoke** is set to false (the default setting).

Name: **openCmd**
Class: **OpenCmd**
Switch: **–opencmd**

Specifies a *callback\|Tk::callbacks* to call whenever an entry needs to be opened (See *"BINDINGS"* below). This method is called with one argument, the pathname of the entry. This method should perform appropriate actions to open the specified entry. If the **–opencmd** option is not specified, the default opening action is to show all the child entries of the specified entry.

DESCRIPTION

The **Tree** method creates a new window and makes it into a Tree widget and return a reference to it. Additional options, described above, may be specified on the command line or in the option database to configure aspects of the Tree widget such as its cursor and relief.

The Tree widget can be used to display hierarchical data in a tree form. The user can adjust the view of the tree by opening or closing parts of the tree.

To display a static tree structure, you can add the entries into the Tree widget and hide any entries as desired. Then you can call the **autosetmode** method. This will set up the Tree widget so that it handles all the *open* and *close* events automatically. the demonstration program ***Tixish/examples/perl−tix−tree***).

The above method is not applicable if you want to maintain a dynamic tree structure, i.e, you do not know all the entries in the tree and you need to add or delete entries subsequently. To do this, you should first create the entries in the Tree widget. Then, use the **setmode** method to indicate the entries that can be opened or closed, and use the **−opencmd** and **−closecmd** options to handle the opening and closing events. (Please see the demonstration program ***Tixish/examples/perl−tix−dyntree***).

Use either

```
$parent->Scrolled('Tree', ... );
```

or

```
$parent->ScrlTree( ... );
```

to create a scrolled **Tree**. See *Tk::Scrolled* for details.

WIDGET METHODS

The **Tree** method creates a widget object. This object supports the **configure** and **cget** methods described in *Tk::options* which can be used to enquire and modify the options described above. The widget also inherits all the methods provided by the generic *Tk::Widget\|Tk::Widget* class.

The following additional methods are available for Tree widgets:

`$tree->`**autosetmode**

This method calls the **setmode** method for all the entries in this Tree widget: if an entry has no child entries, its mode is set to **none**. Otherwise, if the entry has any hidden child entries, its mode is set to **open**; otherwise its mode is set to **close**.

`$tree->`**close***(entryPath)*

Close the entry given by *entryPath* if its *mode* is **close**.

`$tree->`**getmode***(entryPath)*

Returns the current *mode* of the entry given by *entryPath*.

`$tree->`**open***(entryPath)*

Open the entry given by *entryPath* if its *mode* is **open**.

`$tree->`**setmode***(entryPath, mode*)

This method is used to indicate whether the entry given by *entryPath* has children entries and whether the children are visible. *mode* must be one of **open**, **close** or **none**. If *mode* is set to **open**, a (+) indicator is drawn next the the entry. If *mode* is set to **close**, a (−) indicator is drawn next the the entry. If *mode* is set to **none**, no indicators will be drawn for this entry. The default *mode* is none. The **open** mode indicates the entry has hidden children and this entry can be opened by the user. The **close** mode indicates that all the children of the entry are now visible and the entry can be closed by the user.

BINDINGS

The basic mouse and keyboard bindings of the Tree widget are the same as the *BINDINGS in bindings of the HList\|Tk::HList* widget. In addition, the entries can be opened or closed under

the following conditions:

[1] If the *mode* of the entry is **open**, it can be opened by clicking on its (+) indicator.

[2] If the *mode* of the entry is **close**, it can be closed by clicking on its (−) indicator.

SEE ALSO

Tk::HList\Tk::HList

AUTHOR

Perl/TK version by Chris Dean <ctdean@cogit.com. Original Tcl/Tix version by Ioi Kim Lam.

ACKNOWLEDGEMENTS

Thanks to Achim Bohnet <ach@mpe.mpg.de for all his help.

NAME

perl/Tk – Writing Tk applications in perl5.

=for category Introduction

DESCRIPTION

This manual page is for beginners. It assumes you know some perl, and have got perl+Tk running. Please run the 'widget' demo before reading this text; it will teach you the various widget types supported by Tk.

Some background

Tk GUI programming is event–driven. (This may already be familiar to you.) In event–driven programs, the main GUI loop is outside of the user program and inside the GUI library. This loop will watch all events of interest, and activate the correct handler procedures to handle these events. Some of these handler procedures may be user–supplied; others will be part of the library.

For a programmer, this means that you're not watching what is happening; instead, you are requested by the toolkit to perform actions whenever necessary. So, you're not watching for 'raise window / close window / redraw window' requests, but you tell the toolkit which routine will handle such cases, and the toolkit will call the procedures when required.

First requirements

Any perl program that uses Tk needs to include `use Tk`. A program should also use `use strict` and the **–w** switch to ensure the program is working without common errors.

Any Tk application starts by creating the Tk main window. You then create items inside the main window, or create new windows, before starting the mainloop. (You can also create more items and windows while you're running.) The items will be shown on the display after you `pack` them; more info on this later. Then you do a Tk mainloop; this will start the GUI and handle all events. That's your application. A trivial one–window example is show below:

```
#! /usr/bin/perl5 -w

use strict;
use Tk;

my $main = MainWindow->new;
$main->Label(-text => 'Hello, world!')->pack;
$main->Button(-text => 'Quit',
              -command => [$main => 'destroy']
              )->pack;
MainLoop;
```

Please run this example. It shows you two items types also shown in the widget demo; it also shows you how items are created and packed. Finally, note the typical Tk style using `-option => value` pairs.

Item creation

Tk windows and widgets are hierarchical, i.e. one includes one or more others. You create the first Tk window using `MainWindow->new`. This returns a window handle, assigned to `$main` in the example above. Keep track of the main handle.

You can use any Tk handle to create sub–items within the window or widget. This is done by calling the Tk constructor method on the variable. In the example above, the `Label` method called from `$main` creates a label widget inside the main window. In the constructor call, you can specify various options; you can later add or change options for any widget using the `configure` method, which takes the same parameters as the constructor. The one exception to the hierarchical structure is the `Toplevel` constructor, which creates a new outermost window.

After you create any widget, you must render it by calling `pack`. (This is not entirely true; more info later). If you do not need to refer to the widget after construction and packing, call `pack` off the constructor results, as shown for the label and button in the example above. Note that the result of the compound call is the result

of `pack`, which is a valid Tk handle.

Windows and widgets are deleted by calling `destroy` on them; this will delete and un–draw the widget and all its children.

Standard Tk types

Button
Radiobutton
Checkbutton
Listbox
Scrollbar
Entry
Text
Canvas
Frame
Toplevel
Scale
Menu
Menubutton

Variables and callback routines

Most graphical interfaces are used to set up a set of values and conditions, and then perform the appropriate action. The Tk toolkit is different from your average text–based prompting or menu driven system in that you do not collect settings yourself, and decide on an action based on an input code; instead, you leave these values to your toolkit and only get them when the action is performed.

So, where a traditional text–based system would look like this: (yes, this is obviously dumb code)

```
#! /usr/bin/perl5 -w

use strict;

print "Please type a font name\n";
my $font = <>; chomp $font;
# Validate font

print "Please type a file name\n";
my $filename = <>; chomp $filename;
# Validate filename

print "Type <1> to fax, <2> to print\n";
my $option = <>; chomp $option;
if ($option eq 1) {
    print "Faxing $filename in font $font\n";
} elsif ($option eq 2) {
    print "Now sending $filename to printer in font $font\n";
}
```

The (slightly larger) example below shows how to do this is Tk. Note the use of callbacks. Note, also, that Tk handles the values, and the subroutine uses `get` to get at the values. If a user changes his mind and wants to change the font again, the application never notices; it's all handled by Tk.

```
#! /usr/bin/perl5 -w

use strict;
use Tk;

my $main = MainWindow->new;
$main->Label(-text => 'Print file')->pack;
my $font = $main->Entry(-width => 10);
```

```
$font->pack;
my $filename = $main->Entry(-width => 10);
$filename->pack;
$main->Button(-text => 'Fax',
              -command => sub{do_fax($filename, $font)}
              )->pack;
$main->Button(-text => 'Print',
              -command => sub{do_print($filename, $font)}
              )->pack;
MainLoop;

sub do_fax {
    my ($file, $font) = @_;
    my $file_val = $file->get;
    my $font_val = $font->get;
    print "Now faxing $file_val in $font_val\n";
}

sub do_print {
    my ($file, $font) = @_;
    my $file_val = $file->get;
    my $font_val = $font->get;
    print "Sending file $file_val to printer in $font_val\n";
}
```

The packer. Grouping and frames.

In the examples above, you must have noticed the *pack|Tk::pack* calls. This is one of the more complicated parts of Tk. The basic idea is that any window or widget should be subject to a Tk widget placement manager; the *packer* is one of the placement managers.

The actions of the packer are rather simple: when applied to a widget, the packer positions that widget on the indicated position within the remaining space in its parent. By default, the position is on top; this means the next items will be put below. You can also specify the left, right, or bottom positions. Specify position using **–side => 'right'**.

Additional packing parameters specify the behavior of the widget when there is some space left in the frame or when the window size is increased. If widgets should maintain a fixed size, specify nothing; this is the default. For widgets that you want to fill up the current horizontal space, specify **–fill => 'x'**, **y**, or **both**; for widgets that should grow, specify **–expand => 1**. These parameters are not shown in the example below; see the widget demo.

If you want to group some items within a window that have a different packing order than others, you can include them in a Frame. This is a do–nothing window type that is meant for packing (and to play games with borders and colors).

The example below shows the use of pack and frames:

```
#! /usr/bin/perl5 -w

use strict;
use Tk;

# Take top, the bottom -> now implicit top is in the middle
my $main = MainWindow->new;
$main->Label(-text => 'At the top (default)')->pack;
$main->Label(-text => 'At the bottom')->pack(-side => 'bottom');
$main->Label(-text => 'The middle remains')->pack;

# Since left and right are taken, bottom will not work...
my $top1 = $main->Toplevel;
```

```
$top1->Label(-text => 'Left')->pack(-side => 'left');
$top1->Label(-text => 'Right')->pack(-side => 'right');
$top1->Label(-text => '?Bottom?')->pack(-side => 'bottom');

# But when you use frames, things work quite alright
my $top2 = $main->Toplevel;
my $frame = $top2->Frame;
$frame->pack;
$frame->Label(-text => 'Left2')->pack(-side => 'left');
$frame->Label(-text => 'Right2')->pack(-side => 'right');
$top2->Label(-text => 'Bottom2')->pack(-side => 'bottom');

MainLoop;
```

More than one window

Most real applications require more than one window. As you read before, you can create more outermost windows by using Toplevel. Each window is independent; destroying a toplevel window does not affect the others as long as they are not a child of the closed toplevel. Exiting the main window will end the application. The example below shows a trivial three−window application:

```
#! /usr/bin/perl5 -w

use strict;
use Tk;

my $main = MainWindow->new;
fill_window($main, 'Main');
my $top1 = $main->Toplevel;
fill_window($top1, 'First top-level');
my $top2 = $main->Toplevel;
fill_window($top2, 'Second top-level');
MainLoop;

sub fill_window {
    my ($window, $header) = @_;
    $window->Label(-text => $header)->pack;
    $window->Button(-text => 'close',
                    -command => [$window => 'destroy']
                    )->pack(-side => 'left');
    $window->Button(-text => 'exit',
                    -command => [$main => 'destroy']
                    )->pack(-side => 'right');
}
```

More callbacks

So far, all callback routines shown called a user procedure. You can also have a callback routine call another Tk routine. This is the way that scroll bars are implemented: scroll−bars can call a Tk item or a user procedure, whenever their position has changed. The Tk item that has a scrollbar attached calls the scrollbar when its size or offset has changed. In this way, the items are linked. You can still ask a scrollbar‘s position, or set it by hand – but the defaults will be taken care of.

The example below shows a listbox with a scroll bar. Moving the scrollbar moves the listbox. Scanning a listbox (dragging an item with the left mouse button) moves the scrollbar.

```
#! /usr/bin/perl5 -w

use strict;
use Tk;

my $main = MainWindow->new;
```

```
my $box = $main->Listbox(-relief => 'sunken',
                         -width => -1, # Shrink to fit
                         -height => 5,
                         -setgrid => 1);
my @items = qw(One Two Three Four Five Six Seven
               Eight Nine Ten Eleven Twelve);
foreach (@items) {
   $box->insert('end', $_);
}
my $scroll = $main->Scrollbar(-command => ['yview', $box]);
$box->configure(-yscrollcommand => ['set', $scroll]);
$box->pack(-side => 'left', -fill => 'both', -expand => 1);
$scroll->pack(-side => 'right', -fill => 'y');

MainLoop;
```

Canvases and tags

One of the most powerful window types in Tk is the Canvas window. In a canvas window, you can draw simple graphics and include other widgets. The canvas area may be larger than the visible window, and may then be scrolled. Any item you draw on the canvas has its own id, and may optionally have one or more *tags*. You may refer to any item by its id, and may refer to any group of items by a common tag; you can move, delete, or change groups of items using these tags, and you can *bind* actions to tags. For a properly designed (often structured) canvas, you can specify powerful actions quite simply.

In the example below, actions are bound to circles (single click) and blue items (double–click); obviously, this can be extended to any tag or group of tags.

```
#! /usr/bin/perl5 -w

use strict;
use Tk;

# Create main window and canvas
my $main = MainWindow->new;
my $canvas = $main->Canvas;
$canvas->pack(-expand => 1, -fill => 'both');

# Create various items
create_item($canvas, 1, 1, 'circle', 'blue', 'Jane');
create_item($canvas, 4, 4, 'circle', 'red', 'Peter');
create_item($canvas, 4, 1, 'square', 'blue', 'James');
create_item($canvas, 1, 4, 'square', 'red', 'Patricia');

# Single-clicking with left on a 'circle' item invokes a procedure
$canvas->bind('circle', '<1>' => sub {handle_circle($canvas)});
# Double-clicking with left on a 'blue' item invokes a procedure
$canvas->bind('blue', '<Double-1>' => sub {handle_blue($canvas)});
MainLoop;

# Create an item; use parameters as tags (this is not a default!)
sub create_item {
    my ($can, $x, $y, $form, $color, $name) = @_;

    my $x2 = $x + 1;
    my $y2 = $y + 1;
    my $kind;
    $kind = 'oval' if ($form eq 'circle');
    $kind = 'rectangle' if ($form eq 'square');
    $can->create(($kind, "$x" . 'c', "$y" . 'c',
```

```
                    "$x2" . 'c', "$y2" . 'c'),
                   -tags => [$form, $color, $name],
                   -fill => $color);
}

# This gets the real name (not current, blue/red, square/circle)
# Note: you'll want to return a list in realistic situations...
sub get_name {
    my ($can) = @_;
    my $item = $can->find('withtag', 'current');
    my @taglist = $can->gettags($item);
    my $name;
    foreach (@taglist) {
        next if ($_ eq 'current');
        next if ($_ eq 'red' or $_ eq 'blue');
        next if ($_ eq 'square' or $_ eq 'circle');
        $name = $_;
        last;
    }
    return $name;
}

sub handle_circle {
    my ($can) = @_;
    my $name = get_name($can);
    print "Action on circle $name...\n";
}

sub handle_blue {
    my ($can) = @_;
    my $name = get_name($can);
    print "Action on blue item $name...\n";
}
```

NAME

Tk::Widget – Base class of all widgets

=for pm Tk/Widget.pm

=for category Tk Generic Methods

SYNOPSIS

```
package Tk::Whatever;
require Tk::Widget;
@ISA = qw(Tk::Widget);
Construct Tk::Widget 'Whatever';

sub Tk_cmd { \&Tk::whatever }
```

$widget->method(?arg, arg, ...?)

DESCRIPTION

The **Tk::Widget** is an abstract base class for all Tk widgets.

Generic methods available to all widgets include the methods based on core `winfo` mechanism and are used to retrieve information about windows managed by Tk. They can take any of a number of different forms, depending on the *method*. The legal forms are:

$widget->**appname***?(newName)?*

If *newName* isn't specified, this method returns the name of the application (the name that may be used in **send** commands to communicate with the application). If *newName* is specified, then the name of the application is changed to *newName*. If the given name is already in use, then a suffix of the form '' **#2**'' or '' **#3**'' is appended in order to make the name unique. The method's result is the name actually chosen. *newName* should not start with a capital letter. This will interfere with *option|Tk::option* processing, since names starting with capitals are assumed to be classes; as a result, Tk may not be able to find some options for the application. If sends have been disabled by deleting the **send** command, this command will reenable them and recreate the **send** command.

$widget->**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.

$widget->**atomname***(id)*

Returns the textual name for the atom whose integer identifier is *id*. This command is the inverse of the *$widget*->**atom** command. It generates an error if no such atom exists.

$widget->**bell**

This command rings the bell on the display for *$widget* and returns an empty string. The command uses the current bell–related settings for the display, which may be modified with programs such as **xset**.

This command also resets the screen saver for the screen. Some screen savers will ignore this, but others will reset so that the screen becomes visible again.

$widget->**Busy***?(?–recurse* = 1?*–option* = value?)?

This method **configure**s a **–cursor** option for *$widget* and (if **–recurse =** 1 is specified) all its descendants. The cursor to be set may be passed as **–cursor** = *cursor* or defaults to 'watch'. Additional **configure** options are applied to *$widget* only. It also adds a special tag **'Busy'** to the **bindtags** of the widgets so configured so that **KeyPress**, **KeyRelease**, **ButtonPress** and **ButtonRelease** events are ignored (with press events generating a call to **bell**). It then acquires a local **grab** for *$widget*. The state of the widgets and the grab is restored by a call to *$widget*->**Unbusy**.

$widget->**cells**

Returns a decimal string giving the number of cells in the color map for *$widget*.

$widget->**children**

$widget-**children** Returns a list containing all the children of `$widget`. The list is in stacking order, with the lowest window first. Top-level windows are returned as children of their logical parents.

$widget->**class**

Returns the class name for *$widget*.

$widget->**colormapfull**

Returns 1 if the colormap for *$widget* is known to be full, 0 otherwise. The colormap for a window is ''known'' to be full if the last attempt to allocate a new color on that window failed and this application hasn't freed any colors in the colormap since the failed allocation.

$widget->**containing***(rootX,rootY)*

Returns 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.

$widget->**depth**

Returns a decimal string giving the depth of *$widget* (number of bits per pixel).

$widget->**destroy**

This command deletes the window related to *$widget*, plus all its descendants. If all the **MainWindows** are deleted then the entire application will be destroyed.

The perl object *$widget* continues to exist while references to it still exist, e.g. until variable goes out of scope. However any attempt to use Tk methods on the object will fail. **Exists**(*$widget)* will return false on such objects.

Note however that while a window exists for *$widget* the perl object is maintained (due to "references" in perl/Tk internals) even though original variables may have gone out of scope. (Normally this is intuitive.)

Exists(*$widget)*

Returns 1 if there exists a window for *$widget*, 0 if no such window exists.

$widget->**font***(option?, arg, arg, ...?)*

Create and inspect fonts. See *Tk::Font* for further details.

$widget->**fpixels***(number)*

Returns a floating-point value giving the number of pixels in *$widget* corresponding to the distance given by *number*. *Number* may be specified in any of the forms acceptable to **Tk_GetScreenMM**, such as ''2.0c'' or ''1i''. The return value may be fractional; for an integer value, use *$widget*->**pixels**.

$widget->**Getimage***(name)*

Given *name*, look for an image file with that base name and return a *Tk::Image*. File extensions are tried in this order: ***xpm***, ***gif***, ***ppm***, ***xbm*** until a valid iamge is found. If no image is found, try a builtin image with that name.

$widget->**geometry**

Returns the geometry for *$widget,* in the form *width***x***height***+***x***+***y*. All dimensions are in pixels.

$widget->**height**

Returns a decimal string giving *$widget's* height in pixels. When a window is first created its height will be 1 pixel; the height will eventually be changed by a geometry manager to fulfill the window's needs. If you need the true height immediately after creating a widget, invoke **update** to force the geometry manager to arrange it, or use *$widget*->**reqheight** to get the window's requested height instead of its actual height.

$widget->**id**

Returns a hexadecimal string giving a low-level platform-specific identifier for `$widget`. On Unix platforms, this is the X window identifier. Under Windows, this is the Windows HWND. On the Macintosh the value has no meaning outside Tk.

$widget->**idletasks**

One of two methods which are used to bring the application ''up to date'' by entering the event loop repeated until all pending events (including idle callbacks) have been processed.

If the **idletasks** method is specified, then no new events or errors are processed; only idle callbacks are invoked. This causes operations that are normally deferred, such as display updates and window layout calculations, to be performed immediately.

The **idletasks** command is useful in scripts where changes have been made to the application's state and you want those changes to appear on the display immediately, rather than waiting for the script to complete. Most display updates are performed as idle callbacks, so **idletasks** will cause them to run. However, there are some kinds of updates that only happen in response to events, such as those triggered by window size changes; these updates will not occur in **idletasks**.

$widget->**interps**

Returns a list whose members are the names of all Tcl interpreters (e.g. all Tk-based applications) currently registered for a particular display. The return value refers to the display of *$widget*.

$widget->**ismapped**

Returns **1** if *$widget* is currently mapped, otherwise.

$widget->**lower**(?*belowThis*?)

If the *belowThis* argument is omitted then the command lowers `$widget` so that it is below all of its siblings in the stacking order (it will be obscured by any siblings that overlap it and will not obscure any siblings). If *belowThis* is specified then it must be the path name of a window that is either a sibling of `$widget` or the descendant of a sibling of `$widget`. In this case the **lower** command will insert `$widget` into the stacking order just below *belowThis* (or the ancestor of *belowThis* that is a sibling of `$widget`); this could end up either raising or lowering `$widget`.

$widget->**MapWindow**

Cause *$widget* to be "mapped" i.e. made visible on the display. May confuse the geometry manager (pack, grid, place, ...) that thinks it is managing the widget.

$widget->**manager**

Returns the name of the geometry manager currently responsible for *$widget,* or an empty string if *$widget* isn't managed by any geometry manager. The name is usually the name of the method for the geometry manager, such as **pack** or **place**. If the geometry manager is a widget, such as canvases or text, the name is the widget's class command, such as **canvas**.

$widget->**name**

Returns *$widget's* name (i.e. its name within its parent, as opposed to its full path name). The command *$mainwin*->**name** will return the name of the application.

$widget->**OnDestroy***(callback);*

OnDestroy accepts a standard perl/Tk *callback*. When the window associated with *$widget* is destroyed then the callback is invoked. Unlike *$widget*-bind('<Destroy>',...) the widgets methods are still available when *callback* is executed, so (for example) a **Text** widget can save its contents to a file.

OnDestroy was required for new **after** mechanism.

$widget->**parent**

Returns *$widget's* parent, or an empty string if *$widget* is the main window of the application.

$widget->**PathName**

Returns the tk path name of *$widget*. (This is an import from the C interface.)

$widget->**pathname***(id)*

Returns an object whose X identifier is *id*. The identifier is looked up on the display of *$widget*. *Id* must be a decimal, hexadecimal, or octal integer and must correspond to a window in the invoking application, or an error occurs which can be trapped with `eval { }` or `Tk::catch { }`. If the window belongs to the application, but is not an object (for example wrapper windows, HList header, etc.) then `undef` is returned.

$widget->**pixels***(number)*

Returns the number of pixels in *$widget* corresponding to the distance given by *number*. *Number* may be specified in any of the forms acceptable to **Tk_GetPixels**, such as ''2.0c'' or ''1i''. The result is rounded to the nearest integer value; for a fractional result, use *$widget*->**fpixels**.

$widget->**pointerx**

If the mouse pointer is on the same screen as *$widget,* returns the pointer's x coordinate, measured in pixels in the screen's root window. If a virtual root window is in use on the screen, the position is measured in the virtual root. If the mouse pointer isn't on the same screen as *$widget* then −1 is returned.

$widget->**pointerxy**

If the mouse pointer is on the same screen as *$widget,* returns a list with two elements, which are the pointer's x and y coordinates measured in pixels in the screen's root window. If a virtual root window is in use on the screen, the position is computed in the virtual root. If the mouse pointer isn't on the same screen as *$widget* then both of the returned coordinates are −1.

$widget->**pointery**

If the mouse pointer is on the same screen as *$widget,* returns the pointer's y coordinate, measured in pixels in the screen's root window. If a virtual root window is in use on the screen, the position is computed in the virtual root. If the mouse pointer isn't on the same screen as *$widget* then −1 is returned.

$widget->**raise***(?aboveThis?)*

If the *aboveThis* argument is omitted then the command raises `$widget` so that it is above all of its siblings in the stacking order (it will not be obscured by any siblings and will obscure any siblings that overlap it). If *aboveThis* is specified then it must be the path name of a window that is either a sibling of `$widget` or the descendant of a sibling of `$widget.` In this case the **raise** command will insert `$widget` into the stacking order just above *aboveThis* (or the ancestor of *aboveThis* that is a sibling of `$widget);` this could end up either raising or lowering `$widget.`

$widget->**reqheight**

Returns a decimal string giving *$widget's* requested height, in pixels. This is the value used by *$widget's* geometry manager to compute its geometry.

$widget->**reqwidth**

Returns a decimal string giving *$widget's* requested width, in pixels. This is the value used by *$widget's* geometry manager to compute its geometry.

$widget->**rgb***(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 *$widget*. *Color* may be specified in any of the forms acceptable for a color option.

$widget->**rootx**

Returns a decimal string giving the x–coordinate, in the root window of the screen, of the upper–left corner of *$widget's* border (or *$widget* if it has no border).

$widget->**rooty**

Returns a decimal string giving the y–coordinate, in the root window of the screen, of the upper–left corner of *$widget's* border (or *$widget* if it has no border).

scaling

$widget->**scaling***?(number)?*

Sets and queries the current scaling factor used by Tk to convert between physical units (for example, points, inches, or millimeters) and pixels. The *number* argument is a floating point number that specifies the number of pixels per point on $widget's display. If the *number* argument is omitted, the current value of the scaling factor is returned.

A ''point'' is a unit of measurement equal to 1/72 inch. A scaling factor of 1.0 corresponds to 1 pixel per point, which is equivalent to a standard 72 dpi monitor. A scaling factor of 1.25 would mean 1.25 pixels per point, which is the setting for a 90 dpi monitor; setting the scaling factor to 1.25 on a 72 dpi monitor would cause everything in the application to be displayed 1.25 times as large as normal. The initial value for the scaling factor is set when the application starts, based on properties of the installed monitor (as reported via the window system), but it can be changed at any time. Measurements made after the scaling factor is changed will use the new scaling factor, but it is undefined whether existing widgets will resize themselves dynamically to accomodate the new scaling factor.

$widget->**screen**

Returns the name of the screen associated with *$widget,* in the form *displayName.screenIndex*.

$widget->**screencells**

Returns a decimal string giving the number of cells in the default color map for *$widget's* screen.

$widget->**screendepth**

Returns a decimal string giving the depth of the root window of *$widget's* screen (number of bits per pixel).

$widget->**screenheight**

Returns a decimal string giving the height of *$widget's* screen, in pixels.

$widget->**screenmmheight**

Returns a decimal string giving the height of *$widget's* screen, in millimeters.

$widget->**screenmmwidth**

Returns a decimal string giving the width of *$widget's* screen, in millimeters.

$widget->**screenvisual**

Returns one of the following strings to indicate the default visual class for *$widget's* screen: **directcolor**, **grayscale**, **pseudocolor**, **staticcolor**, **staticgray**, or **truecolor**.

$widget->**screenwidth**

Returns a decimal string giving the width of *$widget's* screen, in pixels.

$widget->**server**

Returns a string containing information about the server for *$widget's* display. The exact format of this string may vary from platform to platform. For X servers the string has the form ''**X***major***R***minor vendor vendorVersion*'' where *major* and *minor* are the version and revision numbers provided by the server (e.g., **X11R5**), *vendor* is the name of the vendor for the server, and *vendorRelease* is an integer release number provided by the server.

$widget->**toplevel**

Returns the reference of the top−level window containing *$widget*.

$widget->**UnmapWindow**

Cause *$widget* to be "unmapped" i.e. removed from the display. This does for any widget what *$widget*->*withdraw* does for toplevel widgets. May confuse the geometry manager (pack, grid, place, ...) that thinks it is managing the widget.

$widget->**update**

One of two methods which are used to bring the application ''up to date'' by entering the event loop repeated until all pending events (including idle callbacks) have been processed.

The **update** method is useful in scripts where you are performing a long−running computation but you still want the application to respond to events such as user interactions; if you occasionally call **update** then user input will be processed during the next call to **update**.

$widget->**Unbusy**

Restores widget state after a call to *$widget*->**Busy**.

$widget->**viewable**

Returns 1 if *$widget* and all of its ancestors up through the nearest toplevel window are mapped. Returns 0 if any of these windows are not mapped.

$widget->**visual**

Returns one of the following strings to indicate the visual class for *$widget:* **directcolor**, **grayscale**, **pseudocolor**, **staticcolor**, **staticgray**, or **truecolor**.

$widget->**visualid**

Returns the X identifier for the visual for `$widget`.

$widget->**visualsavailable***(?***includeids***?)***

Returns a list whose elements describe the visuals available for *$widget's* screen. Each element consists of a visual class followed by an integer depth. The class has the same form as returned by *$widget*->**visual**. The depth gives the number of bits per pixel in the visual. In addition, if the **includeids** argument is provided, then the depth is followed by the X identifier for the visual.

$widget->**vrootheight**

Returns the height of the virtual root window associated with *$widget* if there is one; otherwise returns the height of *$widget's* screen.

$widget->**vrootwidth**

Returns the width of the virtual root window associated with *$widget* if there is one; otherwise returns the width of *$widget's* screen.

$widget->**vrootx**

Returns the x−offset of the virtual root window associated with *$widget,* relative to the root window of its screen. This is normally either zero or negative. Returns 0 if there is no virtual root window for *$widget*.

$widget->**vrooty**

Returns the y–offset of the virtual root window associated with *$widget*, relative to the root window of its screen. This is normally either zero or negative. Returns 0 if there is no virtual root window for *$widget*.

$widget–**waitVariable**(\$*name*)
$widget–**waitVisibility**
$widget–**waitWindow**

The **tk wait** methods wait for one of several things to happen, then it returns without taking any other actions. The return value is always an empty string. **waitVariable** expects a reference to a perl variable and the command waits for that variable to be modified. This form is typically used to wait for a user to finish interacting with a dialog which sets the variable as part (possibly final) part of the interaction. **waitVisibility** waits for a change in *$widget's* visibility state (as indicated by the arrival of a VisibilityNotify event). This form is typically used to wait for a newly–created window to appear on the screen before taking some action. **waitWindow** waits for *$widget* 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. Note that creating and destroying the window each time a dialog is required makes code modular but imposes overhead which can be avoided by **withdrawing** the window instead and using **waitVisibility**.

While the **tk wait** methods are waiting they processes events in the normal fashion, so the application will continue to respond to user interactions. If an event handler invokes **tkwait** again, the nested call to **tkwait** must complete before the outer call can complete.

$widget->**width**

Returns a decimal string giving *$widget's* width in pixels. When a window is first created its width will be 1 pixel; the width will eventually be changed by a geometry manager to fulfill the window's needs. If you need the true width immediately after creating a widget, invoke **update** to force the geometry manager to arrange it, or use *$widget*->**reqwidth** to get the window's requested width instead of its actual width.

$widget->**x**

Returns a decimal string giving the x–coordinate, in *$widget's* parent, of the upper–left corner of *$widget's* border (or *$widget* if it has no border).

$widget->**y**

Returns a decimal string giving the y–coordinate, in *$widget's* parent, of the upper–left corner of *$widget's* border (or *$widget* if it has no border).

CAVEATS

The above documentaion on generic methods is incomplete.

KEYWORDS

atom, children, class, geometry, height, identifier, information, interpreters, mapped, parent, path name, screen, virtual root, width, window

NAME

`WidgetDemo()` – create a standard widget demonstration window.

=for pm demos/demos/widget_lib/WidgetDemo.pm

=for category Implementation

SYNOPSIS

```
use WidgetDemo;
my $TOP = $MW->WidgetDemo(
    -name             => $demo,
    -text             => 'Learn how to write a widget demonstration!',
    -title            => 'WidgetDemo Demonstration',
    -iconname         => 'WidgetDemo',
    -geometry_manager => 'grid',
    -font             => $FONT,
);
```

DESCRIPTION

This constructor builds a standard widget demonstration window, composed of three frames. The top frame contains descriptive demonstration text. The bottom frame contains the "Dismiss" and "See Code" buttons. The middle frame is demonstration container, which came be managed by either the pack or grid geometry manager.

The –text attribute is supplied to a Label widget, which is left–adjusted with –wraplength set to 4 inches. If you require different specifications then pass an array to –text; the first element is the text string and the remaining array elements are standard Label widget attributes – WidgetDemo will rearrange things as required..

```
    -text => ['Hello World!', qw/-wraplength 6i/],
```

AUTHOR

Steve Lidie <Stephen.O.Lidie@Lehigh.EDU

HISTORY

lusol@Lehigh.EDU, LUCC, 97/02/11 lusol@Lehigh.EDU, LUCC, 97/06/07
Stephen.O.Lidie@Lehigh.EDU, LUCC, 97/06/07
. Add `Delegates()` call that obviates the need for `Top()`. Many thanks to
Achim Bohnet for this patch.
. Fix –title so that it works.

COPYRIGHT

NAME

Tk::widgets – preload widget classes

=for pm Tk/widgets.pm

=for category Implementation

SYNOPSIS

```
use Tk::widgets qw(Button Label Frame);
```

DESCRIPTION

Does a 'require Tk::Foo' for each 'Foo' in the list. May speed startup by avoiding AUTOLOADs.

NAME

Tk::Wm – Communicate with window manager

=for category Tk Geometry Management

SYNOPSIS

$toplevel->method(?args?)

DESCRIPTION

The **wm** methods are 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** methods can take any of a number of different forms, depending on the particular *method* argument. All of the forms expect *$toplevel,* which must be a top–level window object.

The legal forms for the **wm** methods are:

*$toplevel->***aspect***(?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 *$toplevel*. The aspect ratio of *$toplevel* (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 method returns an empty string. Otherwise, it returns a array 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).

*$toplevel->***client***(?name?)*

If *name* is specified, this method stores *name* (which should be the name of the host on which the application is executing) in *$toplevel's* **WM_CLIENT_MACHINE** property for use by the window manager or session manager. The method returns an empty string in this case. If *name* isn't specified, the method returns the last name set in a **client** method for *$toplevel*. If *name* is specified as an empty string, the method deletes the **WM_CLIENT_MACHINE** property from *$toplevel*.

*$toplevel->***colormapwindows***(?windowList?)*

This method is used to manipulate the **WM_COLORMAP_WINDOWS** property, which provides information to the window managers about windows that have private colormaps. If *windowList* isn't specified, the method returns a list whose elements are the names of the windows in the **WM_COLORMAP_WINDOWS** property. If *windowList* is specified, it consists of a list of widgets; the method overwrites the **WM_COLORMAP_WINDOWS** property with the given windows and returns an empty string. The **WM_COLORMAP_WINDOWS** property should normally contain a list of the internal windows within *$toplevel* whose colormaps differ from their parents. The order of the windows in the property indicates a priority order: the window manager will attempt to install as many colormaps as possible from the head of this list when *$widget* gets the colormap focus. If *$widget* is not included among the windows in *windowList*, Tk implicitly adds it at the end of the **WM_COLORMAP_WINDOWS** property, so that its colormap is lowest in priority. If *$widget->colormapwindows* is not invoked, Tk will automatically set the property for each top–level window to all the internal windows whose colormaps differ from their parents, followed by the top–level itself; the order of the internal windows is undefined. See the ICCCM documentation for more information on the **WM_COLORMAP_WINDOWS** property.

*$toplevel->***command***(?value?)*

If *value* is specified, this method stores *value* in *$toplevel's* **WM_COMMAND** property for use by the window manager or session manager and returns an empty string. *Value* must have proper list structure; the elements should contain the words of the command used to invoke the application. If *value* isn't specified then the method returns the last value set in a **command** method for *$toplevel*. If *value* is specified as an empty string, the method deletes the **WM_COMMAND**

property from *$toplevel*.

$toplevel->**deiconify**

Arrange for *$toplevel* to be displayed in normal (non−iconified) form. This is done by mapping the window. If the window has never been mapped then this method 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.

$toplevel->**focusmodel***(?***active**/**passive***?)*

If **active** or **passive** is supplied as an optional argument to the method, then it specifies the focus model for *$toplevel*. In this case the method returns an empty string. If no additional argument is supplied, then the method returns the current focus model for *$toplevel*. An **active** focus model means that *$toplevel* will claim the input focus for itself or its descendants, even at times when the focus is currently in some other application. **Passive** means that *$toplevel* will never claim the focus for itself: the window manager should give the focus to *$toplevel* at appropriate times. However, once the focus has been given to *$toplevel* or one of its descendants, the application may re−assign the focus among *$toplevel's* descendants. The focus model defaults to **passive**, and Tk's **focus** method assumes a passive model of focusing.

$toplevel->**frame**

If *$widget* has been reparented by the window manager into a decorative frame, the method returns the platform specific window identifier for the outermost frame that contains *$toplevel* (the window whose parent is the root or virtual root). If *$toplevel* hasn't been reparented by the window manager then the method returns the platform specific window identifier for *$toplevel*.

$toplevel->**geometry***(?newGeometry?)*

If *newGeometry* is specified, then the geometry of *$toplevel* is changed and an empty string is returned. Otherwise the current geometry for *$toplevel* is returned (this is the most recent geometry specified either by manual resizing or in a **geometry** method). *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 *$toplevel*. If *$toplevel* 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 *$toplevel* on the screen, in pixels. If *x* is preceded by **+**, it specifies the number of pixels between the left edge of the screen and the left edge of *$toplevel's* border; if preceded by **−** then *x* specifies the number of pixels between the right edge of the screen and the right edge of *$toplevel's* border. If *y* is preceded by **+** then it specifies the number of pixels between the top of the screen and the top of *$toplevel's* border; if *y* is preceded by **−** then it specifies the number of pixels between the bottom of *$toplevel's* border and the bottom of the screen. If *newGeometry* is specified as an empty string then any existing user−specified geometry for *$toplevel* is cancelled, and the window will revert to the size requested internally by its widgets.

$toplevel->**wmGrid***(?baseWidth,baseHeight,widthInc,heightInc?)*

This method indicates that *$toplevel* is to be managed as a gridded window. It also specifies the relationship between grid units and pixel units. *BaseWidth* and *baseHeight* specify the number of grid units corresponding to the pixel dimensions requested internally by *$toplevel* 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 *$toplevel*, corresponding to grid−based widths and heights that are non−negative integers. Tk will pass this information to the window manager; during manual resizing, the window manager will restrict the window's size to one of these acceptable sizes. Furthermore, during manual resizing the window manager will display the window's current size in terms of grid units rather than pixels. If *baseWidth* etc. are all specified as empty strings, then *$toplevel* 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 array containing four elements corresponding to the current *baseWidth*, *baseHeight*, *widthInc*, and *heightInc*; if *$toplevel* 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.

`$toplevel`**–>group***(?$widget?)*

If `$widget` is specified, it is the the leader of a group of related windows. The window manager may use this information, for example, to unmap all of the windows in a group when the group's leader is iconified. `$widget` may be specified as an empty string to remove `$toplevel` from any group association. If `$widget` is specified then the method returns an empty string; otherwise it returns the `$toplevel`*'s* current group leader, or an empty string if `$toplevel` isn't part of any group.

`$toplevel`**–>iconbitmap***(?bitmap?)*

If *bitmap* is specified, then it names a bitmap in the standard forms accepted by Tk (see the **Tk_GetBitmap** documentation for details). This *black and white* bitmap is passed to the window manager to be displayed in `$toplevel`*'s* icon, and the method returns an empty string. If an empty string is specified for *bitmap*, then any current icon bitmap or image is cancelled for `$toplevel`. If *bitmap* is specified then the method returns an empty string. Otherwise it returns the name of the current icon bitmap associated with `$toplevel`, or an empty string if `$toplevel` has no icon bitmap.

`$toplevel`**–>iconify**

Arrange for `$toplevel` to be iconified. It `$toplevel` hasn't yet been mapped for the first time, this method will arrange for it to appear in the iconified state when it is eventually mapped.

`$toplevel`**–>iconimage***(?image?)*

If *image* is specified, then it names a normal Tk image. This image is rendered into a private *coloured* bitmap which is passed to the window manager to be displayed in `$toplevel`*'s* icon, and the method returns an empty string. If an empty string is specified for *image*, then any current icon bitmap or image is cancelled for `$toplevel`. If *image* is specified then the method returns an empty string. Otherwise it returns the name of the current icon image associated with `$toplevel`, or an empty string if `$toplevel` has no icon image. The private pixmap is not pre–cleared so images which are partly "transparent" display rubbish in their transparent parts.

The sizes of images that can be used as icons in this manner are platform dependant. On Win32 this sets the "large" icon, which should be 32x32, it will automatically be scaled down to 16x16 for use as a small icon.

`$toplevel`**–>iconmask***(?bitmap?)*

If *bitmap* is specified, then it names a bitmap in the standard forms accepted by Tk (see the **Tk_GetBitmap** documentation 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 `$toplevel` (this is equivalent to specifying a bitmap of all ones). If *bitmap* is specified then the method returns an empty string. Otherwise it returns the name of the current icon mask associated with `$toplevel`, or an empty string if no mask is in effect.

`$toplevel`**–>iconname***(?newName?)*

If *newName* is specified, then it is passed to the window manager; the window manager should display *newName* inside the icon associated with `$toplevel`. In this case an empty string is returned as result. If *newName* isn't specified then the method returns the current icon name for `$toplevel`, 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 **title** method).

`$toplevel`**–>iconposition***(?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 `$toplevel`. 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 method returns

a array containing two values, which are the current icon position hints (if no hints are in effect then an empty string is returned).

$toplevel*->**iconwindow***(?$widget?)

If *$widget* is specified, it is a window to use as icon for *$toplevel:* when *$toplevel* is iconified then *$widget* will be mapped to serve as icon, and when *$toplevel* is de−iconified then *$widget* will be unmapped again. If *$widget* is specified as an empty string then any existing icon window association for *$toplevel* will be cancelled. If the *$widget* argument is specified then an empty string is returned. Otherwise the method returns the current icon window for *$toplevel,* or an empty string if there is no icon window currently specified for *$toplevel*. Button press events are disabled for *$toplevel* as long as it is an icon window; this is needed in order to allow window managers to ''own'' those events. Note: not all window managers support the notion of an icon window.

$toplevel->**maxsize***(?width,height?)*

If *width* and *height* are specified, they give the maximum permissible dimensions for *$toplevel*. For gridded windows the dimensions are specified in grid units; otherwise they are specified in pixel units. The window manager will restrict the window's dimensions to be less than or equal to *width* and *height*. If *width* and *height* are specified, then the method returns an empty string. Otherwise it returns a array with two elements, which are the maximum width and height currently in effect. The maximum size defaults to the size of the screen. If resizing has been disabled with the **resizable** method, then this method has no effect. See the sections on geometry management below for more information.

$toplevel->**minsize***(?width,height?)*

If *width* and *height* are specified, they give the minimum permissible dimensions for *$toplevel*. For gridded windows the dimensions are specified in grid units; otherwise they are specified in pixel units. The window manager will restrict the window's dimensions to be greater than or equal to *width* and *height*. If *width* and *height* are specified, then the method returns an empty string. Otherwise it returns a array with two elements, which are the minimum width and height currently in effect. The minimum size defaults to one pixel in each dimension. If resizing has been disabled with the **resizable** method, then this method has no effect. See the sections on geometry management below for more information.

$toplevel->**overrideredirect(?***boolean?)*

If *boolean* is specified, it must have a proper boolean form and the override−redirect flag for *$toplevel* is set to that value. If *boolean* is not specified then **1** or is returned to indicate whether or not the override−redirect flag is currently set for *$toplevel*. 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.

$toplevel->**positionfrom***(?who?)*

If *who* is specified, it must be either **program** or **user**, or an abbreviation of one of these two. It indicates whether *$toplevel's* current position was requested by the program or by the user. Many window managers ignore program−requested initial positions and ask the user to manually position the window; if **user** is specified then the window manager should position the window at the given place without asking the user for assistance. If *who* is specified as an empty string, then the current position source is cancelled. If *who* is specified, then the method returns an empty string. Otherwise it returns **user** or `$widget` 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 **geometry** method is invoked, unless the source has been set explicitly to **program**.

$toplevel->**protocol***(?name?,?callback?)*

This method 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 *callback* are specified, then *callback* is associated with the protocol specified by *name*. *Name* will be added to *$toplevel's* **WM_PROTOCOLS** property to tell the window manager that the application has a protocol handler for *name*, and *callback* will be invoked in the future whenever the window manager sends a message to the client for that protocol. In this case the method returns an empty string. If *name* is specified but *callback* isn't, then the current callback for *name* is returned, or an empty string if there is no handler defined for *name*. If *callback* is specified as an empty string then the current handler for *name* is deleted and it is removed from the **WM_PROTOCOLS** property on *$toplevel;* an empty string is returned. Lastly, if neither *name* nor *callback* is specified, the method returns a list of all the protocols for which handlers are currently defined for *$toplevel*.

Tk always defines a protocol handler for **WM_DELETE_WINDOW**, even if you haven't asked for one with **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.

$toplevel->**resizable***(?width,height?)*

This method controls whether or not the user may interactively resize a top–level window. If *width* and *height* are specified, they are boolean values that determine whether the width and height of *$toplevel* may be modified by the user. In this case the method returns an empty string. If *width* and *height* are omitted then the method returns a list with two 0/1 elements that indicate whether the width and height of *$toplevel* are currently resizable. By default, windows are resizable in both dimensions. If resizing is disabled, then the window's size will be the size from the most recent interactive resize or **geometry** method. If there has been no such operation then the window's natural size will be used.

$toplevel->**sizefrom***(?who?)*

If *who* is specified, it must be either **program** or **user**, or an abbreviation of one of these two. It indicates whether *$toplevel's* current size was requested by the program or by the user. Some window managers ignore program–requested sizes and ask the user to manually size the window; if **user** is specified then the window manager should give the window its specified size without asking the user for assistance. If *who* is specified as an empty string, then the current size source is cancelled. If *who* is specified, then the method returns an empty string. Otherwise it returns **user** or `$widget` 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**.

$toplevel->**state**

Returns the current state of `$widget`: either **normal**, **iconic**, **withdrawn**, or **icon**. The difference between **iconic** and **icon** is that **iconic** refers to a window that has been iconified (e.g., with the **iconify** method) while **icon** refers to a window whose only purpose is to serve as the icon for some other window (via the **iconwindow** method).

$toplevel->**title***(?string?)*

If *string* is specified, then it will be passed to the window manager for use as the title for *$toplevel* (the window manager should display this string in *$toplevel's* title bar). In this case the method returns an empty string. If *string* isn't specified then the method returns the current title for the *$toplevel*. The title for a window defaults to its name.

$toplevel->**transient***(?master?)*

If *master* is specified, then the window manager is informed that *$toplevel* is a transient window (e.g. pull–down menu) working on behalf of *master* (where *master* is a top–level window). Some window managers will use this information to manage *$toplevel* specially. If *master* is specified as an empty string then *$toplevel* is marked as not being a transient window any more. If *master* is specified, then the method returns an empty string. Otherwise the method returns the path name of *$toplevel's* current master, or an empty string if *$toplevel* isn't currently a transient window.

$toplevel->**withdraw**

Arranges for *$toplevel* 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 method 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 **deiconify**) to get some window managers to pay attention to changes in window attributes such as group.

$toplevel->**wrapper**

Returns the window id of the wrapper window in which Tk has placed *$toplevel*. This is the id by which window manager will know *$toplevel,* and so is appropriate place to add X properties.

ICON SIZES

The sizes of bitmaps/images that can be used as icons in this manner are platform and window manager dependant. Unix window managers are typically more tolerant than Win32. It is possible that coloured `iconimage` icons may cause problems on some X window managers.

- Win32

  `iconimage` and `iconbitmap` set the "large" icon, which should be 32x32, it will automatically be scaled down to 16x16 for use as a small icon. Win32 ignores `iconwin` requests.

- KDE's "kwm"

  Accepts coloured `iconimage` and black and white `iconbitmap` but will scale either to a small (14x14?) icon. Kwm ignores `iconwin`.

- Sun's "olwm" or "olvwm"

  Honours `iconwin` which will override `iconimage` or `iconbitmap`. Coloured images work.

- Sun's CDE window manager

  Coloured images work. ...

GEOMETRY MANAGEMENT

By default a top−level window appears on the screen in its *natural size*, which is the one determined internally by its widgets and geometry managers. If the natural size of a top−level window changes, then the window's size changes to match. A top−level window can be given a size other than its natural size in two ways. First, the user can resize the window manually using the facilities of the window manager, such as resize handles. Second, the application can request a particular size for a top−level window using the **geometry** method. These two cases are handled identically by Tk; in either case, the requested size overrides the natural size. You can return the window to its natural by invoking **geometry** with an empty *geometry* string.

Normally a top−level window can have any size from one pixel in each dimension up to the size of its screen. However, you can use the **minsize** and **maxsize** methods to limit the range of allowable sizes. The range set by **minsize** and **maxsize** applies to all forms of resizing, including the window's natural size as well as manual resizes and the **geometry** method. You can also use the method **resizable** to completely disable interactive resizing in one or both dimensions.

GRIDDED GEOMETRY MANAGEMENT

Gridded geometry management occurs when one of the widgets of an application supports a range of useful sizes. This occurs, for example, in a text editor where the scrollbars, menus, and other adornments are fixed in size but the edit widget can support any number of lines of text or characters per line. In this case, it is usually desirable to let the user specify the number of lines or characters−per−line, either with the **geometry** method 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 **wmGrid** method 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 **grid** with empty argument strings.

When gridded geometry management is enabled then all the dimensions specified in **minsize**, **maxsize**, and **geometry** methods are treated as grid units rather than pixel units. Interactive resizing is also carried out in even numbers of grid units rather than pixels.

BUGS

Most existing window managers appear to have bugs that affect the operation of the **wm** methods. 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.

SEE ALSO

Tk::Widget|Tk::Widget Tk::tixWm|Tk::tixWm Tk::Mwm|Tk::Mwm

KEYWORDS

aspect ratio, deiconify, focus model, geometry, grid, group, icon, iconify, increments, position, size, title, top−level window, units, window manager

NAME

Tk::X – Perl extension for Xlib constants.

=for pm Xlib/X/X.pm

=for category Other Modules and Languages

SYNOPSIS

```
use Tk::X;
```

DESCRIPTION

A module generated by h2xs. It exists to export Xlib #define type constants for possible use with Tk::Xlib.

Exported constants

```
Above
AllTemporary
AllocAll
AllocNone
AllowExposures
AlreadyGrabbed
Always
AnyButton
AnyKey
AnyModifier
AnyPropertyType
ArcChord
ArcPieSlice
AsyncBoth
AsyncKeyboard
AsyncPointer
AutoRepeatModeDefault
AutoRepeatModeOff
AutoRepeatModeOn
BadAccess
BadAlloc
BadAtom
BadColor
BadCursor
BadDrawable
BadFont
BadGC
BadIDChoice
BadImplementation
BadLength
BadMatch
BadName
BadPixmap
BadRequest
BadValue
BadWindow
Below
BottomIf
Button1
Button1Mask
Button1MotionMask
Button2
```

```
Button2Mask
Button2MotionMask
Button3
Button3Mask
Button3MotionMask
Button4
Button4Mask
Button4MotionMask
Button5
Button5Mask
Button5MotionMask
ButtonMotionMask
ButtonPress
ButtonPressMask
ButtonRelease
ButtonReleaseMask
CWBackPixel
CWBackPixmap
CWBackingPixel
CWBackingPlanes
CWBackingStore
CWBitGravity
CWBorderPixel
CWBorderPixmap
CWBorderWidth
CWColormap
CWCursor
CWDontPropagate
CWEventMask
CWHeight
CWOverrideRedirect
CWSaveUnder
CWSibling
CWStackMode
CWWidth
CWWinGravity
CWX
CWY
CapButt
CapNotLast
CapProjecting
CapRound
CenterGravity
CirculateNotify
CirculateRequest
ClientMessage
ClipByChildren
ColormapChangeMask
ColormapInstalled
ColormapNotify
ColormapUninstalled
Complex
ConfigureNotify
ConfigureRequest
```

```
ControlMapIndex
ControlMask
Convex
CoordModeOrigin
CoordModePrevious
CopyFromParent
CreateNotify
CurrentTime
CursorShape
DefaultBlanking
DefaultExposures
DestroyAll
DestroyNotify
DirectColor
DisableAccess
DisableScreenInterval
DisableScreenSaver
DoBlue
DoGreen
DoRed
DontAllowExposures
DontPreferBlanking
EastGravity
EnableAccess
EnterNotify
EnterWindowMask
EvenOddRule
Expose
ExposureMask
FamilyChaos
FamilyDECnet
FamilyInternet
FillOpaqueStippled
FillSolid
FillStippled
FillTiled
FirstExtensionError
FocusChangeMask
FocusIn
FocusOut
FontChange
FontLeftToRight
FontRightToLeft
ForgetGravity
GCArcMode
GCBackground
GCCapStyle
GCClipMask
GCClipXOrigin
GCClipYOrigin
GCDashList
GCDashOffset
GCFillRule
GCFillStyle
```

```
GCFont
GCForeground
GCFunction
GCGraphicsExposures
GCJoinStyle
GCLastBit
GCLineStyle
GCLineWidth
GCPlaneMask
GCStipple
GCSubwindowMode
GCTile
GCTileStipXOrigin
GCTileStipYOrigin
GXand
GXandInverted
GXandReverse
GXclear
GXcopy
GXcopyInverted
GXequiv
GXinvert
GXnand
GXnoop
GXnor
GXor
GXorInverted
GXorReverse
GXset
GXxor
GrabFrozen
GrabInvalidTime
GrabModeAsync
GrabModeSync
GrabNotViewable
GrabSuccess
GraphicsExpose
GravityNotify
GrayScale
HostDelete
HostInsert
IncludeInferiors
InputFocus
InputOnly
InputOutput
IsUnmapped
IsUnviewable
IsViewable
JoinBevel
JoinMiter
JoinRound
KBAutoRepeatMode
KBBellDuration
KBBellPercent
```

```
KBBellPitch
KBKey
KBKeyClickPercent
KBLed
KBLedMode
KeyPress
KeyPressMask
KeyRelease
KeyReleaseMask
KeymapNotify
KeymapStateMask
LASTEvent
LSBFirst
LastExtensionError
LeaveNotify
LeaveWindowMask
LedModeOff
LedModeOn
LineDoubleDash
LineOnOffDash
LineSolid
LockMapIndex
LockMask
LowerHighest
MSBFirst
MapNotify
MapRequest
MappingBusy
MappingFailed
MappingKeyboard
MappingModifier
MappingNotify
MappingPointer
MappingSuccess
Mod1MapIndex
Mod1Mask
Mod2MapIndex
Mod2Mask
Mod3MapIndex
Mod3Mask
Mod4MapIndex
Mod4Mask
Mod5MapIndex
Mod5Mask
MotionNotify
NoEventMask
NoExpose
NoSymbol
Nonconvex
None
NorthEastGravity
NorthGravity
NorthWestGravity
NotUseful
```

```
NotifyAncestor
NotifyDetailNone
NotifyGrab
NotifyHint
NotifyInferior
NotifyNonlinear
NotifyNonlinearVirtual
NotifyNormal
NotifyPointer
NotifyPointerRoot
NotifyUngrab
NotifyVirtual
NotifyWhileGrabbed
Opposite
OwnerGrabButtonMask
ParentRelative
PlaceOnBottom
PlaceOnTop
PointerMotionHintMask
PointerMotionMask
PointerRoot
PointerWindow
PreferBlanking
PropModeAppend
PropModePrepend
PropModeReplace
PropertyChangeMask
PropertyDelete
PropertyNewValue
PropertyNotify
PseudoColor
RaiseLowest
ReparentNotify
ReplayKeyboard
ReplayPointer
ResizeRedirectMask
ResizeRequest
RetainPermanent
RetainTemporary
RevertToNone
RevertToParent
RevertToPointerRoot
ScreenSaverActive
ScreenSaverReset
SelectionClear
SelectionNotify
SelectionRequest
SetModeDelete
SetModeInsert
ShiftMapIndex
ShiftMask
SouthEastGravity
SouthGravity
SouthWestGravity
```

```
StaticColor
StaticGravity
StaticGray
StippleShape
StructureNotifyMask
SubstructureNotifyMask
SubstructureRedirectMask
Success
SyncBoth
SyncKeyboard
SyncPointer
TileShape
TopIf
TrueColor
UnmapGravity
UnmapNotify
Unsorted
VisibilityChangeMask
VisibilityFullyObscured
VisibilityNotify
VisibilityPartiallyObscured
VisibilityUnobscured
WestGravity
WhenMapped
WindingRule
XYBitmap
XYPixmap
X_H
X_PROTOCOL
X_PROTOCOL_REVISION
YSorted
YXBanded
YXSorted
ZPixmap
```

AUTHOR

Nick Ing−Simmons ran h2xs ...

NAME

Tk::Font – a class for finding X Fonts

=for pm Tk/X11Font.pm

=for category Tk Generic Methods

SYNOPSIS

```
use Tk::X11Font;

$font = $widget->X11Font(foundry => 'adobe',
                         family  => 'times',
                         point   => 120
                        );

$font = $widget->X11Font('*-courier-medium-r-normal-*-*');
```

DESCRIPTION

This module can be use to interrogate the X server what fonts are avaliable.

METHODS

Foundry([`$val`])

Family([`$val`])

Weight([`$val`])

Slant([`$val`])

Swidth([`$val`])

Adstyle([`$val`])

Pixel([`$val`])

Point([`$val`])

Xres([`$val`])

Yres([`$val`])

Space([`$val`])

Avgwidth([`$val`])

Registry([`$val`])

Encoding([`$val`])

Set the given field in the font name to `$val` if given and return the current or previous value

Name([`$max`])

In a list context it returns a list of all font names that match the fields given. It will return a maximum of `$max` names, or 128 if `$max` is not given.

In a scalar contex it returns the first matching name or undef

Clone([key = value, [...]])

Create a duplicate of the curent font object and modify the given fields

AUTHOR

Graham Barr <Graham.Barr@tiuk.ti.com

HISTORY

11−Jan−96 Initial version

08−Nov−98 Renamed for Tk800.012

COPYRIGHT

NAME

Tk::Xrm – X Resource/Defaults/Options routines that obey the rules.

=for pm Tk/Xrm.pm

=for category Creating and Configuring Widgets

SYNOPSIS

```
use Tk;
use Tk::Xrm;
```

DESCRIPTION

Using this modules causes Tk's Option code to be replaced by versions which use routines from <X11/Xresource.h – i.e. same ones every other X toolkit uses.

Result is that "matching" of name/Class with the options database follows the same rules as other X toolkits. This makes it more predictable, and makes it easier to have a single ~/.Xdefaults file which gives sensible results for both Tk and (say) Motif applications.

BUGS

Currently **optionAdd**(*key* => *value*?, *priority*?) ignores optional priority completely and just does `XrmPutStringResource()`. Perhaps it should be more subtle and do `XrmMergeDatabases()` or `XrmCombineDatabase()`.

This version is a little slower than Tk's re–invention but there is more optimization that can be done.

SEE ALSO

Tk::option|Tk::option

KEYWORDS

database, option, priority, retrieve

NAME

Tk_Get3DBorder, Tk_Draw3DRectangle, Tk_Fill3DRectangle, Tk_Draw3DPolygon, Tk_Fill3DPolygon, Tk_3DVerticalBevel, Tk_3DHorizontalBevel, Tk_SetBackgroundFromBorder, Tk_NameOf3DBorder, Tk_3DBorderColor, Tk_3DBorderGC, Tk_Free3DBorder − draw borders with three−dimensional appearance

=for category C Programming

SYNOPSIS

#include <tk.h>

Tk_3DBorder **Tk_Get3DBorder(***interp, tkwin, colorName***)**

void **Tk_Draw3DRectangle(***tkwin, drawable, border, x, y, width, height, borderWidth, relief***)**

void **Tk_Fill3DRectangle(***tkwin, drawable, border, x, y, width, height, borderWidth, relief***)**

void **Tk_Draw3DPolygon(***tkwin, drawable, border, pointPtr, numPoints, polyBorderWidth, leftRelief***)**

void **Tk_Fill3DPolygon(***tkwin, drawable, border, pointPtr, numPoints, polyBorderWidth, leftRelief***)**

void **Tk_3DVerticalBevel(***tkwin, drawable, border, x, y, width, height, leftBevel, relief***)**

void **Tk_3DHorizontalBevel(***tkwin, drawable, border, x, y, width, height, leftIn, rightIn, topBevel, relief***)**

void **Tk_SetBackgroundFromBorder(***tkwin, border***)**

char * **Tk_NameOf3DBorder(***border***)**

XColor * **Tk_3DBorderColor(***border***)**

GC * **Tk_3DBorderGC(***tkwin, border, which***)**

Tk_Free3DBorder(*border***)**

ARGUMENTS

Tcl_Interp *interp (in)

Interpreter to use for error reporting.

Tk_Window tkwin (in)

Token for window (for all procedures except **Tk_Get3DBorder**, must be the window for which the border was allocated).

Tk_Uid colorName (in)

Textual description of color corresponding to background (flat areas). Illuminated edges will be brighter than this and shadowed edges will be darker than this.

Drawable drawable (in)

X token for window or pixmap; indicates where graphics are to be drawn. Must either be the X window for *tkwin* or a pixmap with the same screen and depth as *tkwin*.

Tk_3DBorder border (in)

Token for border previously allocated in call to **Tk_Get3DBorder**.

int x (in)

X−coordinate of upper−left corner of rectangle describing border or bevel, in pixels.

int y (in)

Y−coordinate of upper−left corner of rectangle describing border or bevel, in pixels.

int width (in)

Width of rectangle describing border or bevel, in pixels.

int height (in)

Height of rectangle describing border or bevel, in pixels.

int borderWidth (in)

Width of border in pixels. Positive means border is inside rectangle given by *x*, *y*, *width*, *height*, negative means border is outside rectangle.

int relief (in)

Indicates 3−D position of interior of object relative to exterior; should be TK_RELIEF_RAISED, TK_RELIEF_SUNKEN, TK_RELIEF_GROOVE, TK_RELIEF_SOLID, or TK_RELIEF_RIDGE (may also be TK_RELIEF_FLAT for **Tk_Fill3DRectangle**).

XPoint *pointPtr (in)

Pointer to array of points describing the set of vertices in a polygon. The polygon need not be closed (it will be closed automatically if it isn't).

int numPoints (in)

Number of points at **pointPtr*.

int polyBorderWidth (in)

Width of border in pixels. If positive, border is drawn to left of trajectory given by *pointPtr*; if negative, border is drawn to right of trajectory. If *leftRelief* is TK_RELIEF_GROOVE or TK_RELIEF_RIDGE then the border is centered on the trajectory.

int leftRelief (in)

Height of left side of polygon's path relative to right. TK_RELIEF_RAISED means left side should appear higher and TK_RELIEF_SUNKEN means right side should appear higher;
TK_RELIEF_GROOVE and TK_RELIEF_RIDGE mean the obvious things. For **Tk_Fill3DPolygon**, TK_RELIEF_FLAT may also be specified to indicate no difference in height.

int leftBevel (in)

Non−zero means this bevel forms the left side of the object; zero means it forms the right side.

int leftIn (in)

Non−zero means that the left edge of the horizontal bevel angles in, so that the bottom of the edge is farther to the right than the top. Zero means the edge angles out, so that the bottom is farther to the left than the top.

int rightIn (in)

Non−zero means that the right edge of the horizontal bevel angles in, so that the bottom of the edge is farther to the left than the top. Zero means the edge angles out, so that the bottom is farther to the right than the top.

int topBevel (in)

Non−zero means this bevel forms the top side of the object; zero means it forms the bottom side.

int which (in)

Specifies which of the border's graphics contexts is desired. Must be TK_3D_FLAT_GC, TK_3D_LIGHT_GC, or TK_3D_DARK_GC.

DESCRIPTION

These procedures provide facilities for drawing window borders in a way that produces a three−dimensional appearance. **Tk_Get3DBorder** allocates colors and Pixmaps needed to draw a border in the window given by the *tkwin* argument. The *colorName* argument indicates what colors should be used in the border. *ColorName* may be any value acceptable to **Tk_GetColor**. The color indicated by *colorName* will not actually be used in the border; it indicates the background color for the window (i.e. a color for flat surfaces). The illuminated portions of the border will appear brighter than indicated by *colorName*, and the

shadowed portions of the border will appear darker than *colorName*.

Tk_Get3DBorder returns a token that may be used in later calls to **Tk_Draw3DRectangle**. If an error occurs in allocating information for the border (e.g. *colorName* isn't a legal color specifier), then NULL is returned and an error message is left in *interp->result*.

Once a border structure has been created, **Tk_Draw3DRectangle** may be invoked to draw the border. The *tkwin* argument specifies the window for which the border was allocated, and *drawable* specifies a window or pixmap in which the border is to be drawn. *Drawable* need not refer to the same window as *tkwin*, but it must refer to a compatible pixmap or window: one associated with the same screen and with the same depth as *tkwin*. The *x*, *y*, *width*, and *height* arguments define the bounding box of the border region within *drawable* (usually *x* and *y* are zero and *width* and *height* are the dimensions of the window), and *borderWidth* specifies the number of pixels actually occupied by the border. The *relief* argument indicates which of several three-dimensional effects is desired: TK_RELIEF_RAISED means that the interior of the rectangle should appear raised relative to the exterior of the rectangle, and TK_RELIEF_SUNKEN means that the interior should appear depressed. TK_RELIEF_GROOVE and TK_RELIEF_RIDGE mean that there should appear to be a groove or ridge around the exterior of the rectangle.

Tk_Fill3DRectangle is somewhat like **Tk_Draw3DRectangle** except that it first fills the rectangular area with the background color (one corresponding to the *colorName* used to create *border*). Then it calls **Tk_Draw3DRectangle** to draw a border just inside the outer edge of the rectangular area. The argument *relief* indicates the desired effect (TK_RELIEF_FLAT means no border should be drawn; all that happens is to fill the rectangle with the background color).

The procedure **Tk_Draw3DPolygon** may be used to draw more complex shapes with a three-dimensional appearance. The *pointPtr* and *numPoints* arguments define a trajectory, *polyBorderWidth* indicates how wide the border should be (and on which side of the trajectory to draw it), and *leftRelief* indicates which side of the trajectory should appear raised. **Tk_Draw3DPolygon** draws a border around the given trajectory using the colors from *border* to produce a three-dimensional appearance. If the trajectory is non-self-intersecting, the appearance will be a raised or sunken polygon shape. The trajectory may be self-intersecting, although it's not clear how useful this is.

Tk_Fill3DPolygon is to **Tk_Draw3DPolygon** what **Tk_Fill3DRectangle** is to **Tk_Draw3DRectangle**: it fills the polygonal area with the background color from *border*, then calls **Tk_Draw3DPolygon** to draw a border around the area (unless *leftRelief* is TK_RELIEF_FLAT; in this case no border is drawn).

The procedures **Tk_3DVerticalBevel** and **Tk_3DHorizontalBevel** provide lower-level drawing primitives that are used by procedures such as **Tk_Draw3DRectangle**. These procedures are also useful in their own right for drawing rectilinear border shapes. **Tk_3DVerticalBevel** draws a vertical beveled edge, such as the left or right side of a rectangle, and **Tk_3DHorizontalBevel** draws a horizontal beveled edge, such as the top or bottom of a rectangle. Each procedure takes *x*, *y*, *width*, and *height* arguments that describe the rectangular area of the beveled edge (e.g., *width* is the border width for **Tk_3DVerticalBevel**). The *leftBorder* and *topBorder* arguments indicate the position of the border relative to the ''inside'' of the object, and *relief* indicates the relief of the inside of the object relative to the outside. **Tk_3DVerticalBevel** just draws a rectangular region. **Tk_3DHorizontalBevel** draws a trapezoidal region to generate mitered corners; it should be called after **Tk_3DVerticalBevel** (otherwise **Tk_3DVerticalBevel** will overwrite the mitering in the corner). The *leftIn* and *rightIn* arguments to **Tk_3DHorizontalBevel** describe the mitering at the corners; a value of 1 means that the bottom edge of the trapezoid will be shorter than the top, 0 means it will be longer. For example, to draw a rectangular border the top bevel should be drawn with 1 for both *leftIn* and *rightIn*, and the bottom bevel should be drawn with 0 for both arguments.

The procedure **Tk_SetBackgroundFromBorder** will modify the background pixel and/or pixmap of *tkwin* to produce a result compatible with *border*. For color displays, the resulting background will just be the color given by the *colorName* argument passed to **Tk_Get3DBorder** when *border* was created; for monochrome displays, the resulting background will be a light stipple pattern, in order to distinguish the background from the illuminated portion of the border.

Given a token for a border, the procedure **Tk_NameOf3DBorder** will return the *colorName* string that was

passed to **Tk_Get3DBorder** to create the border.

The procedure **Tk_3DBorderColor** returns the XColor structure that will be used for flat surfaces drawn for its *border* argument by procedures like **Tk_Fill3DRectangle**. The return value corresponds to the *colorName* passed to **Tk_Get3DBorder**. The XColor, and its associated pixel value, will remain allocated as long as *border* exists.

The procedure **Tk_3DBorderGC** returns one of the X graphics contexts that are used to draw the border. The argument *which* selects which one of the three possible GC's: TK_3D_FLAT_GC returns the context used for flat surfaces, TK_3D_LIGHT_GC returns the context for light shadows, and TK_3D_DARK_GC returns the context for dark shadows.

When a border is no longer needed, **Tk_Free3DBorder** should be called to release the resources associated with the border. There should be exactly one call to **Tk_Free3DBorder** for each call to **Tk_Get3DBorder**.

KEYWORDS

3D, background, border, color, depressed, illumination, polygon, raised, shadow, three−dimensional effect

NAME

Tk_BackgroundError − report Tcl error that occurred in background processing

=for category C Programming

SYNOPSIS

#include <tk.h>

Tk_BackgroundError(*interp*)

ARGUMENTS

Tcl_Interp *interp (in)

Interpreter in which the error occurred.

DESCRIPTION

This procedure is typically invoked when a Tcl error occurs during ‘‘background processing‘’ such as executing a Tcl command in response to a button press or menu entry invocation. When such an error occurs, the error condition is reported to Tk or to a widget or some other C code, and there is not usually any obvious way for that code to report the error to the user. In these cases the code calls **Tk_BackgroundError** with an *interp* argument identifying the interpreter in which the error occurred. **Tk_BackgroundError** attempts to invoke the **tkerror** Tcl command to report the error in an application−specific fashion. If no **tkerror** command exists, or if it returns with an error condition, then **Tk_BackgroundError** reports the error itself by printing a message on the standard error file.

Tk_BackgroundError does not invoke **tkerror** immediately (in some cases this could interfere with scripts that are in process at the time the error occurred). Instead, it invokes **tkerror** later as an idle callback. **Tk_BackgroundError** saves the values of the **errorInfo** and **errorCode** variables and restores these values just before invoking **tkerror**.

It is possible for many background errors to accumulate before **tkerror** is invoked. When this happens, each of the errors is processed in order. However, if **tkerror** returns a break exception, then all remaining error reports for the interpreter are skipped.

KEYWORDS

background, error, tkerror

NAME

Tk_CreateBindingTable, Tk_DeleteBindingTable, Tk_CreateBinding, Tk_DeleteBinding, Tk_GetBinding, Tk_GetAllBindings, Tk_DeleteAllBindings, Tk_BindEvent − invoke scripts in response to X events

=for category C Programming

SYNOPSIS

#include <tk.h>

Tk_BindingTable **Tk_CreateBindingTable(***interp***)**

Tk_DeleteBindingTable(*bindingTable***)**

unsigned long **Tk_CreateBinding(***interp, bindingTable, object, eventString, script, append***)**

int **Tk_DeleteBinding(***interp, bindingTable, object, eventString***)**

char * **Tk_GetBinding(***interp, bindingTable, object, eventString***)**

Tk_GetAllBindings(*interp, bindingTable, object***)**

Tk_DeleteAllBindings(*bindingTable, object***)**

Tk_BindEvent(*bindingTable, eventPtr, tkwin, numObjects, objectPtr***)**

ARGUMENTS

Tcl_Interp *interp (in)

Interpreter to use when invoking bindings in binding table. Also used for returning results and errors from binding procedures.

Tk_BindingTable bindingTable (in)

Token for binding table; must have been returned by some previous call to **Tk_CreateBindingTable**.

ClientData object (in)

Identifies object with which binding is associated.

char *eventString (in)

String describing event sequence.

char *script (in)

Callback to invoke when binding triggers.

int append (in)

Non−zero means append *script* to existing script for binding, if any; zero means replace existing script with new one.

XEvent *eventPtr (in)

X event to match against bindings in *bindingTable*.

Tk_Window tkwin (in)

Identifier for any window on the display where the event occurred. Used to find display−related information such as key maps.

int numObjects (in)

Number of object identifiers pointed to by *objectPtr*.

ClientData *objectPtr (in)

Points to an array of object identifiers: bindings will be considered for each of these objects in order from first to last.

DESCRIPTION

These procedures provide a general−purpose mechanism for creating and invoking bindings. Bindings are organized in terms of *binding tables*. A binding table consists of a collection of bindings plus a history of recent events. Within a binding table, bindings are associated with *objects*. The meaning of an object is defined by clients of the binding package. For example, Tk keeps uses one binding table to hold all of the bindings created by the **bind** command. For this table, objects are pointers to strings such as window names, class names, or other binding tags such as **all**. Tk also keeps a separate binding table for each canvas widget, which manages bindings created by the canvas's **bind** method; within this table, an object is either a pointer to the internal structure for a canvas item or a Tk_Uid identifying a tag.

The procedure **Tk_CreateBindingTable** creates a new binding table and associates *interp* with it (when bindings in the table are invoked, the scripts will be evaluated in *interp*). **Tk_CreateBindingTable** returns a token for the table, which must be used in calls to other procedures such as **Tk_CreateBinding** or **Tk_BindEvent**.

Tk_DeleteBindingTable frees all of the state associated with a binding table. Once it returns the caller should not use the *bindingTable* token again.

Tk_CreateBinding adds a new binding to an existing table. The *object* argument identifies the object with which the binding is to be associated, and it may be any one−word value. Typically it is a pointer to a string or data structure. The *eventString* argument identifies the event or sequence of events for the binding; see the documentation for the **bind** command for a description of its format. *script* is the Callback to be evaluated when the binding triggers. *append* indicates what to do if there already exists a binding for *object* and *eventString*: if *append* is zero then *script* replaces the old script; if *append* is non−zero then the new script is appended to the old one. **Tk_CreateBinding** returns an X event mask for all the events associated with the bindings. This information may be useful to invoke **XSelectInput** to select relevant events, or to disallow the use of certain events in bindings. If an error occurred while creating the binding (e.g., *eventString* refers to a non−existent event), then 0 is returned and an error message is left in *interp−>result*.

Tk_DeleteBinding removes from *bindingTable* the binding given by *object* and *eventString*, if such a binding exists. **Tk_DeleteBinding** always returns TCL_OK. In some cases it may reset *interp−>result* to the default empty value.

Tk_GetBinding returns a pointer to the script associated with *eventString* and *object* in *bindingTable*. If no such binding exists then NULL is returned and an error message is left in *interp−>result*.

Tk_GetAllBindings returns in *interp−>result* a list of all the event strings for which there are bindings in *bindingTable* associated with *object*. If there are no bindings for *object* then an empty string is returned in *interp−>result*.

Tk_DeleteAllBindings deletes all of the bindings in *bindingTable* that are associated with *object*.

Tk_BindEvent is called to process an event. It makes a copy of the event in an internal history list associated with the binding table, then it checks for bindings that match the event. **Tk_BindEvent** processes each of the objects pointed to by *objectPtr* in turn. For each object, it finds all the bindings that match the current event history, selects the most specific binding using the priority mechanism described in the documentation for **bind**, and invokes the script for that binding. If there are no matching bindings for a particular object, then the object is skipped. **Tk_BindEvent** continues through all of the objects, handling exceptions such as errors, **break**, and **continue** as described in the documentation for **bind**.

KEYWORDS

binding, event, object, script

NAME

Tk_CanvasPsY, Tk_CanvasPsBitmap, Tk_CanvasPsColor, Tk_CanvasPsFont, Tk_CanvasPsPath, Tk_CanvasPsStipple − utility procedures for generating Postscript for canvases

=for category C Programming

SYNOPSIS

#include <tk.h>

double **Tk_CanvasPsY**(*canvas, canvasY*)

int **Tk_CanvasPsBitmap**(*interp, canvas, bitmap, x, y, width, height*)

int **Tk_CanvasPsColor**(*interp, canvas, colorPtr*)

int **Tk_CanvasPsFont**(*interp, canvas, fontStructPtr*)

Tk_CanvasPsPath(*interp, canvas, coordPtr, numPoints*)

int **Tk_CanvasPsStipple**(*interp, canvas, bitmap*)

ARGUMENTS

Tk_Canvas canvas (in)

A token that identifies a canvas widget for which Postscript is being generated.

double canvasY (in)

Y−coordinate in the space of the canvas.

Tcl_Interp *interp (in/out)

A Tcl interpreter; Postscript is appended to its result, or the result may be replaced with an error message.

Pixmap bitmap (in)

Bitmap to use for generating Postscript.

int x (in)

X−coordinate within *bitmap* of left edge of region to output.

int y (in)

Y−coordinate within *bitmap* of top edge of region to output.

"int" width (in)

Width of region of bitmap to output, in pixels.

"int" height (in)

Height of region of bitmap to output, in pixels.

XColor *colorPtr (in)

Information about color value to set in Postscript.

XFontStruct *fontStructPtr (in)

Font for which Postscript is to be generated.

double *coordPtr (in)

Pointer to an array of coordinates for one or more points specified in canvas coordinates. The order of values in *coordPtr* is x1, y1, x2, y2, x3, y3, and so on.

int numPoints (in)

Number of points at *coordPtr*.

DESCRIPTION

These procedures are called by canvas type managers to carry out common functions related to generating Postscript. Most of the procedures take a *canvas* argument, which refers to a canvas widget for which Postscript is being generated.

Tk_CanvasY takes as argument a y−coordinate in the space of a canvas and returns the value that should be used for that point in the Postscript currently being generated for *canvas*. Y coordinates require transformation because Postscript uses an origin at the lower−left corner whereas X uses an origin at the upper−left corner. Canvas x coordinates can be used directly in Postscript without transformation.

Tk_CanvasPsBitmap generates Postscript to describe a region of a bitmap. The Postscript is generated in proper image data format for Postscript, i.e., as data between angle brackets, one bit per pixel. The Postscript is appended to *interp−>result* and TCL_OK is returned unless an error occurs, in which case TCL_ERROR is returned and *interp−>result* is overwritten with an error message.

Tk_CanvasPsColor generates Postscript to set the current color to correspond to its *colorPtr* argument, taking into account any color map specified in the **postscript** command. It appends the Postscript to *interp−>result* and returns TCL_OK unless an error occurs, in which case TCL_ERROR is returned and *interp−>result* is overwritten with an error message.

Tk_CanvasPsFont generates Postscript that sets the current font to match *fontStructPtr* as closely as possible. **Tk_CanvasPsFont** takes into account any font map specified in the **postscript** command, and it does the best it can at mapping X fonts to Postscript fonts. It appends the Postscript to *interp−>result* and returns TCL_OK unless an error occurs, in which case TCL_ERROR is returned and *interp−>result* is overwritten with an error message.

Tk_CanvasPsPath generates Postscript to set the current path to the set of points given by *coordPtr* and *numPoints*. It appends the resulting Postscript to *interp−>result*.

Tk_CanvasPsStipple generates Postscript that will fill the current path in stippled fashion. It uses *bitmap* as the stipple pattern and the current Postscript color; ones in the stipple bitmap are drawn in the current color, and zeroes are not drawn at all. The Postscript is appended to *interp−>result* and TCL_OK is returned, unless an error occurs, in which case TCL_ERROR is returned and *interp−>result* is overwritten with an error message.

KEYWORDS

bitmap, canvas, color, font, path, Postscript, stipple

NAME

Tk_CanvasTkwin, Tk_CanvasGetCoord, Tk_CanvasDrawableCoords, Tk_CanvasSetStippleOrigin, Tk_CanvasWindowCoords, Tk_CanvasEventuallyRedraw, Tk_CanvasTagsOption − utility procedures for canvas type managers

=for category C Programming

SYNOPSIS

#include <tk.h>

Tk_Window **Tk_CanvasTkwin**(*canvas*)

int **Tk_CanvasGetCoord**(*interp, canvas, string, doublePtr*)

Tk_CanvasDrawableCoords(*canvas, x, y, drawableXPtr, drawableYPtr*)

Tk_CanvasSetStippleOrigin(*canvas, gc*)

Tk_CanvasWindowCoords(*canvas, x, y, screenXPtr, screenYPtr*)

Tk_CanvasEventuallyRedraw(*canvas, x1, y1, x2, y2*)

Tk_OptionParseProc ***Tk_CanvasTagsParseProc**;

Tk_OptionPrintProc ***Tk_CanvasTagsPrintProc**;

ARGUMENTS

Tk_Canvas canvas (in)

A token that identifies a canvas widget.

Tcl_Interp *interp (in/out)

Interpreter to use for error reporting.

char *string (in)

Textual description of a canvas coordinate.

double *doublePtr (out)

Points to place to store a converted coordinate.

double x (in)

An x coordinate in the space of the canvas.

double y (in)

A y coordinate in the space of the canvas.

short *drawableXPtr (out)

Pointer to a location in which to store an x coordinate in the space of the drawable currently being used to redisplay the canvas.

short *drawableYPtr (out)

Pointer to a location in which to store a y coordinate in the space of the drawable currently being used to redisplay the canvas.

GC gc (out)

Graphics context to modify.

short *screenXPtr (out)

Points to a location in which to store the screen coordinate in the canvas window that corresponds to *x*.

short *screenYPtr (out)

Points to a location in which to store the screen coordinate in the canvas window that corresponds to *y*.

int x1 (in)

Left edge of the region that needs redisplay. Only pixels at or to the right of this coordinate need to be redisplayed.

int y1 (in)

Top edge of the region that needs redisplay. Only pixels at or below this coordinate need to be redisplayed.

int x2 (in)

Right edge of the region that needs redisplay. Only pixels to the left of this coordinate need to be redisplayed.

int y2 (in)

Bottom edge of the region that needs redisplay. Only pixels above this coordinate need to be redisplayed.

DESCRIPTION

These procedures are called by canvas type managers to perform various utility functions.

Tk_CanvasTkwin returns the Tk_Window associated with a particular canvas.

Tk_CanvasGetCoord translates a string specification of a coordinate (such as **2p** or **1.6c**) into a double−precision canvas coordinate. If *string* is a valid coordinate description then **Tk_CanvasGetCoord** stores the corresponding canvas coordinate at **doublePtr* and returns TCL_OK. Otherwise it stores an error message in *interp−>result* and returns TCL_ERROR.

Tk_CanvasDrawableCoords is called by type managers during redisplay to compute where to draw things. Given *x* and *y* coordinates in the space of the canvas, **Tk_CanvasDrawableCoords** computes the corresponding pixel in the drawable that is currently being used for redisplay; it returns those coordinates in **drawableXPtr* and **drawableYPtr*. This procedure should not be invoked except during redisplay.

Tk_CanvasSetStippleOrigin is also used during redisplay. It sets the stipple origin in *gc* so that stipples drawn with *gc* in the current offscreen pixmap will line up with stipples drawn with origin (0,0) in the canvas's actual window. **Tk_CanvasSetStippleOrigin** is needed in order to guarantee that stipple patterns line up properly when the canvas is redisplayed in small pieces. Redisplays are carried out in double−buffered fashion where a piece of the canvas is redrawn in an offscreen pixmap and then copied back onto the screen. In this approach the stipple origins in graphics contexts need to be adjusted during each redisplay to compensate for the position of the off−screen pixmap relative to the window. If an item is being drawn with stipples, its type manager typically calls **Tk_CanvasSetStippleOrigin** just before using *gc* to draw something; after it is finished drawing, the type manager calls **XSetTSOrigin** to restore the origin in *gc* back to (0,0) (the restore is needed because graphics contexts are shared, so they cannot be modified permanently).

Tk_CanvasWindowCoords is similar to **Tk_CanvasDrawableCoords** except that it returns coordinates in the canvas's window on the screen, instead of coordinates in an off−screen pixmap.

Tk_CanvasEventuallyRedraw may be invoked by a type manager to inform Tk that a portion of a canvas needs to be redrawn. The *x1*, *y1*, *x2*, and *y2* arguments specify the region that needs to be redrawn, in canvas coordinates. Type managers rarely need to invoke **Tk_CanvasEventuallyRedraw**, since Tk can normally figure out when an item has changed and make the redisplay request on its behalf (this happens, for example whenever Tk calls a *configureProc* or *scaleProc*). The only time that a type manager needs to call **Tk_CanvasEventuallyRedraw** is if an item has changed on its own without being invoked through one of the procedures in its Tk_ItemType; this could happen, for example, in an image item if the image is modified using image commands.

Tk_CanvasTagsParseProc and **Tk_CanvasTagsPrintProc** are procedures that handle the **–tags** option for canvas items. The code of a canvas type manager won't call these procedures directly, but will use their addresses to create a **Tk_CustomOption** structure for the **–tags** option. The code typically looks like this:

```
static Tk_CustomOption tagsOption = {Tk_CanvasTagsParseProc,
       Tk_CanvasTagsPrintProc, (ClientData) NULL
};

static Tk_ConfigSpec configSpecs[] = {
       ...
       {TK_CONFIG_CUSTOM, "-tags", (char *) NULL, (char *) NULL,
               (char *) NULL, 0, TK_CONFIG_NULL_OK, &tagsOption},
       ...
};
```

KEYWORDS

canvas, focus, item type, redisplay, selection, type manager

NAME

Tk_CanvasTextInfo – additional information for managing text items in canvases

=for category C Programming

SYNOPSIS

#include <tk.h>

Tk_CanvasTextInfo * **Tk_CanvasGetTextInfo**(*canvas*)

ARGUMENTS

Tk_Canvas canvas (in)

A token that identifies a particular canvas widget.

DESCRIPTION

Textual canvas items are somewhat more complicated to manage than other items, due to things like the selection and the input focus. **Tk_CanvasGetTextInfo** may be invoked by a type manager to obtain additional information needed for items that display text. The return value from **Tk_CanvasGetTextInfo** is a pointer to a structure that is shared between Tk and all the items that display text. The structure has the following form:

```
typedef struct Tk_CanvasTextInfo {
        Tk_3DBorder selBorder;
        int selBorderWidth;
        XColor *selFgColorPtr;
        Tk_Item *selItemPtr;
        int selectFirst;
        int selectLast;
        Tk_Item *anchorItemPtr;
        int selectAnchor;
        Tk_3DBorder insertBorder;
        int insertWidth;
        int insertBorderWidth;
        Tk_Item *focusItemPtr;
        int gotFocus;
        int cursorOn;
} Tk_CanvasTextInfo;
```

The **selBorder** field identifies a Tk_3DBorder that should be used for drawing the background under selected text. *selBorderWidth* gives the width of the raised border around selected text, in pixels. *selFgColorPtr* points to an XColor that describes the foreground color to be used when drawing selected text. *selItemPtr* points to the item that is currently selected, or NULL if there is no item selected or if the canvas doesn‘t have the selection. *selectFirst* and *selectLast* give the indices of the first and last selected characters in *selItemPtr*, as returned by the *indexProc* for that item. *anchorItemPtr* points to the item that currently has the selection anchor; this is not necessarily the same as *selItemPtr*. *selectAnchor* is an index that identifies the anchor position within *anchorItemPtr*. *insertBorder* contains a Tk_3DBorder to use when drawing the insertion cursor; *insertWidth* gives the total width of the insertion cursor in pixels, and *insertBorderWidth* gives the width of the raised border around the insertion cursor. *focusItemPtr* identifies the item that currently has the input focus, or NULL if there is no such item. *gotFocus* is 1 if the canvas widget has the input focus and 0 otherwise. *cursorOn* is 1 if the insertion cursor should be drawn in *focusItemPtr* and 0 if it should not be drawn; this field is toggled on and off by Tk to make the cursor blink.

The structure returned by **Tk_CanvasGetTextInfo** is shared between Tk and the type managers; typically the type manager calls **Tk_CanvasGetTextInfo** once when an item is created and then saves the pointer in the item‘s record. Tk will update information in the Tk_CanvasTextInfo; for example, a **configure** method might change the *selBorder* field, or a **select** method might change the *selectFirst* field, or Tk might change *cursorOn* in order to make the insertion cursor flash on and off during successive redisplays.

Type managers should treat all of the fields of the Tk_CanvasTextInfo structure as read–only, except for *selItemPtr*, *selectFirst*, *selectLast*, and *selectAnchor*. Type managers may change *selectFirst*, *selectLast*, and *selectAnchor* to adjust for insertions and deletions in the item (but only if the item is the current owner of the selection or anchor, as determined by *selItemPtr* or *anchorItemPtr*). If all of the selected text in the item is deleted, the item should set *selItemPtr* to NULL to indicate that there is no longer a selection.

KEYWORDS

canvas, focus, insertion cursor, selection, selection anchor, text

NAME

Tk_ClipboardClear, Tk_ClipboardAppend − Manage the clipboard

=for category C Programming

SYNOPSIS

#include <tk.h>

int **Tk_ClipboardClear**(*interp, tkwin*)

int **Tk_ClipboardAppend**(*interp, tkwin, target, format, buffer*)

ARGUMENTS

Tcl_Interp *interp (in)

Interpreter to use for reporting errors.

Tk_Window tkwin (in)

Window that determines which display's clipboard to manipulate.

Atom target (in)

Conversion type for this clipboard item; has same meaning as *target* argument to **Tk_CreateSelHandler**.

Atom format (in)

Representation to use when data is retrieved; has same meaning as *format* argument to **Tk_CreateSelHandler**.

char *buffer (in)

Null terminated string containing the data to be appended to the clipboard.

DESCRIPTION

These two procedures manage the clipboard for Tk. The clipboard is typically managed by calling **Tk_ClipboardClear** once, then calling **Tk_ClipboardAppend** to add data for any number of targets.

Tk_ClipboardClear claims the CLIPBOARD selection and frees any data items previously stored on the clipboard in this application. It normally returns TCL_OK, but if an error occurs it returns TCL_ERROR and leaves an error message in *interp−>result*. **Tk_ClipboardClear** must be called before a sequence of **Tk_ClipboardAppend** calls can be issued.

Tk_ClipboardAppend appends a buffer of data to the clipboard. The first buffer for a given *target* determines the *format* for that *target*. Any successive appends for that *target* must have the same format or an error will be returned. **Tk_ClipboardAppend** returns TCL_OK if the buffer is successfully copied onto the clipboard. If the clipboard is not currently owned by the application, either because **Tk_ClipboardClear** has not been called or because ownership of the clipboard has changed since the last call to **Tk_ClipboardClear**, **Tk_ClipboardAppend** returns TCL_ERROR and leaves an error message in *interp−>result*.

In order to guarantee atomicity, no event handling should occur between **Tk_ClipboardClear** and the following **Tk_ClipboardAppend** calls (otherwise someone could retrieve a partially completed clipboard or claim ownership away from this application).

Tk_ClipboardClear may invoke callbacks, including arbitrary Callbacks, as a result of losing the CLIPBOARD selection, so any calling function should take care to be reentrant at the point **Tk_ClipboardClear** is invoked.

KEYWORDS

append, clipboard, clear, format, type

NAME

Tk_ClearSelection − Deselect a selection

=for category C Programming

SYNOPSIS

#include <tk.h>

Tk_ClearSelection(*tkwin, selection*)

ARGUMENTS

Tk_Window tkwin (in)

The selection will be cleared from the display containing this window.

Atom selection (in)

The name of selection to be cleared.

DESCRIPTION

Tk_ClearSelection cancels the selection specified by the atom *selection* for the display containing *tkwin*. The selection need not be in *tkwin* itself or even in *tkwin*'s application. If there is a window anywhere on *tkwin*'s display that owns *selection*, the window will be notified and the selection will be cleared. If there is no owner for *selection* on the display, then the procedure has no effect.

KEYWORDS

clear, selection

NAME

Tk_ConfigureWidget, Tk_Offset, Tk_ConfigureInfo, Tk_ConfigureValue, Tk_FreeOptions − process configuration options for widgets

=for category C Programming

SYNOPSIS

#include <tk.h>

int **Tk_ConfigureWidget(***interp, tkwin, specs, argc, argv, widgRec, flags***)**

int **Tk_Offset(***type, field***)**

int **Tk_ConfigureInfo(***interp, tkwin, specs, widgRec, argvName, flags***)**

int

Tk_FreeOptions(*specs, widgRec, display, flags***)**

ARGUMENTS

Tcl_Interp *interp (in)

Interpreter to use for returning error messages.

Tk_Window tkwin (in)

Window used to represent widget (needed to set up X resources).

Tk_ConfigSpec *specs (in)

Pointer to table specifying legal configuration options for this widget.

int argc (in)

Number of arguments in *argv*.

char **argv (in)

Command−line options for configuring widget.

char *widgRec (in/out)

Points to widget record structure. Fields in this structure get modified by **Tk_ConfigureWidget** to hold configuration information.

int flags (in)

If non−zero, then it specifies an OR−ed combination of flags that control the processing of configuration information. TK_CONFIG_ARGV_ONLY causes the option database and defaults to be ignored, and flag bits TK_CONFIG_USER_BIT and higher are used to selectively disable entries in *specs*.

"type name" type (in)

The name of the type of a widget record.

"field name" field (in)

The name of a field in records of type *type*.

char *argvName (in)

The name used on Tcl command lines to refer to a particular option (e.g. when creating a widget or invoking the **configure** widget command). If non−NULL, then information is returned only for this option. If NULL, then information is returned for all available options.

Display *display (in)

Display containing widget whose record is being freed; needed in order to free up resources.

DESCRIPTION

Tk_ConfigureWidget is called to configure various aspects of a widget, such as colors, fonts, border width, etc. It is intended as a convenience procedure to reduce the amount of code that must be written in individual widget managers to handle configuration information. It is typically invoked when widgets are created, and again when the **configure** command is invoked for a widget. Although intended primarily for widgets, **Tk_ConfigureWidget** can be used in other situations where *argc−argv* information is to be used to fill in a record structure, such as configuring graphical elements for a canvas widget or entries of a menu.

Tk_ConfigureWidget processes a table specifying the configuration options that are supported (*specs*) and a collection of command−line arguments (*argc* and *argv*) to fill in fields of a record (*widgRec*). It uses the option database and defaults specified in *specs* to fill in fields of *widgRec* that are not specified in *argv*. **Tk_ConfigureWidget** normally returns the value TCL_OK; in this case it does not modify *interp*. If an error occurs then TCL_ERROR is returned and **Tk_ConfigureWidget** will leave an error message in *interp−>result* in the standard Tcl fashion. In the event of an error return, some of the fields of *widgRec* could already have been set, if configuration information for them was successfully processed before the error occurred. The other fields will be set to reasonable initial values so that **Tk_FreeOptions** can be called for cleanup.

The *specs* array specifies the kinds of configuration options expected by the widget. Each of its entries specifies one configuration option and has the following structure:

```
typedef struct {
        int type;
        char *argvName;
        char *dbName;
        char *dbClass;
        char *defValue;
        int offset;
        int specFlags;
        Tk_CustomOption *customPtr;
} Tk_ConfigSpec;
```

The *type* field indicates what type of configuration option this is (e.g. TK_CONFIG_COLOR for a color value, or TK_CONFIG_INT for an integer value). The *type* field indicates how to use the value of the option (more on this below). The *argvName* field is a string such as ‘‘−font’’ or ‘‘−bg’’, which is compared with the values in *argv* (if *argvName* is NULL it means this is a grouped entry; see */GROUPED ENTRIES* below). The *dbName* and *dbClass* fields are used to look up a value for this option in the option database. The *defValue* field specifies a default value for this configuration option if no value is specified in either *argv* or the option database. *Offset* indicates where in *widgRec* to store information about this option, and *specFlags* contains additional information to control the processing of this configuration option (see FLAGS below). The last field, *customPtr*, is only used if *type* is TK_CONFIG_CUSTOM; see */CUSTOM OPTION TYPES* below.

Tk_ConfigureWidget first processes *argv* to see which (if any) configuration options are specified there. *Argv* must contain an even number of fields; the first of each pair of fields must match the *argvName* of some entry in *specs* (unique abbreviations are acceptable), and the second field of the pair contains the value for that configuration option. If there are entries in *spec* for which there were no matching entries in *argv*, **Tk_ConfigureWidget** uses the *dbName* and *dbClass* fields of the *specs* entry to probe the option database; if a value is found, then it is used as the value for the option. Finally, if no entry is found in the option database, the *defValue* field of the *specs* entry is used as the value for the configuration option. If the *defValue* is NULL, or if the TK_CONFIG_DONT_SET_DEFAULT bit is set in *flags*, then there is no default value and this *specs* entry will be ignored if no value is specified in *argv* or the option database.

Once a string value has been determined for a configuration option, **Tk_ConfigureWidget** translates the string value into a more useful form, such as a color if *type* is TK_CONFIG_COLOR or an integer if *type* is TK_CONFIG_INT. This value is then stored in the record pointed to by *widgRec*. This record is assumed to contain information relevant to the manager of the widget; its exact type is unknown to

Tk_ConfigureWidget. The *offset* field of each *specs* entry indicates where in *widgRec* to store the information about this configuration option. You should use the **Tk_Offset** macro to generate *offset* values (see below for a description of **Tk_Offset**). The location indicated by *widgRec* and *offset* will be referred to as the ''target'' in the descriptions below.

The *type* field of each entry in *specs* determines what to do with the string value of that configuration option. The legal values for *type*, and the corresponding actions, are:

TK_CONFIG_ACTIVE_CURSOR

The value must be an ASCII string identifying a cursor in a form suitable for passing to **Tk_GetCursor**. The value is converted to a **Tk_Cursor** by calling **Tk_GetCursor** and the result is stored in the target. In addition, the resulting cursor is made the active cursor for *tkwin* by calling **XDefineCursor**. If TK_CONFIG_NULL_OK is specified in *specFlags* then the value may be an empty string, in which case the target and *tkwin*'s active cursor will be set to **None**. If the previous value of the target wasn't **None**, then it is freed by passing it to **Tk_FreeCursor**.

TK_CONFIG_ANCHOR

The value must be an ASCII string identifying an anchor point in one of the ways accepted by **Tk_GetAnchor**. The string is converted to a **Tk_Anchor** by calling **Tk_GetAnchor** and the result is stored in the target.

TK_CONFIG_BITMAP

The value must be an ASCII string identifying a bitmap in a form suitable for passing to **Tk_GetBitmap**. The value is converted to a **Pixmap** by calling **Tk_GetBitmap** and the result is stored in the target. If TK_CONFIG_NULL_OK is specified in *specFlags* then the value may be an empty string, in which case the target is set to **None**. If the previous value of the target wasn't **None**, then it is freed by passing it to **Tk_FreeBitmap**.

TK_CONFIG_BOOLEAN

The value must be an ASCII string specifying a boolean value. Any of the values ''true'', ''yes'', ''on'', or ''1'', or an abbreviation of one of these values, means true; any of the values ''false'', ''no'', ''off'', or ''0'', or an abbreviation of one of these values, means false. The target is expected to be an integer; for true values it will be set to 1 and for false values it will be set to 0.

TK_CONFIG_BORDER

The value must be an ASCII string identifying a border color in a form suitable for passing to **Tk_Get3DBorder**. The value is converted to a (**Tk_3DBorder ***) by calling **Tk_Get3DBorder** and the result is stored in the target. If TK_CONFIG_NULL_OK is specified in *specFlags* then the value may be an empty string, in which case the target will be set to NULL. If the previous value of the target wasn't NULL, then it is freed by passing it to **Tk_Free3DBorder**.

TK_CONFIG_CAP_STYLE

The value must be an ASCII string identifying a cap style in one of the ways accepted by **Tk_GetCapStyle**. The string is converted to an integer value corresponding to the cap style by calling **Tk_GetCapStyle** and the result is stored in the target.

TK_CONFIG_COLOR

The value must be an ASCII string identifying a color in a form suitable for passing to **Tk_GetColor**. The value is converted to an (**XColor ***) by calling **Tk_GetColor** and the result is stored in the target. If TK_CONFIG_NULL_OK is specified in *specFlags* then the value may be an empty string, in which case the target will be set to **None**. If the previous value of the target wasn't NULL, then it is freed by passing it to **Tk_FreeColor**.

TK_CONFIG_CURSOR

This option is identical to **TK_CONFIG_ACTIVE_CURSOR** except that the new cursor is not made the active one for *tkwin*.

TK_CONFIG_CUSTOM

This option allows applications to define new option types. The *customPtr* field of the entry points to a structure defining the new option type. See the section *[/CUSTOM OPTION TYPES](#)* below for details.

TK_CONFIG_DOUBLE

The value must be an ASCII floating−point number in the format accepted by **strtol**. The string is converted to a **double** value, and the value is stored in the target.

TK_CONFIG_END

Marks the end of the table. The last entry in *specs* must have this type; all of its other fields are ignored and it will never match any arguments.

TK_CONFIG_FONT

The value must be an ASCII string identifying a font in a form suitable for passing to **Tk_GetFontStruct**. The value is converted to an (**XFontStruct ***) by calling **Tk_GetFontStruct** and the result is stored in the target. If TK_CONFIG_NULL_OK is specified in *specFlags* then the value may be an empty string, in which case the target will be set to NULL. If the previous value of the target wasn‘t NULL, then it is freed by passing it to **Tk_FreeFontStruct**.

TK_CONFIG_INT

The value must be an ASCII integer string in the format accepted by **strtol** (e.g. ‘‘0‘’ and ‘‘0x‘’ prefixes may be used to specify octal or hexadecimal numbers, respectively). The string is converted to an integer value and the integer is stored in the target.

TK_CONFIG_JOIN_STYLE

The value must be an ASCII string identifying a join style in one of the ways accepted by **Tk_GetJoinStyle**. The string is converted to an integer value corresponding to the join style by calling **Tk_GetJoinStyle** and the result is stored in the target.

TK_CONFIG_JUSTIFY

The value must be an ASCII string identifying a justification method in one of the ways accepted by **Tk_GetJustify**. The string is converted to a **Tk_Justify** by calling **Tk_GetJustify** and the result is stored in the target.

TK_CONFIG_MM

The value must specify a screen distance in one of the forms acceptable to **Tk_GetScreenMM**. The string is converted to double−precision floating−point distance in millimeters and the value is stored in the target.

TK_CONFIG_PIXELS

The value must specify screen units in one of the forms acceptable to **Tk_GetPixels**. The string is converted to an integer distance in pixels and the value is stored in the target.

TK_CONFIG_RELIEF

The value must be an ASCII string identifying a relief in a form suitable for passing to **Tk_GetRelief**. The value is converted to an integer relief value by calling **Tk_GetRelief** and the result is stored in the target.

TK_CONFIG_STRING

A copy of the value is made by allocating memory space with **malloc** and copying the value into the dynamically−allocated space. A pointer to the new string is stored in the target. If TK_CONFIG_NULL_OK is specified in *specFlags* then the value may be an empty string, in which case the target will be set to NULL. If the previous value of the target wasn‘t NULL, then it is freed by passing it to **free**.

TK_CONFIG_SYNONYM

This *type* value identifies special entries in *specs* that are synonyms for other entries. If an *argv* value matches the *argvName* of a TK_CONFIG_SYNONYM entry, the entry isn't used directly. Instead, **Tk_ConfigureWidget** searches *specs* for another entry whose *argvName* is the same as the *dbName* field in the TK_CONFIG_SYNONYM entry; this new entry is used just as if its *argvName* had matched the *argv* value. The synonym mechanism allows multiple *argv* values to be used for a single configuration option, such as ''−background'' and ''−bg''.

TK_CONFIG_UID

The value is translated to a **Tk_Uid** (by passing it to **Tk_GetUid**). The resulting value is stored in the target. If TK_CONFIG_NULL_OK is specified in *specFlags* and the value is an empty string then the target will be set to NULL.

TK_CONFIG_WINDOW

The value must be a window path name. It is translated to a **Tk_Window** token and the token is stored in the target.

GROUPED ENTRIES

In some cases it is useful to generate multiple resources from a single configuration value. For example, a color name might be used both to generate the background color for a widget (using TK_CONFIG_COLOR) and to generate a 3−D border to draw around the widget (using TK_CONFIG_BORDER). In cases like this it is possible to specify that several consecutive entries in *specs* are to be treated as a group. The first entry is used to determine a value (using its *argvName*, *dbName*, *dbClass*, and *defValue* fields). The value will be processed several times (one for each entry in the group), generating multiple different resources and modifying multiple targets within *widgRec*. Each of the entries after the first must have a NULL value in its *argvName* field; this indicates that the entry is to be grouped with the entry that precedes it. Only the *type* and *offset* fields are used from these follow−on entries.

FLAGS

The *flags* argument passed to **Tk_ConfigureWidget** is used in conjunction with the *specFlags* fields in the entries of *specs* to provide additional control over the processing of configuration options. These values are used in three different ways as described below.

First, if the *flags* argument to **Tk_ConfigureWidget** has the TK_CONFIG_ARGV_ONLY bit set (i.e., *flags* | TK_CONFIG_ARGV_ONLY != 0), then the option database and *defValue* fields are not used. In this case, if an entry in *specs* doesn't match a field in *argv* then nothing happens: the corresponding target isn't modified. This feature is useful when the goal is to modify certain configuration options while leaving others in their current state, such as when a **configure** method is being processed.

Second, the *specFlags* field of an entry in *specs* may be used to control the processing of that entry. Each *specFlags* field may consists of an OR−ed combination of the following values:

TK_CONFIG_COLOR_ONLY

If this bit is set then the entry will only be considered if the display for *tkwin* has more than one bit plane. If the display is monochromatic then this *specs* entry will be ignored.

TK_CONFIG_MONO_ONLY

If this bit is set then the entry will only be considered if the display for *tkwin* has exactly one bit plane. If the display is not monochromatic then this *specs* entry will be ignored.

TK_CONFIG_NULL_OK

This bit is only relevant for some types of entries (see the descriptions of the various entry types above). If this bit is set, it indicates that an empty string value for the field is acceptable and if it occurs then the target should be set to NULL or **None**, depending on the type of the target. This flag is typically used to allow a feature to be turned off entirely, e.g. set a cursor value to **None** so that a window simply inherits its parent's cursor. If this bit isn't set then empty strings are processed as strings, which generally results in an error.

TK_CONFIG_DONT_SET_DEFAULT

If this bit is one, it means that the *defValue* field of the entry should only be used for returning the default value in **Tk_ConfigureInfo**. In calls to **Tk_ConfigureWidget** no default will be supplied for entries with this flag set; it is assumed that the caller has already supplied a default value in the target location. This flag provides a performance optimization where it is expensive to process the default string: the client can compute the default once, save the value, and provide it before calling **Tk_ConfigureWidget**.

TK_CONFIG_OPTION_SPECIFIED

This bit is set and cleared by **Tk_ConfigureWidget**. Whenever **Tk_ConfigureWidget** returns, this bit will be set in all the entries where a value was specified in *argv*. It will be zero in all other entries. This bit provides a way for clients to determine which values actually changed in a call to **Tk_ConfigureWidget**.

The TK_CONFIG_MONO_ONLY and TK_CONFIG_COLOR_ONLY flags are typically used to specify different default values for monochrome and color displays. This is done by creating two entries in *specs* that are identical except for their *defValue* and *specFlags* fields. One entry should have the value TK_CONFIG_MONO_ONLY in its *specFlags* and the default value for monochrome displays in its *defValue*; the other entry entry should have the value TK_CONFIG_COLOR_ONLY in its *specFlags* and the appropriate *defValue* for color displays.

Third, it is possible to use *flags* and *specFlags* together to selectively disable some entries. This feature is not needed very often. It is useful in cases where several similar kinds of widgets are implemented in one place. It allows a single *specs* table to be created with all the configuration options for all the widget types. When processing a particular widget type, only entries relevant to that type will be used. This effect is achieved by setting the high−order bits (those in positions equal to or greater than TK_CONFIG_USER_BIT) in *specFlags* values or in *flags*. In order for a particular entry in *specs* to be used, its high−order bits must match exactly the high−order bits of the *flags* value passed to **Tk_ConfigureWidget**. If a *specs* table is being used for N different widget types, then N of the high−order bits will be used. Each *specs* entry will have one of more of those bits set in its *specFlags* field to indicate the widget types for which this entry is valid. When calling **Tk_ConfigureWidget**, *flags* will have a single one of these bits set to select the entries for the desired widget type. For a working example of this feature, see the code in tkButton.c.

TK_OFFSET

The **Tk_Offset** macro is provided as a safe way of generating the *offset* values for entries in Tk_ConfigSpec structures. It takes two arguments: the name of a type of record, and the name of a field in that record. It returns the byte offset of the named field in records of the given type.

TK_CONFIGUREINFO

The **Tk_ConfigureInfo** procedure may be used to obtain information about one or all of the options for a given widget. Given a token for a window (*tkwin*), a table describing the configuration options for a class of widgets (*specs*), a pointer to a widget record containing the current information for a widget (*widgRec*), and a NULL *argvName* argument, **Tk_ConfigureInfo** generates a string describing all of the configuration options for the window. The string is placed in *interp−>result*. Under normal circumstances it returns TCL_OK; if an error occurs then it returns TCL_ERROR and *interp−>result* contains an error message.

If *argvName* is NULL, then the value left in *interp−>result* by **Tk_ConfigureInfo** consists of a list of one or more entries, each of which describes one configuration option (i.e. one entry in *specs*). Each entry in the list will contain either two or five values. If the corresponding entry in *specs* has type
TK_CONFIG_SYNONYM, then the list will contain two values: the *argvName* for the entry and the *dbName* (synonym name). Otherwise the list will contain five values: *argvName*, *dbName*, *dbClass*, *defValue*, and current value. The current value is computed from the appropriate field of *widgRec* by calling procedures like **Tk_NameOfColor**.

If the *argvName* argument to **Tk_ConfigureInfo** is non−NULL, then it indicates a single option, and information is returned only for that option. The string placed in *interp−>result* will be a list containing two

or five values as described above; this will be identical to the corresponding sublist that would have been returned if *argvName* had been NULL.

The *flags* argument to **Tk_ConfigureInfo** is used to restrict the *specs* entries to consider, just as for **Tk_ConfigureWidget**.

TK_CONFIGUREVALUE

Tk_ConfigureValue takes arguments similar to **Tk_ConfigureInfo**; instead of returning a list of values, it just returns the current value of the option given by *argvName* (*argvName* must not be NULL). The value is returned in *interp->result* and TCL_OK is normally returned as the procedure's result. If an error occurs in **Tk_ConfigureValue** (e.g., *argvName* is not a valid option name), TCL_ERROR is returned and an error message is left in *interp->result*. This procedure is typically called to implement **cget** widget commands.

TK_FREEOPTIONS

The **Tk_FreeOptions** procedure may be invoked during widget cleanup to release all of the resources associated with configuration options. It scans through *specs* and for each entry corresponding to a resource that must be explicitly freed (e.g. those with type TK_CONFIG_COLOR), it frees the resource in the widget record. If the field in the widget record doesn't refer to a resource (e.g. it contains a null pointer) then no resource is freed for that entry. After freeing a resource, **Tk_FreeOptions** sets the corresponding field of the widget record to null.

CUSTOM OPTION TYPES

Applications can extend the built-in configuration types with additional configuration types by writing procedures to parse and print options of the a type and creating a structure pointing to those procedures:

```
typedef struct Tk_CustomOption {
        Tk_OptionParseProc *parseProc;
        Tk_OptionPrintProc *printProc;
        ClientData clientData;
} Tk_CustomOption;

typedef int Tk_OptionParseProc(
        ClientData clientData,
        Tcl_Interp *interp,
        Tk_Window tkwin,
        char *value,
        char *widgRec,
        int offset);

typedef char *Tk_OptionPrintProc(
        ClientData clientData,
        Tk_Window tkwin,
        char *widgRec,
        int offset,
        Tcl_FreeProc **freeProcPtr);
```

The Tk_CustomOption structure contains three fields, which are pointers to the two procedures and a *clientData* value to be passed to those procedures when they are invoked. The *clientData* value typically points to a structure containing information that is needed by the procedures when they are parsing and printing options.

The *parseProc* procedure is invoked by **Tk_ConfigureWidget** to parse a string and store the resulting value in the widget record. The *clientData* argument is a copy of the *clientData* field in the Tk_CustomOption structure. The *interp* argument points to a Tcl interpreter used for error reporting. *Tkwin* is a copy of the *tkwin* argument to **Tk_ConfigureWidget**. The *value* argument is a string describing the value for the option; it could have been specified explicitly in the call to **Tk_ConfigureWidget** or it could come from the option database or a default. *Value* will never be a null pointer but it may point to an empty string. *RecordPtr* is the same as the *widgRec* argument to **Tk_ConfigureWidget**; it points to the start of the widget record to

modify. The last argument, *offset*, gives the offset in bytes from the start of the widget record to the location where the option value is to be placed. The procedure should translate the string to whatever form is appropriate for the option and store the value in the widget record. It should normally return TCL_OK, but if an error occurs in translating the string to a value then it should return TCL_ERROR and store an error message in *interp->result*.

The *printProc* procedure is called by **Tk_ConfigureInfo** to produce a string value describing an existing option. Its *clientData*, *tkwin*, *widgRec*, and *offset* arguments all have the same meaning as for Tk_OptionParseProc procedures. The *printProc* procedure should examine the option whose value is stored at *offset* in *widgRec*, produce a string describing that option, and return a pointer to the string. If the string is stored in dynamically-allocated memory, then the procedure must set **freeProcPtr* to the address of a procedure to call to free the string's memory; **Tk_ConfigureInfo** will call this procedure when it is finished with the string. If the result string is stored in static memory then *printProc* need not do anything with the *freeProcPtr* argument.

Once *parseProc* and *printProc* have been defined and a Tk_CustomOption structure has been created for them, options of this new type may be manipulated with Tk_ConfigSpec entries whose *type* fields are TK_CONFIG_CUSTOM and whose *customPtr* fields point to the Tk_CustomOption structure.

EXAMPLES

Although the explanation of **Tk_ConfigureWidget** is fairly complicated, its actual use is pretty straightforward. The easiest way to get started is to copy the code from an existing widget. The library implementation of frames (tkFrame.c) has a simple configuration table, and the library implementation of buttons (tkButton.c) has a much more complex table that uses many of the fancy *specFlags* mechanisms.

KEYWORDS

anchor, bitmap, boolean, border, cap style, color, configuration options, cursor, custom, double, font, integer, join style, justify, millimeters, pixels, relief, synonym, uid

NAME

Tk_ConfigureWindow, Tk_MoveWindow, Tk_ResizeWindow, Tk_MoveResizeWindow, Tk_SetWindowBorderWidth, Tk_ChangeWindowAttributes, Tk_SetWindowBackground, Tk_SetWindowBackgroundPixmap, Tk_SetWindowBorder, Tk_SetWindowBorderPixmap, Tk_SetWindowColormap, Tk_DefineCursor, Tk_UndefineCursor − change window configuration or attributes

=for category C Programming

SYNOPSIS

#include <tk.h>

Tk_ConfigureWindow(*tkwin, valueMask, valuePtr*)

Tk_MoveWindow(*tkwin, x, y*)

Tk_ResizeWindow(*tkwin, width, height*)

Tk_MoveResizeWindow(*tkwin, x, y, width, height*)

Tk_SetWindowBorderWidth(*tkwin, borderWidth*)

Tk_ChangeWindowAttributes(*tkwin, valueMask, attsPtr*)

Tk_SetWindowBackground(*tkwin, pixel*)

Tk_SetWindowBackgroundPixmap(*tkwin, pixmap*)

Tk_SetWindowBorder(*tkwin, pixel*)

Tk_SetWindowBorderPixmap(*tkwin, pixmap*)

Tk_SetWindowColormap(*tkwin, colormap*)

Tk_DefineCursor(*tkwin, cursor*)

Tk_UndefineCursor(*tkwin*)

ARGUMENTS

Tk_Window tkwin (in)

Token for window.

"unsigned int" valueMask (in)

OR−ed mask of values like **CWX** or **CWBorderPixel**, indicating which fields of **valuePtr* or **attsPtr* to use.

XWindowChanges *valuePtr (in)

Points to a structure containing new values for the configuration parameters selected by *valueMask*. Fields not selected by *valueMask* are ignored.

int x (in)

New x−coordinate for *tkwin*‘s top left pixel (including border, if any) within tkwin‘s parent.

int y (in)

New y−coordinate for *tkwin*‘s top left pixel (including border, if any) within tkwin‘s parent.

"int" width (in)

New width for *tkwin* (interior, not including border).

"int" height (in)

New height for *tkwin* (interior, not including border).

"int" borderWidth (in)

New width for *tkwin*'s border.

XSetWindowAttributes *attsPtr (in)

Points to a structure containing new values for the attributes given by the *valueMask* argument. Attributes not selected by *valueMask* are ignored.

"unsigned long" pixel (in)

New background or border color for window.

Pixmap pixmap (in)

New pixmap to use for background or border of *tkwin*. WARNING: cannot necessarily be deleted immediately, as for Xlib calls. See note below.

Colormap colormap (in)

New colormap to use for *tkwin*.

Tk_Cursor cursor (in)

New cursor to use for *tkwin*. If **None** is specified, then *tkwin* will not have its own cursor; it will use the cursor of its parent.

DESCRIPTION

These procedures are analogous to the X library procedures with similar names, such as **XConfigureWindow**. Each one of the above procedures calls the corresponding X procedure and also saves the configuration information in Tk's local structure for the window. This allows the information to be retrieved quickly by the application (using macros such as **Tk_X** and **Tk_Height**) without having to contact the X server. In addition, if no X window has actually been created for *tkwin* yet, these procedures do not issue X operations or cause event handlers to be invoked; they save the information in Tk's local structure for the window; when the window is created later, the saved information will be used to configure the window.

See the X library documentation for details on what these procedures do and how they use their arguments.

In the procedures **Tk_ConfigureWindow**, **Tk_MoveWindow**, **Tk_ResizeWindow**, **Tk_MoveResizeWindow**, and **Tk_SetWindowBorderWidth**, if *tkwin* is an internal window then event handlers interested in configure events are invoked immediately, before the procedure returns. If *tkwin* is a top−level window then the event handlers will be invoked later, after X has seen the request and returned an event for it.

Applications using Tk should never call procedures like **XConfigureWindow** directly; they should always use the corresponding Tk procedures.

The size and location of a window should only be modified by the appropriate geometry manager for that window and never by a window itself (but see *Tk::MoveToplev* for moving a top−level window).

You may not use **Tk_ConfigureWindow** to change the stacking order of a window (*valueMask* may not contain the **CWSibling** or **CWStackMode** bits). To change the stacking order, use the procedure **Tk_RestackWindow**.

The procedure **Tk_SetWindowColormap** will automatically add *tkwin* to the **TK_COLORMAP_WINDOWS** property of its nearest top−level ancestor if the new colormap is different from that of *tkwin*'s parent and *tkwin* isn't already in the **TK_COLORMAP_WINDOWS** property.

BUGS

Tk_SetWindowBackgroundPixmap and **Tk_SetWindowBorderPixmap** differ slightly from their Xlib counterparts in that the *pixmap* argument may not necessarily be deleted immediately after calling one of these procedures. This is because *tkwin*'s window may not exist yet at the time of the call, in which case *pixmap* is merely saved and used later when *tkwin*'s window is actually created. If you wish to delete *pixmap*, then call **Tk_MakeWindowExist** first to be sure that *tkwin*'s window exists and *pixmap* has been

passed to the X server.

A similar problem occurs for the *cursor* argument passed to **Tk_DefineCursor**. The solution is the same as for pixmaps above: call **Tk_MakeWindowExist** before freeing the cursor.

SEE ALSO

Tk::MoveToplev, *Tk::Restack*

KEYWORDS

attributes, border, color, configure, height, pixel, pixmap, width, window, x, y

NAME

Tk_CoordsToWindow – Find window containing a point

=for category C Programming

SYNOPSIS

#include <tk.h>

Tk_Window **Tk_CoordsToWindow**(*rootX, rootY, tkwin*)

ARGUMENTS

int rootX (in)

X–coordinate (in root window coordinates).

int rootY (in)

Y–coordinate (in root window coordinates).

Tk_Window tkwin (in)

Token for window that identifies application.

DESCRIPTION

Tk_CoordsToWindow locates the window that contains a given point. The point is specified in root coordinates with *rootX* and *rootY* (if a virtual–root window manager is in use then *rootX* and *rootY* are in the coordinate system of the virtual root window). The return value from the procedure is a token for the window that contains the given point. If the point is not in any window, or if the containing window is not in the same application as *tkwin*, then NULL is returned.

The containing window is decided using the same rules that determine which window contains the mouse cursor: if a parent and a child both contain the point then the child gets preference, and if two siblings both contain the point then the highest one in the stacking order (i.e. the one that's visible on the screen) gets preference.

KEYWORDS

containing, coordinates, root window

NAME

Tk_CreateErrorHandler, Tk_DeleteErrorHandler − handle X protocol errors

=for category C Programming

SYNOPSIS

#include <tk.h>

Tk_ErrorHandler **Tk_CreateErrorHandler**(*display, error, request, minor, proc, clientData*)

Tk_DeleteErrorHandler(*handler*)

ARGUMENTS

Display *display (in)

Display whose errors are to be handled.

int error (in)

Match only error events with this value in the *error_code* field. If −1, then match any *error_code* value.

int request (in)

Match only error events with this value in the *request_code* field. If −1, then match any *request_code* value.

int minor (in)

Match only error events with this value in the *minor_code* field. If −1, then match any *minor_code* value.

Tk_ErrorProc *proc (in)

Procedure to invoke whenever an error event is received for *display* and matches *error*, *request*, and *minor*. NULL means ignore any matching errors.

ClientData clientData (in)

Arbitrary one−word value to pass to *proc*.

Tk_ErrorHandler handler (in)

Token for error handler to delete (return value from a previous call to **Tk_CreateErrorHandler**).

DESCRIPTION

Tk_CreateErrorHandler arranges for a particular procedure (*proc*) to be called whenever certain protocol errors occur on a particular display (*display*). Protocol errors occur when the X protocol is used incorrectly, such as attempting to map a window that doesn't exist. See the Xlib documentation for **XSetErrorHandler** for more information on the kinds of errors that can occur. For *proc* to be invoked to handle a particular error, five things must occur:

[1] The error must pertain to *display*.

[2] Either the *error* argument to **Tk_CreateErrorHandler** must have been −1, or the *error* argument must match the *error_code* field from the error event.

[3] Either the *request* argument to **Tk_CreateErrorHandler** must have been −1, or the *request* argument must match the *request_code* field from the error event.

[4] Either the *minor* argument to **Tk_CreateErrorHandler** must have been −1, or the *minor* argument must match the *minor_code* field from the error event.

[5] The protocol request to which the error pertains must have been made when the handler was active (see below for more information).

Proc should have arguments and result that match the following type:

```
typedef int Tk_ErrorProc(
       ClientData clientData,
       XErrorEvent *errEventPtr);
```

The *clientData* parameter to *proc* is a copy of the *clientData* argument given to **Tcl_CreateErrorHandler** when the callback was created. Typically, *clientData* points to a data structure containing application−specific information that is needed to deal with the error. *ErrEventPtr* is a pointer to the X error event. The procedure *proc* should return an integer value. If it returns 0 it means that *proc* handled the error completely and there is no need to take any other action for the error. If it returns non−zero it means *proc* was unable to handle the error.

If a value of NULL is specified for *proc*, all matching errors will be ignored: this will produce the same result as if a procedure had been specified that always returns 0.

If more than more than one handler matches a particular error, then they are invoked in turn. The handlers will be invoked in reverse order of creation: most recently declared handler first. If any handler returns 0, then subsequent (older) handlers will not be invoked. If no handler returns 0, then Tk invokes X‘es default error handler, which prints an error message and aborts the program. If you wish to have a default handler that deals with errors that no other handler can deal with, then declare it first.

The X documentation states that ‘‘the error handler should not call any functions (directly or indirectly) on the display that will generate protocol requests or that will look for input events.’’ This restriction applies to handlers declared by **Tk_CreateErrorHandler**; disobey it at your own risk.

Tk_DeleteErrorHandler may be called to delete a previously−created error handler. The *handler* argument identifies the error handler, and should be a value returned by a previous call to **Tk_CreateEventHandler**.

A particular error handler applies to errors resulting from protocol requests generated between the call to **Tk_CreateErrorHandler** and the call to **Tk_DeleteErrorHandler**. However, the actual callback to *proc* may not occur until after the **Tk_DeleteErrorHandler** call, due to buffering in the client and server. If an error event pertains to a protocol request made just before calling **Tk_DeleteErrorHandler**, then the error event may not have been processed before the **Tk_DeleteErrorHandler** call. When this situation arises, Tk will save information about the handler and invoke the handler‘s *proc* later when the error event finally arrives. If an application wishes to delete an error handler and know for certain that all relevant errors have been processed, it should first call **Tk_DeleteErrorHandler** and then call **XSync**; this will flush out any buffered requests and errors, but will result in a performance penalty because it requires communication to and from the X server. After the **XSync** call Tk is guaranteed not to call any error handlers deleted before the **XSync** call.

For the Tk error handling mechanism to work properly, it is essential that application code never calls **XSetErrorHandler** directly; applications should use only **Tk_CreateErrorHandler**.

KEYWORDS

callback, error, event, handler

NAME

Tk_CreateGenericHandler, Tk_DeleteGenericHandler − associate procedure callback with all X events

=for category C Programming

SYNOPSIS

#include <tk.h>

Tk_CreateGenericHandler(*proc, clientData*)

Tk_DeleteGenericHandler(*proc, clientData*)

ARGUMENTS

Tk_GenericProc *proc (in)

Procedure to invoke whenever any X event occurs on any display.

ClientData clientData (in)

Arbitrary one−word value to pass to *proc*.

DESCRIPTION

Tk_CreateGenericHandler arranges for *proc* to be invoked in the future whenever any X event occurs. This mechanism is *not* intended for dispatching X events on windows managed by Tk (you should use **Tk_CreateEventHandler** for this purpose). **Tk_CreateGenericHandler** is intended for other purposes, such as tracing X events, monitoring events on windows not owned by Tk, accessing X−related libraries that were not originally designed for use with Tk, and so on.

The callback to *proc* will be made by **Tk_HandleEvent**; this mechanism only works in programs that dispatch events through **Tk_HandleEvent** (or through other Tk procedures that call **Tk_HandleEvent**, such as **Tk_DoOneEvent** or **Tk_MainLoop**).

Proc should have arguments and result that match the type **Tk_GenericProc**:

```
typedef int Tk_GenericProc(
        ClientData clientData,
        XEvent *eventPtr);
```

The *clientData* parameter to *proc* is a copy of the *clientData* argument given to **Tk_CreateGenericHandler** when the callback was created. Typically, *clientData* points to a data structure containing application−specific information about how to handle events. *EventPtr* is a pointer to the X event.

Whenever an X event is processed by **Tk_HandleEvent**, *proc* is called. The return value from *proc* is normally 0. A non−zero return value indicates that the event is not to be handled further; that is, *proc* has done all processing that is to be allowed for the event.

If there are multiple generic event handlers, each one is called for each event, in the order in which they were established.

Tk_DeleteGenericHandler may be called to delete a previously−created generic event handler: it deletes each handler it finds that matches the *proc* and *clientData* arguments. If no such handler exists, then **Tk_DeleteGenericHandler** returns without doing anything. Although Tk supports it, it's probably a bad idea to have more than one callback with the same *proc* and *clientData* arguments.

Establishing a generic event handler does nothing to ensure that the process will actually receive the X events that the handler wants to process. For example, it is the caller's responsibility to invoke **XSelectInput** to select the desired events, if that is necessary.

KEYWORDS

bind, callback, event, handler

NAME

Tk_CreateImageType, Tk_GetImageMasterData – define new kind of image

=for category C Programming

SYNOPSIS

#include <tk.h>

Tk_CreateImageType(*typePtr*) ClientData

Tk_GetImageMasterData(*interp, name, typePtrPtr*)

ARGUMENTS

Tk_ImageType *typePtr (in)

Structure that defines the new type of image. Must be static: a pointer to this structure is retained by the image code.

Tcl_Interp *interp (in)

Interpreter in which image was created.

char *name (in)

Name of existing image.

Tk_ImageType **typePtrPtr (out)

Points to word in which to store a pointer to type information for the given image, if it exists.

DESCRIPTION

Tk_CreateImageType is invoked to define a new kind of image. An image type corresponds to a particular value of the *type* argument for the **image create** command. There may exist any number of different image types, and new types may be defined dynamically by calling **Tk_CreateImageType**. For example, there might be one type for 2−color bitmaps, another for multi−color images, another for dithered images, another for video, and so on.

The code that implements a new image type is called an *image manager*. It consists of a collection of procedures plus three different kinds of data structures. The first data structure is a Tk_ImageType structure, which contains the name of the image type and pointers to five procedures provided by the image manager to deal with images of this type:

```
typedef struct Tk_ImageType {
        char *name;
        Tk_ImageCreateProc *createProc;
        Tk_ImageGetProc *getProc;
        Tk_ImageDisplayProc *displayProc;
        Tk_ImageFreeProc *freeProc;
        Tk_ImageDeleteProc *deleteProc;
} Tk_ImageType;
```

The fields of this structure will be described in later subsections of this entry.

The second major data structure manipulated by an image manager is called an *image master*; it contains overall information about a particular image, such as the values of the configuration options specified in an **image create** command. There will usually be one of these structures for each invocation of the **image create** command.

The third data structure related to images is an *image instance*. There will usually be one of these structures for each usage of an image in a particular widget. It is possible for a single image to appear simultaneously in multiple widgets, or even multiple times in the same widget. Furthermore, different instances may be on different screens or displays. The image instance data structure describes things that may vary from instance to instance, such as colors and graphics contexts for redisplay. There is usually one instance structure for

each **−image** option specified for a widget or canvas item.

The following subsections describe the fields of a Tk_ImageType in more detail.

name

typePtr−>name provides a name for the image type. Once **Tk_CreateImageType** returns, this name may be used in **image create** commands to create images of the new type. If there already existed an image type by this name then the new image type replaces the old one.

=for category Tk Library Procedures

CREATEPROC

typePtr−>createProc provides the address of a procedure for Tk to call whenever **image create** is invoked to create an image of the new type. *typePtr−>createProc* must match the following prototype:

```
typedef int Tk_ImageCreateProc(
        Tcl_Interp *interp,
        char *name,
        int argc,
        char **argv,
        Tk_ImageType *typePtr,
        Tk_ImageMaster master,
        ClientData *masterDataPtr);
```

The *interp* argument is the interpreter in which the **image** command was invoked, and *name* is the name for the new image, which was either specified explicitly in the **image** command or generated automatically by the **image** command. The *argc* and *argv* arguments describe all the configuration options for the new image (everything after the name argument to **image**). The *master* argument is a token that refers to Tk's information about this image; the image manager must return this token to Tk when invoking the **Tk_ImageChanged** procedure. Typically *createProc* will parse *argc* and *argv* and create an image master data structure for the new image. *createProc* may store an arbitrary one−word value at **masterDataPtr*, which will be passed back to the image manager when other callbacks are invoked. Typically the value is a pointer to the master data structure for the image.

If *createProc* encounters an error, it should leave an error message in *interp−>result* and return **TCL_ERROR**; otherwise it should return **TCL_OK**.

createProc should call **Tk_ImageChanged** in order to set the size of the image and request an initial redisplay.

GETPROC

typePtr−>getProc is invoked by Tk whenever a widget calls **Tk_GetImage** to use a particular image. This procedure must match the following prototype:

```
typedef ClientData Tk_ImageGetProc(
        Tk_Window tkwin,
        ClientData masterData);
```

The *tkwin* argument identifies the window in which the image will be used and *masterData* is the value returned by *createProc* when the image master was created. *getProc* will usually create a data structure for the new instance, including such things as the resources needed to display the image in the given window. *getProc* returns a one−word token for the instance, which is typically the address of the instance data structure. Tk will pass this value back to the image manager when invoking its *displayProc* and *freeProc* procedures.

DISPLAYPROC

typePtr−>displayProc is invoked by Tk whenever an image needs to be displayed (i.e., whenever a widget calls **Tk_RedrawImage**). *displayProc* must match the following prototype:

```
typedef void Tk_ImageDisplayProc(
```

```
        ClientData instanceData,
        Display *display,
        Drawable drawable,
        int imageX,
        int imageY,
        int width,
        int height,
        int drawableX,
        int drawableY);
```

The *instanceData* will be the same as the value returned by *getProc* when the instance was created. *display* and *drawable* indicate where to display the image; *drawable* may be a pixmap rather than the window specified to *getProc* (this is usually the case, since most widgets double−buffer their redisplay to get smoother visual effects). *imageX*, *imageY*, *width*, and *height* identify the region of the image that must be redisplayed. This region will always be within the size of the image as specified in the most recent call to **Tk_ImageChanged**. *drawableX* and *drawableY* indicate where in *drawable* the image should be displayed; *displayProc* should display the given region of the image so that point (*imageX*, *imageY*) in the image appears at (*drawableX*, *drawableY*) in *drawable*.

FREEPROC

typePtr−>freeProc contains the address of a procedure that Tk will invoke when an image instance is released (i.e., when **Tk_FreeImage** is invoked). This can happen, for example, when a widget is deleted or a image item in a canvas is deleted, or when the image displayed in a widget or canvas item is changed. *freeProc* must match the following prototype:

```
typedef void Tk_ImageFreeProc(
        ClientData instanceData,
        Display *display);
```

The *instanceData* will be the same as the value returned by *getProc* when the instance was created, and *display* is the display containing the window for the instance. *freeProc* should release any resources associated with the image instance, since the instance will never be used again.

DELETEPROC

typePtr−>deleteProc is a procedure that Tk invokes when an image is being deleted (i.e. when the **image delete** command is invoked). Before invoking *deleteProc* Tk will invoke *freeProc* for each of the image‘s instances. *deleteProc* must match the following prototype:

```
typedef void Tk_ImageDeleteProc(
        ClientData masterData);
```

The *masterData* argument will be the same as the value stored in **masterDataPtr* by *createProc* when the image was created. *deleteProc* should release any resources associated with the image.

TK_GETIMAGEMASTERDATA

The procedure **Tk_GetImageMasterData** may be invoked to retrieve information about an image. For example, an image manager can use this procedure to locate its image master data for an image. If there exists an image named *name* in the interpreter given by *interp*, then **typePtrPtr* is filled in with type information for the image (the *typePtr* value passed to **Tk_CreateImageType** when the image type was registered) and the return value is the ClientData value returned by the *createProc* when the image was created (this is typically a pointer to the image master data structure). If no such image exists then NULL is returned and NULL is stored at **typePtrPtr*.

SEE ALSO

Tk::ImgChanged, *Tk::GetImage*, *Tk::GetImage*, *Tk::GetImage*, *Tk::GetImage*

KEYWORDS

image manager, image type, instance, master

NAME

Tk_CreateItemType, Tk_GetItemTypes – define new kind of canvas item

=for category C Programming

SYNOPSIS

#include <tk.h>

Tk_CreateItemType(*typePtr*)

Tk_ItemType * **Tk_GetItemTypes**()

ARGUMENTS

Tk_ItemType *typePtr (in)

Structure that defines the new type of canvas item.

INTRODUCTION

Tk_CreateItemType is invoked to define a new kind of canvas item described by the *typePtr* argument. An item type corresponds to a particular value of the *type* argument to the **create** method for canvases, and the code that implements a canvas item type is called a *type manager*. Tk defines several built–in item types, such as **rectangle** and **text** and **image**, but **Tk_CreateItemType** allows additional item types to be defined. Once **Tk_CreateItemType** returns, the new item type may be used in new or existing canvas widgets just like the built–in item types.

Tk_GetItemTypes returns a pointer to the first in the list of all item types currently defined for canvases. The entries in the list are linked together through their *nextPtr* fields, with the end of the list marked by a NULL *nextPtr*.

You may find it easier to understand the rest of this manual entry by looking at the code for an existing canvas item type such as bitmap (file tkCanvBmap.c) or text (tkCanvText.c). The easiest way to create a new type manager is to copy the code for an existing type and modify it for the new type.

Tk provides a number of utility procedures for the use of canvas type managers, such as **Tk_CanvasCoords** and **Tk_CanvasPsColor**; these are described in separate manual entries.

DATA STRUCTURES

A type manager consists of a collection of procedures that provide a standard set of operations on items of that type. The type manager deals with three kinds of data structures. The first data structure is a Tk_ItemType; it contains information such as the name of the type and pointers to the standard procedures implemented by the type manager:

```
typedef struct Tk_ItemType {
        char *name;
        int itemSize;
        Tk_ItemCreateProc *createProc;
        Tk_ConfigSpec *configSpecs;
        Tk_ItemConfigureProc *configProc;
        Tk_ItemCoordProc *coordProc;
        Tk_ItemDeleteProc *deleteProc;
        Tk_ItemDisplayProc *displayProc;
        int alwaysRedraw;
        Tk_ItemPointProc *pointProc;
        Tk_ItemAreaProc *areaProc;
        Tk_ItemPostscriptProc *postscriptProc;
        Tk_ItemScaleProc *scaleProc;
        Tk_ItemTranslateProc *translateProc;
        Tk_ItemIndexProc *indexProc;
        Tk_ItemCursorProc *icursorProc;
```

```
        Tk_ItemSelectionProc *selectionProc;
        Tk_ItemInsertProc *insertProc;
        Tk_ItemDCharsProc *dCharsProc;
        Tk_ItemType *nextPtr;
} Tk_ItemType;
```

The fields of a Tk_ItemType structure are described in more detail later in this manual entry. When **Tk_CreateItemType** is called, its *typePtr* argument must point to a structure with all of the fields initialized except *nextPtr*, which Tk sets to link all the types together into a list. The structure must be in permanent memory (either statically allocated or dynamically allocated but never freed); Tk retains a pointer to this structure.

The second data structure manipulated by a type manager is an *item record*. For each item in a canvas there exists one item record. All of the items of a given type generally have item records with the same structure, but different types usually have different formats for their item records. The first part of each item record is a header with a standard structure defined by Tk via the type Tk_Item; the rest of the item record is defined by the type manager. A type manager must define its item records with a Tk_Item as the first field. For example, the item record for bitmap items is defined as follows:

```
typedef struct BitmapItem {
        Tk_Item header;
        double x, y;
        Tk_Anchor anchor;
        Pixmap bitmap;
        XColor *fgColor;
        XColor *bgColor;
        GC gc;
} BitmapItem;
```

The *header* substructure contains information used by Tk to manage the item, such as its identifier, its tags, its type, and its bounding box. The fields starting with *x* belong to the type manager: Tk will never read or write them. The type manager should not need to read or write any of the fields in the header except for four fields whose names are *x1*, *y1*, *x2*, and *y2*. These fields give a bounding box for the items using integer canvas coordinates: the item should not cover any pixels with x−coordinate lower than *x1* or y−coordinate lower than *y1*, nor should it cover any pixels with x−coordinate greater than or equal to *x2* or y−coordinate greater than or equal to *y2*. It is up to the type manager to keep the bounding box up to date as the item is moved and reconfigured.

Whenever Tk calls a procedure in a type manager it passes in a pointer to an item record. The argument is always passed as a pointer to a Tk_Item; the type manager will typically cast this into a pointer to its own specific type, such as BitmapItem.

The third data structure used by type managers has type Tk_Canvas; it serves as an opaque handle for the canvas widget as a whole. Type managers need not know anything about the contents of this structure. A Tk_Canvas handle is typically passed in to the procedures of a type manager, and the type manager can pass the handle back to library procedures such as Tk_CanvasTkwin to fetch information about the canvas.

name

This section and the ones that follow describe each of the fields in a Tk_ItemType structure in detail. The *name* field provides a string name for the item type. Once **Tk_CreateImageType** returns, this name may be used in **create** methods to create items of the new type. If there already existed an item type by this name then the new item type replaces the old one.

itemSize

typePtr−>itemSize gives the size in bytes of item records of this type, including the Tk_Item header. Tk uses this size to allocate memory space for items of the type. All of the item records for a given type must have the same size. If variable length fields are needed for an item (such as a list of points for a polygon), the type manager can allocate a separate object of variable length and keep a pointer to it in the item record.

CREATEPROC

typePtr−>createProc points to a procedure for Tk to call whenever a new item of this type is created. *typePtr−>createProc* must match the following prototype:

```
typedef int Tk_ItemCreateProc(
        Tcl_Interp *interp,
        Tk_Canvas canvas,
        Tk_Item *itemPtr,
        int argc,
        char **argv);
```

The *interp* argument is the interpreter in which the canvas's **create** method was invoked, and *canvas* is a handle for the canvas widget. *itemPtr* is a pointer to a newly−allocated item of size *typePtr−>itemSize*. Tk has already initialized the item's header (the first **sizeof(Tk_ItemType)** bytes). The *argc* and *argv* arguments describe all of the arguments to the **create** command after the *type* argument. For example, in the method

```
.c create rectangle 10 20 50 50 −fill black
```

argc will be **6** and *argv*[0] will contain the string **10**.

createProc should use *argc* and *argv* to initialize the type−specific parts of the item record and set an initial value for the bounding box in the item's header. It should return a standard Tcl completion code and leave an error message in *interp−>result* if an error occurs. If an error occurs Tk will free the item record, so *createProc* must be sure to leave the item record in a clean state if it returns an error (e.g., it must free any additional memory that it allocated for the item).

CONFIGSPECS

Each type manager must provide a standard table describing its configuration options, in a form suitable for use with **Tk_ConfigureWidget**. This table will normally be used by *typePtr−>createProc* and *typePtr−>configProc*, but Tk also uses it directly to retrieve option information in the **itemcget** and **itemconfigure** methods. *typePtr−>configSpecs* must point to the configuration table for this type. Note: Tk provides a custom option type **tk_CanvasTagsOption** for implementing the **−tags** option; see an existing type manager for an example of how to use it in *configSpecs*.

CONFIGPROC

typePtr−>configProc is called by Tk whenever the **itemconfigure** method is invoked to change the configuration options for a canvas item. This procedure must match the following prototype:

```
typedef int Tk_ItemConfigureProc(
        Tcl_Interp *interp,
        Tk_Canvas canvas,
        Tk_Item *itemPtr,
        int argc,
        char **argv,
        int flags);
```

The *interp* argument identifies the interpreter in which the method was invoked, *canvas* is a handle for the canvas widget, and *itemPtr* is a pointer to the item being configured. *argc* and *argv* contain the configuration options. For example, if the following command is invoked:

```
.c itemconfigure 2 −fill red −outline black
```

argc is **4** and *argv* contains the strings **−fill** through **black**. *argc* will always be an even value. The *flags* argument contains flags to pass to **Tk_ConfigureWidget**; currently this value is always TK_CONFIG_ARGV_ONLY when Tk invokes *typePtr−>configProc*, but the type manager's *createProc* procedure will usually invoke *configProc* with different flag values.

typePtr−>configProc returns a standard Tcl completion code and leaves an error message in *interp−>result* if an error occurs. It must update the item's bounding box to reflect the new configuration options.

COORDPROC

typePtr−>coordProc is invoked by Tk to implement the **coords** method for an item. It must match the following prototype:

```
typedef int Tk_ItemCoordProc(
        Tcl_Interp *interp,
        Tk_Canvas canvas,
        Tk_Item *itemPtr,
        int argc,
        char **argv);
```

The arguments *interp*, *canvas*, and *itemPtr* all have the standard meanings, and *argc* and *argv* describe the coordinate arguments. For example, if the following method is invoked:

```
.c coords 2 30 90
```

argc will be **2** and **argv** will contain the string values **30** and **90**.

The *coordProc* procedure should process the new coordinates, update the item appropriately (e.g., it must reset the bounding box in the item's header), and return a standard Tcl completion code. If an error occurs, *coordProc* must leave an error message in *interp−>result*.

DELETEPROC

typePtr−>deleteProc is invoked by Tk to delete an item and free any resources allocated to it. It must match the following prototype:

```
typedef void Tk_ItemDeleteProc(
        Tk_Canvas canvas,
        Tk_Item *itemPtr,
        Display *display);
```

The *canvas* and *itemPtr* arguments have the usual interpretations, and *display* identifies the X display containing the canvas. *deleteProc* must free up any resources allocated for the item, so that Tk can free the item record. *deleteProc* should not actually free the item record; this will be done by Tk when *deleteProc* returns.

DISPLAYPROC AND ALWAYSREDRAW

typePtr−>displayProc is invoked by Tk to redraw an item on the screen. It must match the following prototype:

```
typedef void Tk_ItemDisplayProc(
        Tk_Canvas canvas,
        Tk_Item *itemPtr,
        Display *display,
        Drawable dst,
        int x,
        int y,
        int width,
        int height);
```

The *canvas* and *itemPtr* arguments have the usual meaning. *display* identifies the display containing the canvas, and *dst* specifies a drawable in which the item should be rendered; typically this is an off−screen pixmap, which Tk will copy into the canvas's window once all relevant items have been drawn. *x*, *y*, *width*, and *height* specify a rectangular region in canvas coordinates, which is the area to be redrawn; only information that overlaps this area needs to be redrawn. Tk will not call *displayProc* unless the item's bounding box overlaps the redraw area, but the type manager may wish to use the redraw area to optimize the redisplay of the item.

Because of scrolling and the use of off−screen pixmaps for double−buffered redisplay, the item's coordinates

in *dst* will not necessarily be the same as those in the canvas. *displayProc* should call **Tk_CanvasDrawableCoords** to transform coordinates from those of the canvas to those of *dst*.

Normally an item's *displayProc* is only invoked if the item overlaps the area being displayed. However, if *typePtr->alwaysRedraw* has a non-zero value, then *displayProc* is invoked during every redisplay operation, even if the item doesn't overlap the area of redisplay. *alwaysRedraw* should normally be set to 0; it is only set to 1 in special cases such as window items that need to be unmapped when they are off-screen.

POINTPROC

typePtr->pointProc is invoked by Tk to find out how close a given point is to a canvas item. Tk uses this procedure for purposes such as locating the item under the mouse or finding the closest item to a given point. The procedure must match the following prototype:

```
typedef double Tk_ItemPointProc(
        Tk_Canvas canvas,
        Tk_Item *itemPtr,
        double *pointPtr);
```

canvas and *itemPtr* have the usual meaning. *pointPtr* points to an array of two numbers giving the x and y coordinates of a point. *pointProc* must return a real value giving the distance from the point to the item, or 0 if the point lies inside the item.

AREAPROC

typePtr->areaProc is invoked by Tk to find out the relationship between an item and a rectangular area. It must match the following prototype:

```
typedef int Tk_ItemAreaProc(
        Tk_Canvas canvas,
        Tk_Item *itemPtr,
        double *rectPtr);
```

canvas and *itemPtr* have the usual meaning. *rectPtr* points to an array of four real numbers; the first two give the x and y coordinates of the upper left corner of a rectangle, and the second two give the x and y coordinates of the lower right corner. *areaProc* must return -1 if the item lies entirely outside the given area, 0 if it lies partially inside and partially outside the area, and 1 if it lies entirely inside the area.

POSTSCRIPTPROC

typePtr->postscriptProc is invoked by Tk to generate Postscript for an item during the **postscript** method. If the type manager is not capable of generating Postscript then *typePtr->postscriptProc* should be NULL. The procedure must match the following prototype:

```
typedef int Tk_ItemPostscriptProc(
        Tcl_Interp *interp,
        Tk_Canvas canvas,
        Tk_Item *itemPtr,
        int prepass);
```

The *interp*, *canvas*, and *itemPtr* arguments all have standard meanings; *prepass* will be described below. If *postscriptProc* completes successfully, it should append Postscript for the item to the information in *interp->result* (e.g. by calling **Tcl_AppendResult**, not **Tcl_SetResult**) and return TCL_OK. If an error occurs, *postscriptProc* should clear the result and replace its contents with an error message; then it should return TCL_ERROR.

Tk provides a collection of utility procedures to simplify *postscriptProc*. For example, **Tk_CanvasPsColor** will generate Postscript to set the current color to a given Tk color and **Tk_CanvasPsFont** will set up font information. When generating Postscript, the type manager is free to change the graphics state of the Postscript interpreter, since Tk places **gsave** and **grestore** commands around the Postscript for the item. The type manager can use canvas x coordinates directly in its Postscript, but it must call **Tk_CanvasPsY** to convert y coordinates from the space of the canvas (where the origin is at the upper left) to the space of

Postscript (where the origin is at the lower left).

In order to generate Postscript that complies with the Adobe Document Structuring Conventions, Tk actually generates Postscript in two passes. It calls each item's *postscriptProc* in each pass. The only purpose of the first pass is to collect font information (which is done by **Tk_CanvPsFont**); the actual Postscript is discarded. Tk sets the *prepass* argument to *postscriptProc* to 1 during the first pass; the type manager can use *prepass* to skip all Postscript generation except for calls to **Tk_CanvasPsFont**. During the second pass *prepass* will be 0, so the type manager must generate complete Postscript.

SCALEPROC

typePtr->scaleProc is invoked by Tk to rescale a canvas item during the **scale** method. The procedure must match the following prototype:

```
typedef void Tk_ItemScaleProc(
        Tk_Canvas canvas,
        Tk_Item *itemPtr,
        double originX,
        double originY,
        double scaleX,
        double scaleY);
```

The *canvas* and *itemPtr* arguments have the usual meaning. *originX* and *originY* specify an origin relative to which the item is to be scaled, and *scaleX* and *scaleY* give the x and y scale factors. The item should adjust its coordinates so that a point in the item that used to have coordinates *x* and *y* will have new coordinates *x'* and *y'*, where

```
x' = originX  + scaleX*(x-originX)
y' = originY + scaleY*(y-originY)
```

scaleProc must also update the bounding box in the item's header.

TRANSLATEPROC

typePtr->translateProc is invoked by Tk to translate a canvas item during the **move** method. The procedure must match the following prototype:

```
typedef void Tk_ItemTranslateProc(
        Tk_Canvas canvas,
        Tk_Item *itemPtr,
        double deltaX,
        double deltaY);
```

The *canvas* and *itemPtr* arguments have the usual meaning, and *deltaX* and *deltaY* give the amounts that should be added to each x and y coordinate within the item. The type manager should adjust the item's coordinates and update the bounding box in the item's header.

INDEXPROC

typePtr->indexProc is invoked by Tk to translate a string index specification into a numerical index, for example during the **index** method. It is only relevant for item types that support indexable text; *typePtr->indexProc* may be specified as NULL for non-textual item types. The procedure must match the following prototype:

```
typedef int Tk_ItemIndexProc(
        Tcl_Interp *interp,
        Tk_Canvas canvas,
        Tk_Item *itemPtr,
        char indexString,
        int *indexPtr);
```

The *interp*, *canvas*, and *itemPtr* arguments all have the usual meaning. *indexString* contains a textual description of an index, and *indexPtr* points to an integer value that should be filled in with a numerical

index. It is up to the type manager to decide what forms of index are supported (e.g., numbers, **insert**, **sel.first**, **end**, etc.). *indexProc* should return a Tcl completion code and set *interp->result* in the event of an error.

ICURSORPROC

typePtr->icursorProc is invoked by Tk during the **icursor** method to set the position of the insertion cursor in a textual item. It is only relevant for item types that support an insertion cursor; *typePtr->icursorProc* may be specified as NULL for item types that don't support an insertion cursor. The procedure must match the following prototype:

```
typedef void Tk_ItemIndexProc(
        Tk_Canvas canvas,
        Tk_Item *itemPtr,
        int index);
```

canvas and *itemPtr* have the usual meanings, and *index* is an index into the item's text, as returned by a previous call to *typePtr->insertProc*. The type manager should position the insertion cursor in the item just before the character given by *index*. Whether or not to actually display the insertion cursor is determined by other information provided by **Tk_CanvasGetTextInfo**.

SELECTIONPROC

typePtr->selectionProc is invoked by Tk during selection retrievals; it must return part or all of the selected text in the item (if any). It is only relevant for item types that support text; *typePtr->selectionProc* may be specified as NULL for non-textual item types. The procedure must match the following prototype:

```
typedef int Tk_ItemSelectionProc(
        Tk_Canvas canvas,
        Tk_Item *itemPtr,
        int offset,
        char *buffer,
        int maxBytes);
```

canvas and *itemPtr* have the usual meanings. *offset* is an offset in bytes into the selection where 0 refers to the first byte of the selection; it identifies the first character that is to be returned in this call. *buffer* points to an area of memory in which to store the requested bytes, and *maxBytes* specifies the maximum number of bytes to return. *selectionProc* should extract up to *maxBytes* characters from the selection and copy them to *maxBytes*; it should return a count of the number of bytes actually copied, which may be less than *maxBytes* if there aren't *offset+maxBytes* bytes in the selection.

INSERTPROC

typePtr->insertProc is invoked by Tk during the **insert** method to insert new text into a canvas item. It is only relevant for item types that support text; *typePtr->insertProc* may be specified as NULL for non-textual item types. The procedure must match the following prototype:

```
typedef void Tk_ItemInsertProc(
        Tk_Canvas canvas,
        Tk_Item *itemPtr,
        int index,
        char *string);
```

canvas and *itemPtr* have the usual meanings. *index* is an index into the item's text, as returned by a previous call to *typePtr->insertProc*, and *string* contains new text to insert just before the character given by *index*. The type manager should insert the text and recompute the bounding box in the item's header.

DCHARSPROC

typePtr->dCharsProc is invoked by Tk during the **dchars** method to delete a range of text from a canvas item. It is only relevant for item types that support text; *typePtr->dCharsProc* may be specified as NULL for non-textual item types. The procedure must match the following prototype:

```
typedef void Tk_ItemDCharsProc(
        Tk_Canvas canvas,
        Tk_Item *itemPtr,
        int first,
        int last);
```

canvas and *itemPtr* have the usual meanings. *first* and *last* give the indices of the first and last bytes to be deleted, as returned by previous calls to *typePtr->indexProc*. The type manager should delete the specified characters and update the bounding box in the item's header.

SEE ALSO

Tk::CanvPsY, *Tk::CanvTxtInfo*, *Tk::CanvTkwin*

KEYWORDS

canvas, focus, item type, selection, type manager

NAME

Tk_CreateMainWindow, Tk_CreateWindow, Tk_CreateWindowFromPath, Tk_DestroyWindow, Tk_MakeWindowExist − create or delete window

=for category C Programming

SYNOPSIS

#include <tk.h>

Tk_Window **Tk_CreateMainWindow**(*interp, screenName, baseName, className*)

Tk_Window **Tk_CreateWindow**(*interp, parent, name, topLevScreen*)

Tk_Window **Tk_CreateWindowFromPath**(*interp, tkwin, pathName, topLevScreen*)

Tk_DestroyWindow(*tkwin*)

Tk_MakeWindowExist(*tkwin*)

ARGUMENTS

Tcl_Interp *interp (out)

Tcl interpreter to use for error reporting. If no error occurs, then **interp* isn't modified. For **Tk_CreateMainWindow**, this interpreter is associated permanently with the created window, and Tk−related commands are bound into the interpreter.

char *screenName (in)

String name of screen on which to create window. Has the form *displayName***.***screenNum*, where *displayName* is the name of a display and *screenNum* is a screen number. If the dot and *screenNum* are omitted, the screen number defaults to 0. If *screenName* is NULL or empty string, defaults to contents of DISPLAY environment variable.

char *baseName (in)

Name to use for this main window. See below for details.

char *className (in)

Class to use for application and for main window.

Tk_Window parent (in)

Token for the window that is to serve as the logical parent of the new window.

char *name (in)

Name to use for this window. Must be unique among all children of the same *parent*.

char *topLevScreen (in)

Has same format as *screenName*. If NULL, then new window is created as an internal window. If non−NULL, new window is created as a top−level window on screen *topLevScreen*. If *topLevScreen* is an empty string (‘‘‘) then new window is created as top−level window of *parent*'s screen.

Tk_Window tkwin (in)

Token for window.

char *pathName (in)

Name of new window, specified as path name within application (e.g. **.a.b.c**).

DESCRIPTION

The three procedures **Tk_CreateMainWindow**, **Tk_CreateWindow**, and **Tk_CreateWindowFromPath** are used to create new windows for use in Tk−based applications. Each of the procedures returns a token that can be used to manipulate the window in other calls to the Tk library. If the window couldn't be created successfully, then NULL is returned and *interp−>result* is modified to hold an error message.

Tk supports three different kinds of windows: main windows, internal windows, and top−level windows. A main window is the outermost window corresponding to an application. Main windows correspond to the independent units of an application, such as a view on a file that is part of an editor, or a clock, or a terminal emulator. A main window is created as a child of the root window of the screen indicated by the *screenName*. Each main window, and all its descendants, are typically associated with a single Tcl command interpreter. An internal window is an interior window of a Tk application, such as a scrollbar or menu bar or button. A top−level window is one that is created as a child of a screen's root window, rather than as an interior window, but which is logically part of some existing main window. Examples of top−level windows are pop−up menus and dialog boxes.

Tk_CreateMainWindow creates a new main window and associates its *interp* argument with that window and all its eventual descendants. **Tk_CreateMainWindow** also carries out several other actions to set up the new application. First, it adds all the Tk commands to those already defined for *interp*. Second, it turns the new window into a **toplevel** widget, which will cause the X window to be created and mapped as soon as the application goes idle. Third, **Tk_CreateMainWindow** registers *interp* so that it can be accessed remotely by other Tk applications using the **send** command and the name *baseName*. Normally, *baseName* consists of the name of the application followed by a space and an identifier for this particular main window (if such an identifier is relevant). For example, an editor named **mx** displaying the file **foo.c** would use ''mx foo.c'' as the basename. An application that doesn't usually have multiple instances, such as a clock program, would just use the name of the application, e.g. ''xclock''. If *baseName* is already in use by some other registered interpreter, then **Tk_CreateMainWindow** extends *baseName* with a number to produce a unique name like ''mx foo.c #2'' or ''xclock #12''. This name is used both as the name of the window (returned by **Tk_Name**) and as the registered name of the interpreter. Fourth, **Tk_CreateMainWindow** sets *className* as the class of the application (among other things, this is used for lookups in the option database), and also as the class of the main widget.

Either internal or top−level windows may be created by calling **Tk_CreateWindow**. If the *topLevScreen* argument is NULL, then the new window will be an internal window. If *topLevScreen* is non−NULL, then the new window will be a top−level window: *topLevScreen* indicates the name of a screen and the new window will be created as a child of the root window of *topLevScreen*. In either case Tk will consider the new window to be the logical child of *parent*: the new window's path name will reflect this fact, options may be specified for the new window under this assumption, and so on. The only difference is that new X window for a top−level window will not be a child of *parent*'s X window. For example, a pull−down menu's *parent* would be the button−like window used to invoke it, which would in turn be a child of the menu bar window. A dialog box might have the application's main window as its parent. This approach means that all the windows of an application fall into a hierarchical arrangement with a single logical root: the application's main window.

Tk_CreateWindowFromPath offers an alternate way of specifying new windows. In **Tk_CreateWindowFromPath** the new window is specified with a token for any window in the target application (*tkwin*), plus a path name for the new window. It produces the same effect as **Tk_CreateWindow** and allows both top−level and internal windows to be created, depending on the value of *topLevScreen*. In calls to **Tk_CreateWindowFromPath**, as in calls to **Tk_CreateWindow**, the parent of the new window must exist at the time of the call, but the new window must not already exist.

In truth, the window−creation procedures don't actually issue the command to X to create a window. Instead, they create a local data structure associated with the window and defer the creation of the X window. The window will actually be created by the first call to **Tk_MapWindow**. Deferred window creation allows various aspects of the window (such as its size, background color, etc.) to be modified after its creation without incurring any overhead in the X server. When the window is finally mapped all of the window attributes can be set while creating the window.

The value returned by a window−creation procedure is not the X token for the window (it can't be, since X hasn't been asked to create the window yet). Instead, it is a token for Tk's local data structure for the window. Most of the Tk library procedures take Tk_Window tokens, rather than X identifiers. The actual X window identifier can be retrieved from the local data structure using the **Tk_WindowId** macro; see the *Tk::WindowId* documentation for details.

Tk_DestroyWindow deletes a window and all the data structures associated with it, including any event handlers created with **Tk_CreateEventHandler**. In addition, **Tk_DestroyWindow** will delete any children of *tkwin* recursively (where children are defined in the Tk sense, consisting of all windows that were created with the given window as *parent*). If *tkwin* was created by **Tk_CreateInternalWindow** then event handlers interested in destroy events are invoked immediately. If *tkwin* is a top−level or main window, then the event handlers will be invoked later, after X has seen the request and returned an event for it.

If a window has been created but hasn't been mapped, so no X window exists, it is possible to force the creation of the X window by calling **Tk_MakeWindowExist**. This procedure issues the X commands to instantiate the window given by *tkwin*.

KEYWORDS

create, deferred creation, destroy, display, internal window, main window, register, screen, top−level window, window

NAME

Tk_CreatePhotoImageFormat – define new file format for photo images

=for category C Programming

SYNOPSIS

#include <tk.h> #include <tkPhoto.h>

Tk_CreatePhotoImageFormat(*formatPtr*)

ARGUMENTS

Tk_PhotoImageFormat *formatPtr (in)

Structure that defines the new file format.

DESCRIPTION

Tk_CreatePhotoImageFormat is invoked to define a new file format for image data for use with photo images. The code that implements an image file format is called an image file format handler, or handler for short. The photo image code maintains a list of handlers that can be used to read and write data to or from a file. Some handlers may also support reading image data from a string or converting image data to a string format. The user can specify which handler to use with the **–format** image configuration option or the **–format** option to the **read** and **write** photo image subcommands.

An image file format handler consists of a collection of procedures plus a Tk_PhotoImageFormat structure, which contains the name of the image file format and pointers to six procedures provided by the handler to deal with files and strings in this format. The Tk_PhotoImageFormat structure contains the following fields:

```
typedef struct Tk_PhotoImageFormat {
        char *name;
        Tk_ImageFileMatchProc *fileMatchProc;
        Tk_ImageStringMatchProc *stringMatchProc;
        Tk_ImageFileReadProc *fileReadProc;
        Tk_ImageStringReadProc *stringReadProc;
        Tk_ImageFileWriteProc *fileWriteProc;
        Tk_ImageStringWriteProc *stringWriteProc;
} Tk_PhotoImageFormat;
```

The handler need not provide implementations of all six procedures. For example, the procedures that handle string data would not be provided for a format in which the image data are stored in binary, and could therefore contain null characters. If any procedure is not implemented, the corresponding pointer in the Tk_PhotoImageFormat structure should be set to NULL. The handler must provide the *fileMatchProc* procedure if it provides the *fileReadProc* procedure, and the *stringMatchProc* procedure if it provides the *stringReadProc* procedure.

name

formatPtr−>name provides a name for the image type. Once **Tk_CreatePhotoImageFormat** returns, this name may be used in the **–format** photo image configuration and subcommand option. The manual page for the photo image (photo(n)) describes how image file formats are chosen based on their names and the value given to the **–format** option.

FILEMATCHPROC

formatPtr−>fileMatchProc provides the address of a procedure for Tk to call when it is searching for an image file format handler suitable for reading data in a given file. *formatPtr−>fileMatchProc* must match the following prototype:

```
typedef int Tk_ImageFileMatchProc(
        Tcl_Channel chan,
        char *fileName,
        char *formatString,
```

```
        int *widthPtr,
        int *heightPtr);
```

The *fileName* argument is the name of the file containing the image data, which is open for reading as *chan*. The *formatString* argument contains the value given for the **–format** option, or NULL if the option was not specified. If the data in the file appears to be in the format supported by this handler, the *formatPtr–>fileMatchProc* procedure should store the width and height of the image in **widthPtr* and **heightPtr* respectively, and return 1. Otherwise it should return 0.

STRINGMATCHPROC

formatPtr–>stringMatchProc provides the address of a procedure for Tk to call when it is searching for an image file format handler for suitable for reading data from a given string. *formatPtr–>stringMatchProc* must match the following prototype:

```
typedef int Tk_ImageStringMatchProc(
        char *string,
        char *formatString,
        int *widthPtr,
        int *heightPtr);
```

The *string* argument points to the string containing the image data. The *formatString* argument contains the value given for the **–format** option, or NULL if the option was not specified. If the data in the string appears to be in the format supported by this handler, the *formatPtr–>stringMatchProc* procedure should store the width and height of the image in **widthPtr* and **heightPtr* respectively, and return 1. Otherwise it should return 0.

FILEREADPROC

formatPtr–>fileReadProc provides the address of a procedure for Tk to call to read data from an image file into a photo image. *formatPtr–>fileReadProc* must match the following prototype:

```
typedef int Tk_ImageFileReadProc(
        Tcl_Interp *interp,
        Tcl_Channel chan,
        char *fileName,
        char *formatString,
        PhotoHandle imageHandle,
        int destX, int destY,
        int width, int height,
        int srcX, int srcY);
```

The *interp* argument is the interpreter in which the command was invoked to read the image; it should be used for reporting errors. The image data is in the file named *fileName*, which is open for reading as *chan*. The *formatString* argument contains the value given for the **–format** option, or NULL if the option was not specified. The image data in the file, or a subimage of it, is to be read into the photo image identified by the handle *imageHandle*. The subimage of the data in the file is of dimensions *width* x *height* and has its top–left corner at coordinates (*srcX*,*srcY*). It is to be stored in the photo image with its top–left corner at coordinates (*destX*,*destY*) using the **Tk_PhotoPutBlock** procedure. The return value is a standard Tcl return value.

STRINGREADPROC

formatPtr–>stringReadProc provides the address of a procedure for Tk to call to read data from a string into a photo image. *formatPtr–>stringReadProc* must match the following prototype:

```
typedef int Tk_ImageStringReadProc(
        Tcl_Interp *interp,
        char *string,
        char *formatString,
        PhotoHandle imageHandle,
```

```
        int destX, int destY,
        int width, int height,
        int srcX, int srcY);
```

The *interp* argument is the interpreter in which the command was invoked to read the image; it should be used for reporting errors. The *string* argument points to the image data in string form. The *formatString* argument contains the value given for the **–format** option, or NULL if the option was not specified. The image data in the string, or a subimage of it, is to be read into the photo image identified by the handle *imageHandle*. The subimage of the data in the string is of dimensions *width* x *height* and has its top–left corner at coordinates (*srcX*,*srcY*). It is to be stored in the photo image with its top–left corner at coordinates (*destX*,*destY*) using the **Tk_PhotoPutBlock** procedure. The return value is a standard Tcl return value.

FILEWRITEPROC

formatPtr–>fileWriteProc provides the address of a procedure for Tk to call to write data from a photo image to a file. *formatPtr–>fileWriteProc* must match the following prototype:

```
typedef int Tk_ImageFileWriteProc(
        Tcl_Interp *interp,
        char *fileName,
        char *formatString,
        Tk_PhotoImageBlock *blockPtr);
```

The *interp* argument is the interpreter in which the command was invoked to write the image; it should be used for reporting errors. The image data to be written are in memory and are described by the Tk_PhotoImageBlock structure pointed to by *blockPtr*; see the manual page FindPhoto(3) for details. The *fileName* argument points to the string giving the name of the file in which to write the image data. The *formatString* argument contains the value given for the **–format** option, or NULL if the option was not specified. The format string can contain extra characters after the name of the format. If appropriate, the *formatPtr–>fileWriteProc* procedure may interpret these characters to specify further details about the image file. The return value is a standard Tcl return value.

STRINGWRITEPROC

formatPtr–>stringWriteProc provides the address of a procedure for Tk to call to translate image data from a photo image into a string. *formatPtr–>stringWriteProc* must match the following prototype:

```
typedef int Tk_ImageStringWriteProc(
        Tcl_Interp *interp,
        Tcl_DString *dataPtr,
        char *formatString,
        Tk_PhotoImageBlock *blockPtr);
```

The *interp* argument is the interpreter in which the command was invoked to convert the image; it should be used for reporting errors. The image data to be converted are in memory and are described by the Tk_PhotoImageBlock structure pointed to by *blockPtr*; see the manual page FindPhoto(3) for details. The data for the string should be appended to the dynamic string given by *dataPtr*. The *formatString* argument contains the value given for the **–format** option, or NULL if the option was not specified. The format string can contain extra characters after the name of the format. If appropriate, the *formatPtr–>stringWriteProc* procedure may interpret these characters to specify further details about the image file. The return value is a standard Tcl return value.

SEE ALSO

Tk::FindPhoto, *Tk::FindPhoto*

KEYWORDS

photo image, image file

NAME

Tk_CreateSelHandler, Tk_DeleteSelHandler − arrange to handle requests for a selection

=for category C Programming

SYNOPSIS

#include <tk.h>

Tk_CreateSelHandler(*tkwin, selection, target, proc, clientData, format*)

Tk_DeleteSelHandler(*tkwin, selection, target*)

ARGUMENTS

Tk_Window tkwin (in)

Window for which *proc* will provide selection information.

Atom selection (in)

The name of the selection for which *proc* will provide selection information.

Atom target (in)

Form in which *proc* can provide the selection (e.g. STRING or FILE_NAME). Corresponds to *type* arguments in **selection** commands.

Tk_SelectionProc *proc (in)

Procedure to invoke whenever the selection is owned by *tkwin* and the selection contents are requested in the format given by *target*.

ClientData clientData (in)

Arbitrary one−word value to pass to *proc*.

Atom format (in)

If the selection requestor isn't in this process, *format* determines the representation used to transmit the selection to its requestor.

DESCRIPTION

Tk_CreateSelHandler arranges for a particular procedure (*proc*) to be called whenever *selection* is owned by *tkwin* and the selection contents are requested in the form given by *target*. *Target* should be one of the entries defined in the left column of Table 2 of the X Inter−Client Communication Conventions Manual (ICCCM) or any other form in which an application is willing to present the selection. The most common form is STRING.

Proc should have arguments and result that match the type **Tk_SelectionProc**:

```
typedef int Tk_SelectionProc(
        ClientData clientData,
        int offset,
        char *buffer,
        int maxBytes);
```

The *clientData* parameter to *proc* is a copy of the *clientData* argument given to **Tk_CreateSelHandler**. Typically, *clientData* points to a data structure containing application−specific information that is needed to retrieve the selection. *Offset* specifies an offset position into the selection, *buffer* specifies a location at which to copy information about the selection, and *maxBytes* specifies the amount of space available at *buffer*. *Proc* should place a NULL−terminated string at *buffer* containing *maxBytes* or fewer characters (not including the terminating NULL), and it should return a count of the number of non−NULL characters stored at *buffer*. If the selection no longer exists (e.g. it once existed but the user deleted the range of characters containing it), then *proc* should return −1.

When transferring large selections, Tk will break them up into smaller pieces (typically a few thousand bytes

each) for more efficient transmission. It will do this by calling *proc* one or more times, using successively higher values of *offset* to retrieve successive portions of the selection. If *proc* returns a count less than *maxBytes* it means that the entire remainder of the selection has been returned. If *proc*'s return value is *maxBytes* it means there may be additional information in the selection, so Tk must make another call to *proc* to retrieve the next portion.

Proc always returns selection information in the form of a character string. However, the ICCCM allows for information to be transmitted from the selection owner to the selection requestor in any of several formats, such as a string, an array of atoms, an array of integers, etc. The *format* argument to **Tk_CreateSelHandler** indicates what format should be used to transmit the selection to its requestor (see the middle column of Table 2 of the ICCCM for examples). If *format* is not STRING, then Tk will take the value returned by *proc* and divided it into fields separated by white space. If *format* is ATOM, then Tk will return the selection as an array of atoms, with each field in *proc*'s result treated as the name of one atom. For any other value of *format*, Tk will return the selection as an array of 32−bit values where each field of *proc*'s result is treated as a number and translated to a 32−bit value. In any event, the *format* atom is returned to the selection requestor along with the contents of the selection.

If **Tk_CreateSelHandler** is called when there already exists a handler for *selection* and *target* on *tkwin*, then the existing handler is replaced with a new one.

Tk_DeleteSelHandler removes the handler given by *tkwin*, *selection*, and *target*, if such a handler exists. If there is no such handler then it has no effect.

KEYWORDS

format, handler, selection, target

NAME

Tk_CreateWindow, Tk_CreateWindowFromPath, Tk_DestroyWindow, Tk_MakeWindowExist − create or delete window

=for category C Programming

SYNOPSIS

#include <tk.h>

Tk_Window **Tk_CreateWindow**(*interp, parent, name, topLevScreen*)

Tk_Window **Tk_CreateWindowFromPath**(*interp, tkwin, pathName, topLevScreen*)

Tk_DestroyWindow(*tkwin*)

Tk_MakeWindowExist(*tkwin*)

ARGUMENTS

Tcl_Interp *interp (out)

Tcl interpreter to use for error reporting. If no error occurs, then **interp* isn't modified.

Tk_Window parent (in)

Token for the window that is to serve as the logical parent of the new window.

char *name (in)

Name to use for this window. Must be unique among all children of the same *parent*.

char *topLevScreen (in)

Has same format as *screenName*. If NULL, then new window is created as an internal window. If non−NULL, new window is created as a top−level window on screen *topLevScreen*. If *topLevScreen* is an empty string (``'') then new window is created as top−level window of *parent*'s screen.

Tk_Window tkwin (in)

Token for window.

char *pathName (in)

Name of new window, specified as path name within application (e.g. **.a.b.c**).

DESCRIPTION

The procedures **Tk_CreateWindow** and **Tk_CreateWindowFromPath** are used to create new windows for use in Tk−based applications. Each of the procedures returns a token that can be used to manipulate the window in other calls to the Tk library. If the window couldn't be created successfully, then NULL is returned and *interp−>result* is modified to hold an error message.

Tk supports two different kinds of windows: internal windows and top−level windows. An internal window is an interior window of a Tk application, such as a scrollbar or menu bar or button. A top−level window is one that is created as a child of a screen's root window, rather than as an interior window, but which is logically part of some existing main window. Examples of top−level windows are pop−up menus and dialog boxes.

New windows may be created by calling **Tk_CreateWindow**. If the *topLevScreen* argument is NULL, then the new window will be an internal window. If *topLevScreen* is non−NULL, then the new window will be a top−level window: *topLevScreen* indicates the name of a screen and the new window will be created as a child of the root window of *topLevScreen*. In either case Tk will consider the new window to be the logical child of *parent*: the new window's path name will reflect this fact, options may be specified for the new window under this assumption, and so on. The only difference is that new X window for a top−level window will not be a child of *parent*'s X window. For example, a pull−down menu's *parent* would be the button−like window used to invoke it, which would in turn be a child of the menu bar window. A dialog box might have the application's main window as its parent.

Tk_CreateWindowFromPath offers an alternate way of specifying new windows. In **Tk_CreateWindowFromPath** the new window is specified with a token for any window in the target application (*tkwin*), plus a path name for the new window. It produces the same effect as **Tk_CreateWindow** and allows both top−level and internal windows to be created, depending on the value of *topLevScreen*. In calls to **Tk_CreateWindowFromPath**, as in calls to **Tk_CreateWindow**, the parent of the new window must exist at the time of the call, but the new window must not already exist.

The window creation procedures don't actually issue the command to X to create a window. Instead, they create a local data structure associated with the window and defer the creation of the X window. The window will actually be created by the first call to **Tk_MapWindow**. Deferred window creation allows various aspects of the window (such as its size, background color, etc.) to be modified after its creation without incurring any overhead in the X server. When the window is finally mapped all of the window attributes can be set while creating the window.

The value returned by a window−creation procedure is not the X token for the window (it can't be, since X hasn't been asked to create the window yet). Instead, it is a token for Tk's local data structure for the window. Most of the Tk library procedures take Tk_Window tokens, rather than X identifiers. The actual X window identifier can be retrieved from the local data structure using the **Tk_WindowId** macro; see the manual entry for **Tk_WindowId** for details.

Tk_DestroyWindow deletes a window and all the data structures associated with it, including any event handlers created with **Tk_CreateEventHandler**. In addition, **Tk_DestroyWindow** will delete any children of *tkwin* recursively (where children are defined in the Tk sense, consisting of all windows that were created with the given window as *parent*). If *tkwin* was created by **Tk_CreateInternalWindow** then event handlers interested in destroy events are invoked immediately. If *tkwin* is a top−level or main window, then the event handlers will be invoked later, after X has seen the request and returned an event for it.

If a window has been created but hasn't been mapped, so no X window exists, it is possible to force the creation of the X window by calling **Tk_MakeWindowExist**. This procedure issues the X commands to instantiate the window given by *tkwin*.

KEYWORDS

create, deferred creation, destroy, display, internal window, screen, top−level window, window

NAME

Tk_DeleteImage – Destroy an image.

=for category C Programming

SYNOPSIS

#include <tk.h>

Tk_DeleteImage(*interp, name*)

ARGUMENTS

Tcl_Interp *interp (in)

Interpreter for which the image was created.

char *name (in)

Name of the image.

DESCRIPTION

Tk_DeleteImage deletes the image given by *interp* and *name*, if there is one. All instances of that image will redisplay as empty regions. If the given image does not exist then the procedure has no effect.

KEYWORDS

delete image, image manager

NAME

Tk_DoOneEvent, Tk_MainLoop, Tk_HandleEvent − wait for events and invoke event handlers

=for category C Programming

SYNOPSIS

#include <tk.h>

int **Tk_DoOneEvent**(*flags*)

Tk_MainLoop( )

Tk_HandleEvent(*eventPtr*)

ARGUMENTS

int flags (in)

This parameter is normally zero. It may be an OR−ed combination of any of the following flag bits: TK_X_EVENTS, TK_FILE_EVENTS, TK_TIMER_EVENTS, TK_IDLE_EVENTS, TK_ALL_EVENTS, or TK_DONT_WAIT.

XEvent *eventPtr (in)

Pointer to X event to dispatch to relevant handler(s).

DESCRIPTION

These three procedures are responsible for waiting for events and dispatching to event handlers created with the procedures **Tk_CreateEventHandler**, **Tk_CreateFileHandler**, **Tk_CreateTimerHandler**, and **Tk_DoWhenIdle**. **Tk_DoOneEvent** is the key procedure. It waits for a single event of any sort to occur, invokes the handler(s) for that event, and then returns. **Tk_DoOneEvent** first checks for X events and file−related events; if one is found then it calls the handler(s) for the event and returns. If there are no X or file events pending, then **Tk_DoOneEvent** checks to see if timer callbacks are ready; if so, it makes a single callback and returns. If no timer callbacks are ready, **Tk_DoOneEvent** checks for **Tk_DoWhenIdle** callbacks; if any are found, it invokes all of them and returns. Finally, if no events or work have been found, **Tk_DoOneEvent** sleeps until a timer, file, or X event occurs; then it processes the first event found (in the order given above) and returns. The normal return value is 1 to signify that some event or callback was processed. If no event or callback is processed (under various conditions described below), then 0 is returned.

If the *flags* argument to **Tk_DoOneEvent** is non−zero then it restricts the kinds of events that will be processed by **Tk_DoOneEvent**. *Flags* may be an OR−ed combination of any of the following bits:

TK_X_EVENTS −

Process X events.

TK_FILE_EVENTS −

Process file events.

TK_TIMER_EVENTS −

Process timer events.

TK_IDLE_EVENTS −

Process **Tk_DoWhenIdle** callbacks.

TK_ALL_EVENTS −

Process all kinds of events: equivalent to OR−ing together all of the above flags or specifying none of them.

TK_DONT_WAIT −

Don‘t sleep: process only events that are ready at the time of the call.

If any of the flags **TK_X_EVENTS**, **TK_FILE_EVENTS**, **TK_TIMER_EVENTS**, or **TK_IDLE_EVENTS** is set, then the only events that will be considered are those for which flags are set. Setting none of these flags is equivalent to the value **TK_ALL_EVENTS**, which causes all event types to be processed.

The **TK_DONT_WAIT** flag causes **Tk_DoWhenIdle** not to put the process to sleep: it will check for events but if none are found then it returns immediately with a return value of 0 to indicate that no work was done. **Tk_DoOneEvent** will also return 0 without doing anything if *flags* is **TK_IDLE_EVENTS** and there are no **Tk_DoWhenIdle** callbacks pending. Lastly, **Tk_DoOneEvent** will return 0 without doing anything if there are no events or work found and if there are no files, displays, or timer handlers to wait for.

Tk_MainLoop is a procedure that loops repeatedly calling **Tk_DoOneEvent**. It returns only when there are no applications left in this process (i.e. no main windows exist anymore). Most X applications will call **Tk_MainLoop** after initialization; the main execution of the application will consist entirely of callbacks invoked by **Tk_DoOneEvent**.

Tk_HandleEvent is a lower−level procedure invoked by **Tk_DoOneEvent**. It makes callbacks to any event handlers (created by calls to **Tk_CreateEventHandler**) that match *eventPtr* and then returns. In some cases it may be useful for an application to read events directly from X and dispatch them by calling **Tk_HandleEvent**, without going through the additional mechanism provided by **Tk_DoOneEvent**.

These procedures may be invoked recursively. For example, it is possible to invoke **Tk_DoOneEvent** recursively from a handler called by **Tk_DoOneEvent**. This sort of operation is useful in some modal situations, such as when a notifier has been popped up and an application wishes to wait for the user to click a button in the notifier before doing anything else.

KEYWORDS

callback, event, handler, idle, timer

NAME

Tk_DoWhenIdle, Tk_CancelIdleCall − invoke a procedure when there are no pending events

=for category C Programming

SYNOPSIS

#include <tk.h>

Tk_DoWhenIdle(*proc, clientData*)

Tk_CancelIdleCall(*proc, clientData*)

ARGUMENTS

Tk_IdleProc *proc (in)

Procedure to invoke.

ClientData clientData (in)

Arbitrary one−word value to pass to *proc*.

DESCRIPTION

Tk_DoWhenIdle arranges for *proc* to be invoked when the application becomes idle. The application is considered to be idle when **Tk_DoOneEvent** has been called, it couldn't find any events to handle, and it is about to go to sleep waiting for an event to occur. At this point all pending **Tk_DoWhenIdle** handlers are invoked. For each call to **Tk_DoWhenIdle** there will be a single call to *proc*; after *proc* is invoked the handler is automatically removed. **Tk_DoWhenIdle** is only useable in programs that use **Tk_DoOneEvent** to dispatch events.

Proc should have arguments and result that match the type **Tk_IdleProc**:

typedef void Tk_IdleProc(ClientData *clientData*);

The *clientData* parameter to *proc* is a copy of the *clientData* argument given to **Tk_DoWhenIdle**. Typically, *clientData* points to a data structure containing application−specific information about what *proc* should do.

Tk_CancelIdleCall may be used to cancel one or more previous calls to **Tk_DoWhenIdle**: if there is a **Tk_DoWhenIdle** handler registered for *proc* and *clientData*, then it is removed without invoking it. If there is more than one handler on the idle list that refers to *proc* and *clientData*, all of the handlers are removed. If no existing handlers match *proc* and *clientData* then nothing happens.

Tk_DoWhenIdle is most useful in situations where (a) a piece of work will have to be done but (b) it's possible that something will happen in the near future that will change what has to be done, or require something different to be done. **Tk_DoWhenIdle** allows the actual work to be deferred until all pending events have been processed. At this point the exact work to be done will presumably be known and it can be done exactly once.

For example, **Tk_DoWhenIdle** might be used by an editor to defer display updates until all pending commands have been processed. Without this feature, redundant redisplays might occur in some situations, such as the processing of a command file.

KEYWORDS

callback, defer, handler, idle

NAME

Tk_DrawFocusHighlight – draw the traversal highlight ring for a widget

=for category C Programming

SYNOPSIS

#include <tk.h>

Tk_GetPixels(*tkwin, gc, width, drawable***)**

ARGUMENTS

Tk_Window tkwin (in)

Window for which the highlight is being drawn. Used to retrieve the window's dimensions, among other things.

GC gc (in)

Graphics context to use for drawing the highlight.

int width (in)

Width of the highlight ring, in pixels.

Drawable drawable (in)

Drawable in which to draw the highlight; usually an offscreen pixmap for double buffering.

DESCRIPTION

Tk_DrawFocusHighlight is a utility procedure that draws the traversal highlight ring for a widget. It is typically invoked by widgets during redisplay.

KEYWORDS

focus, traversal highlight

NAME

Tk_CreateEventHandler, Tk_DeleteEventHandler − associate procedure callback with an X event

=for category C Programming

SYNOPSIS

#include <tk.h>

Tk_CreateEventHandler(*tkwin, mask, proc, clientData*)

Tk_DeleteEventHandler(*tkwin, mask, proc, clientData*)

ARGUMENTS

Tk_Window tkwin (in)

Token for window in which events may occur.

"unsigned long" mask (in)

Bit−mask of events (such as **ButtonPressMask**) for which *proc* should be called.

Tk_EventProc *proc (in)

Procedure to invoke whenever an event in *mask* occurs in the window given by *tkwin*.

ClientData clientData (in)

Arbitrary one−word value to pass to *proc*.

DESCRIPTION

Tk_CreateEventHandler arranges for *proc* to be invoked in the future whenever one of the event types specified by *mask* occurs in the window specified by *tkwin*. The callback to *proc* will be made by **Tk_HandleEvent**; this mechanism only works in programs that dispatch events through **Tk_HandleEvent** (or through other Tk procedures that call **Tk_HandleEvent**, such as **Tk_DoOneEvent** or **Tk_MainLoop**).

Proc should have arguments and result that match the type **Tk_EventProc**:

```
typedef void Tk_EventProc(
        ClientData clientData,
        XEvent *eventPtr);
```

The *clientData* parameter to *proc* is a copy of the *clientData* argument given to **Tk_CreateEventHandler** when the callback was created. Typically, *clientData* points to a data structure containing application−specific information about the window in which the event occurred. *EventPtr* is a pointer to the X event, which will be one of the ones specified in the *mask* argument to **Tk_CreateEventHandler**.

Tk_DeleteEventHandler may be called to delete a previously−created event handler: it deletes the first handler it finds that is associated with *tkwin* and matches the *mask*, *proc*, and *clientData* arguments. If no such handler exists, then **Tk_EventHandler** returns without doing anything. Although Tk supports it, it's probably a bad idea to have more than one callback with the same *mask*, *proc*, and *clientData* arguments. When a window is deleted all of its handlers will be deleted automatically; in this case there is no need to call **Tk_DeleteEventHandler**.

If multiple handlers are declared for the same type of X event on the same window, then the handlers will be invoked in the order they were created.

KEYWORDS

bind, callback, event, handler

NAME

Tk_EventInit – Use the Tk event loop without the rest of Tk

=for category C Programming

SYNOPSIS

#include <tk.h>

int **Tk_EventInit**(*interp*)

ARGUMENTS

Tcl_Interp *interp (in)

Interpreter in which event–related Tk commands are to be created.

DESCRIPTION

This procedure may be invoked to use the Tk event management functions without the rest of Tk (e.g., in applications that do not have access to a display). **Tk_EventInit** creates the **after** and **fileevent** commands in *interp*. It also creates versions of the **tkwait** and **update** commands with reduced functionality: the **tkwait** command supports only the **variable** option, not **visibility** or `$widget`, and **update** does not check for X events. **Tk_EventInit** always returns **TCL_OK** to signal that it completed successfully.

The event–management procedures in Tk are divided into two groups, those that can be used stand–alone and those that require the full Tk library to be present. The following procedures may be used stand–alone: **Tk_CreateFileHandler**, **Tk_CreateFileHandler2**, **Tk_DeleteFileHandler**, **Tk_CreateTimerHandler**, **Tk_DeleteTimerHandler**, **Tk_DoWhenIdle**, **Tk_CancelIdleCall**, **Tk_DoOneEvent**, **Tk_Sleep**, and **Tk_BackgroundError**. Note that **Tk_MainLoop** cannot be used without the full Tk library, since it checks to see if windows are still open. If an application uses the event procedures stand–alone, it must include its own main loop that invokes **Tk_DoOneEvent** repeatedly.

Tk_EventInit is typically called from an application's **Tcl_AppInit** procedure; it should not be invoked in applications that use the full Tk library (e.g., those that have already invoked **Tk_CreateMainWindow**). However, it is OK for an application to start up using **Tk_EventInit**, compute without X for a while, and later invoke **Tk_CreateMainWindow**. When **Tk_CreateMainWindow** is invoked, the full suite of windowing Tcl commands will become available, and the full–blown versions of **tkwait** and **update** will replace the abridged versions created with **Tk_EventInit**.

KEYWORDS

event management, Tcl_AppInit

NAME

Tk_CreateFileHandler, Tk_CreateFileHandler2, Tk_DeleteFileHandler − associate procedure callbacks with files or devices

=for category C Programming

SYNOPSIS

#include <tk.h>

Tk_CreateFileHandler(*id, mask, proc, clientData*)

Tk_CreateFileHandler2(*id, proc2, clientData*)

Tk_DeleteFileHandler(*id*)

ARGUMENTS

int id (in)

Integer identifier for an open file or device (such as returned by **open** system call).

int mask (in)

Conditions under which *proc* should be called: OR−ed combination of **TK_READABLE**, **TK_WRITABLE**, and **TK_EXCEPTION**.

Tk_FileProc *proc (in)

Procedure to invoke whenever the file or device indicated by *id* meets the conditions specified by *mask*.

Tk_FileProc2 *proc2 (in)

Procedure to invoke from event loop to check whether *fd* is ready and, if so, handle it.

ClientData clientData (in)

Arbitrary one−word value to pass to *proc*.

DESCRIPTION

Tk_CreateFileHandler arranges for *proc* to be invoked in the future whenever I/O becomes possible on a file or an exceptional condition exists for the file. The file is indicated by *id*, and the conditions of interest are indicated by *mask*. For example, if *mask* is **TK_READABLE**, *proc* will be called when the file is readable. The callback to *proc* is made by **Tk_DoOneEvent**, so **Tk_CreateFileHandler** is only useful in programs that dispatch events through **Tk_DoOneEvent** or through other Tk procedures that call **Tk_DoOneEvent**, such as **Tk_MainLoop**.

Proc should have arguments and result that match the type **Tk_FileProc**:

typedef void Tk_FileProc(

ClientData *clientData*, int *mask*);

The *clientData* parameter to *proc* is a copy of the *clientData* argument given to **Tk_CreateFileHandler** when the callback was created. Typically, *clientData* points to a data structure containing application−specific information about the file. *Mask* is an integer mask indicating which of the requested conditions actually exists for the file; it will contain a subset of the bits in the *mask* argument to **Tk_CreateFileHandler**.

Tk_CreateFileHandler2 also creates a file handler, but it provides a lower−level and more flexible interface. The callback procedure *proc2* must have arguments and result that match the following prototype:

typedef int Tk_FileProc2(

ClientData *clientData*, int *mask*, int *flags*);

Whereas a file handler created by **Tk_CreateFileHandler** is only invoked when the file is known to be

''ready'', a file handler created by **Tk_CreateFileHandler2** is invoked on every pass through the the event loop (**Tk_DoWhenIdle**); it gets to determine whether the file is ''ready'' or not. The *mask* argument contains an OR'ed combination of the bits **TK_READABLE**, **TK_WRITABLE**, and **TK_EXCEPTION**, which indicate whether the file is known to be readable, writable, or to have an exceptional condition present (this is the case if **select** has been invoked since the previous call to *proc2*, and if it indicated that the specified conditions were present). *proc2* may use this information along with additional information of its own, such as knowledge about buffered data, to decide whether the file is really ''ready''. The *flags* argument is a copy of the flags passed to **Tk_DoOneEvent**, which may be used by *proc2* to ignore the file if the appropriate bit, such as **TK_FILE_EVENTS**, is not present.

proc2 must return an integer value that is either **TK_FILE_HANDLED** or an OR−ed combination of **TK_READABLE**, **TK_WRITABLE**, and **TK_EXCEPTION**. If the return value is **TK_FILE_HANDLED** it means that the file was ''ready'' and that *proc2* handled the ready condition; **Tk_DoOneEvent** will return immediately. If the return value is not **TK_FILE_HANDLED**, then it indicates the set of conditions that should be checked for the file if the current invocation of **Tk_DoWhenIdle** invokes **select**. Typically the return value reflects all of the conditions that *proc2* cares about. A zero return value means that the file should be ignored if **Tk_DoWhenIdle** calls **select** (this could happen, for example, if the *flags* argument specified that this file's events should be ignored). The value returned by *proc2* only affects a **select** call from the current invocation of **Tk_DoOneEvent**; the next invocation of **Tk_DoOneEvent** will call *proc2* afresh to get new information.

There may exist only one handler for a given file at a given time. If **Tk_CreateFileHandler** or **Tk_CreateFileHandler2** is called when a handler already exists for *id*, then the new callback replaces the information that was previously recorded.

Tk_DeleteFileHandler may be called to delete the file handler for *id*; if no handler exists for the file given by *id* then the procedure has no effect.

The purpose of file handlers is to enable an application to respond to X events and other events while waiting for files to become ready for I/O. For this to work correctly, the application may need to use non−blocking I/O operations on the files for which handlers are declared. Otherwise the application may be put to sleep if it reads or writes too much data; while waiting for the I/O to complete the application won't be able to service other events. In BSD−based UNIX systems, non−blocking I/O can be specified for a file using the **fcntl** kernel call with the **FNDELAY** flag.

KEYWORDS

callback, file, handler

NAME

Tk_FindPhoto, Tk_PhotoPutBlock, Tk_PhotoPutZoomedBlock, Tk_PhotoGetImage, Tk_PhotoBlank, Tk_PhotoExpand, Tk_PhotoGetSize, Tk_PhotoSetSize − manipulate the image data stored in a photo image.

=for category C Programming

SYNOPSIS

#include <tk.h> #include <tkPhoto.h>

Tk_PhotoHandle **Tk_FindPhoto**(*interp, imageName*)

void **Tk_PhotoPutBlock**(*handle, blockPtr, x, y, width, height*)

void **Tk_PhotoPutZoomedBlock**(*handle, blockPtr, x, y, width, height,*\ zoomX, zoomY, subsampleX, subsampleY)

int **Tk_PhotoGetImage**(*handle, blockPtr*)

void **Tk_PhotoBlank**(*handle*)

void **Tk_PhotoExpand**(*handle, width, height*)

void **Tk_PhotoGetSize**(*handle, widthPtr, heightPtr*)

void **Tk_PhotoSetSize**(*handle, width, height*)

ARGUMENTS

Tcl_Interp *interp (in)

Interpreter in which image was created.

char *imageName (in)

Name of the photo image.

Tk_PhotoHandle handle (in)

Opaque handle identifying the photo image to be affected.

Tk_PhotoImageBlock *blockPtr (in)

Specifies the address and storage layout of image data.

int x (in)

Specifies the X coordinate where the top−left corner of the block is to be placed within the image.

int y (in)

Specifies the Y coordinate where the top−left corner of the block is to be placed within the image.

int width (in)

Specifies the width of the image area to be affected (for **Tk_PhotoPutBlock**) or the desired image width (for **Tk_PhotoExpand** and **Tk_PhotoSetSize**).

int height (in)

Specifies the height of the image area to be affected (for **Tk_PhotoPutBlock**) or the desired image height (for **Tk_PhotoExpand** and **Tk_PhotoSetSize**).

int *widthPtr (out)

Pointer to location in which to store the image width.

int *heightPtr (out)

Pointer to location in which to store the image height.

int subsampleX (in)

Specifies the subsampling factor in the X direction for input image data.

int subsampleY (in)

Specifies the subsampling factor in the Y direction for input image data.

int zoomX (in)

Specifies the zoom factor to be applied in the X direction to pixels being written to the photo image.

int zoomY (in)

Specifies the zoom factor to be applied in the Y direction to pixels being written to the photo image.

DESCRIPTION

Tk_FindPhoto returns an opaque handle that is used to identify a particular photo image to the other procedures. The parameter is the name of the image, that is, the name specified to the **image create** photo command, or assigned by that command if no name was specified.

Tk_PhotoPutBlock is used to supply blocks of image data to be displayed. The call affects an area of the image of size *width* x *height* pixels, with its top−left corner at coordinates (*x*,*y*). All of *width*, *height*, *x*, and *y* must be non−negative. If part of this area lies outside the current bounds of the image, the image will be expanded to include the area, unless the user has specified an explicit image size with the **−width** and/or **−height** widget configuration options (see photo(n)); in that case the area is silently clipped to the image boundaries.

The *block* parameter is a pointer to a **Tk_PhotoImageBlock** structure, defined as follows:

```
typedef struct {
        unsigned char *pixelPtr;
        int width;
        int height;
        int pitch;
        int pixelSize;
        int offset[3];
} Tk_PhotoImageBlock;
```

The *pixelPtr* field points to the first pixel, that is, the top−left pixel in the block. The *width* and *height* fields specify the dimensions of the block of pixels. The *pixelSize* field specifies the address difference between two horizontally adjacent pixels. Often it is 3 or 4, but it can have any value. The *pitch* field specifies the address difference between two vertically adjacent pixels. The *offset* array contains the offsets from the address of a pixel to the addresses of the bytes containing the red, green and blue components. These are normally 0, 1 and 2, but can have other values, e.g., for images that are stored as separate red, green and blue planes.

The value given for the *width* and *height* parameters to **Tk_PhotoPutBlock** do not have to correspond to the values specified in *block*. If they are smaller, **Tk_PhotoPutBlock** extracts a sub−block from the image data supplied. If they are larger, the data given are replicated (in a tiled fashion) to fill the specified area. These rules operate independently in the horizontal and vertical directions.

Tk_PhotoPutZoomedBlock works like **Tk_PhotoPutBlock** except that the image can be reduced or enlarged for display. The *subsampleX* and *subsampleY* parameters allow the size of the image to be reduced by subsampling. **Tk_PhotoPutZoomedBlock** will use only pixels from the input image whose X coordinates are multiples of *subsampleX*, and whose Y coordinates are multiples of *subsampleY*. For example, an image of 512x512 pixels can be reduced to 256x256 by setting *subsampleX* and *subsampleY* to 2.

The *zoomX* and *zoomY* parameters allow the image to be enlarged by pixel replication. Each pixel of the (possibly subsampled) input image will be written to a block *zoomX* pixels wide and *zoomY* pixels high of the displayed image. Subsampling and zooming can be used together for special effects.

Tk_PhotoGetImage can be used to retrieve image data from a photo image. **Tk_PhotoGetImage** fills in the structure pointed to by the *blockPtr* parameter with values that describe the address and layout of the image data that the photo image has stored internally. The values are valid until the image is destroyed or its size is changed. **Tk_PhotoGetImage** returns 1 for compatibility with the corresponding procedure in the old photo widget.

Tk_PhotoBlank blanks the entire area of the photo image. Blank areas of a photo image are transparent.

Tk_PhotoExpand requests that the widget's image be expanded to be at least *width* x *height* pixels in size. The width and/or height are unchanged if the user has specified an explicit image width or height with the **−width** and/or **−height** configuration options, respectively. If the image data are being supplied in many small blocks, it is more efficient to use **Tk_PhotoExpand** or **Tk_PhotoSetSize** at the beginning rather than allowing the image to expand in many small increments as image blocks are supplied.

Tk_PhotoSetSize specifies the size of the image, as if the user had specified the given *width* and *height* values to the **−width** and **−height** configuration options. A value of zero for *width* or *height* does not change the image's width or height, but allows the width or height to be changed by subsequent calls to **Tk_PhotoPutBlock**, **Tk_PhotoPutZoomedBlock** or **Tk_PhotoExpand**.

Tk_PhotoGetSize returns the dimensions of the image in **widthPtr* and **heightPtr*.

CREDITS

The code for the photo image type was developed by Paul Mackerras, based on his earlier photo widget code.

KEYWORDS

photo, image

NAME

Tk_FontId, Tk_FontMetrics, Tk_PostscriptFontName − accessor functions for fonts

=for category C Programming

SYNOPSIS

#include <tk.h>

Font **Tk_FontId**(*tkfont*)

void **Tk_GetFontMetrics**(*tkfont, fmPtr*)

int **Tk_PostscriptFontName**(*tkfont, dsPtr*)

ARGUMENTS

Tk_Font tkfont (in)

Opaque font token being queried. Must have been returned by a previous call to **Tk_GetFont**.

Tk_FontMetrics *fmPtr (out)

Pointer to structure in which the font metrics for *tkfont* will be stored.

Tcl_DString *dsPtr (out)

Pointer to an initialized **Tcl_DString** to which the name of the Postscript font that corresponds to *tkfont* will be appended.

DESCRIPTION

Given a *tkfont*, **Tk_FontId** returns the token that should be selected into an XGCValues structure in order to construct a graphics context that can be used to draw text in the specified font.

Tk_GetFontMetrics computes the ascent, descent, and linespace of the *tkfont* in pixels and stores those values in the structure pointer to by *fmPtr*. These values can be used in computations such as to space multiple lines of text, to align the baselines of text in different fonts, and to vertically align text in a given region. See the documentation for the **font** command for definitions of the terms ascent, descent, and linespace, used in font metrics.

Tk_PostscriptFontName maps a *tkfont* to the corresponding Postcript font name that should be used when printing. The return value is the size in points of the *tkfont* and the Postscript font name is appended to *dsPtr*. *DsPtr* must refer to an initialized **Tcl_DString**. Given a ‘‘reasonable‘‘ Postscript printer, the following screen font families should print correctly:

Avant Garde, **Arial**, **Bookman**, **Courier**, **Courier New**, **Geneva**, **Helvetica**, **Monaco**, **New Century Schoolbook**, **New York**, **Palatino**, **Symbol**, **Times**, **Times New Roman**, **Zapf Chancery**, and **Zapf Dingbats**.

Any other font families may not print correctly because the computed Postscript font name may be incorrect or not exist on the printer.

DATA STRUCTURES

The Tk_FontMetrics data structure is used by Tk_GetFontMetrics to return information about a font and is defined as follows:

```
typedef struct Tk_FontMetrics {
    int ascent;
    int descent;
    int linespace;
} Tk_FontMetrics;
```

The *linespace* field is the amount in pixels that the tallest letter sticks up above the baseline, plus any extra blank space added by the designer of the font.

The *descent* is the largest amount in pixels that any letter sticks below the baseline, plus any extra blank space added by the designer of the font.

The *linespace* is the sum of the ascent and descent. How far apart two lines of text in the same font should be placed so that none of the characters in one line overlap any of the characters in the other line.

KEYWORDS

font

NAME

Tk_FreeXId − make X resource identifier available for reuse

=for category C Programming

SYNOPSIS

#include <tk.h>

Tk_FreeXId(*display, id***)**

ARGUMENTS

Display *display (in)

Display for which *id* was allocated.

XID id (in)

Identifier of X resource (window, font, pixmap, cursor, graphics context, or colormap) that is no longer in use.

DESCRIPTION

The default allocator for resource identifiers provided by Xlib is very simple−minded and does not allow resource identifiers to be re−used. If a long−running application reaches the end of the resource id space, it will generate an X protocol error and crash. Tk replaces the default id allocator with its own allocator, which allows identifiers to be reused. In order for this to work, **Tk_FreeXId** must be called to tell the allocator about resources that have been freed. Tk automatically calls **Tk_FreeXId** whenever it frees a resource, so if you use procedures like **Tk_GetFontStruct**, **Tk_GetGC**, and **Tk_GetPixmap** then you need not call **Tk_FreeXId**. However, if you allocate resources directly from Xlib, for example by calling **XCreatePixmap**, then you should call **Tk_FreeXId** when you call the corresponding Xlib free procedure, such as **XFreePixmap**. If you don't call **Tk_FreeXId** then the resource identifier will be lost, which could cause problems if the application runs long enough to lose all of the available identifiers.

KEYWORDS

resource identifier

NAME

Tk_GeometryRequest, Tk_SetInternalBorder − specify desired geometry or internal border for a window

=for category C Programming

SYNOPSIS

#include <tk.h>

Tk_GeometryRequest(*tkwin, reqWidth, reqHeight*)

Tk_SetInternalBorder(*tkwin, width*)

ARGUMENTS

Tk_Window tkwin (in)

Window for which geometry is being requested.

int reqWidth (in)

Desired width for *tkwin*, in pixel units.

int reqHeight (in)

Desired height for *tkwin*, in pixel units.

int width (in)

Space to leave for internal border for *tkwin*, in pixel units.

DESCRIPTION

Tk_GeometryRequest is called by widget code to indicate its preference for the dimensions of a particular window. The arguments to **Tk_GeometryRequest** are made available to the geometry manager for the window, which then decides on the actual geometry for the window. Although geometry managers generally try to satisfy requests made to **Tk_GeometryRequest**, there is no guarantee that this will always be possible. Widget code should not assume that a geometry request will be satisfied until it receives a **ConfigureNotify** event indicating that the geometry change has occurred. Widget code should never call procedures like **Tk_ResizeWindow** directly. Instead, it should invoke **Tk_GeometryRequest** and leave the final geometry decisions to the geometry manager.

If *tkwin* is a top−level window, then the geometry information will be passed to the window manager using the standard ICCCM protocol.

Tk_SetInternalBorder is called by widget code to indicate that the widget has an internal border. This means that the widget draws a decorative border inside the window instead of using the standard X borders, which are external to the window‘s area. For example, internal borders are used to draw 3−D effects. *Width* specifies the width of the border in pixels. Geometry managers will use this information to avoid placing any children of *tkwin* overlapping the outermost *width* pixels of *tkwin*‘s area.

The information specified in calls to **Tk_GeometryRequest** and **Tk_SetInternalBorder** can be retrieved using the macros **Tk_ReqWidth**, **Tk_ReqHeight**, and **Tk_InternalBorderWidth**. See the *Tk::WindowId* documentation for details.

KEYWORDS

geometry, request

NAME

Tk_GetAnchor, Tk_NameOfAnchor − translate between strings and anchor positions

=for category C Programming

SYNOPSIS

#include <tk.h>

int **Tk_GetAnchor(***interp, string, anchorPtr***)**

char * **Tk_NameOfAnchor(***anchor***)**

ARGUMENTS

Tcl_Interp *interp (in)

Interpreter to use for error reporting.

char *string (in)

String containing name of anchor point: one of ‘‘n‘‘, ‘‘ne‘‘, ‘‘e‘‘, ‘‘se‘‘, ‘‘s‘‘, ‘‘sw‘‘, ‘‘w‘‘, ‘‘nw‘‘, or ‘‘center‘‘.

int *anchorPtr (out)

Pointer to location in which to store anchor position corresponding to *string*.

Tk_Anchor anchor (in)

Anchor position, e.g. **TCL_ANCHOR_CENTER**.

DESCRIPTION

Tk_GetAnchor places in **anchorPtr* an anchor position (enumerated type **Tk_Anchor**) corresponding to *string*, which will be one of **TK_ANCHOR_N**, **TK_ANCHOR_NE**, **TK_ANCHOR_E**, **TK_ANCHOR_SE**, **TK_ANCHOR_S**, **TK_ANCHOR_SW**, **TK_ANCHOR_W**, **TK_ANCHOR_NW**, or **TK_ANCHOR_CENTER**. Anchor positions are typically used for indicating a point on an object that will be used to position that object, e.g. **TK_ANCHOR_N** means position the top center point of the object at a particular place.

Under normal circumstances the return value is **TCL_OK** and *interp* is unused. If *string* doesn‘t contain a valid anchor position or an abbreviation of one of these names, then an error message is stored in *interp−>result*, **TCL_ERROR** is returned, and **anchorPtr* is unmodified.

Tk_NameOfAnchor is the logical inverse of **Tk_GetAnchor**. Given an anchor position such as **TK_ANCHOR_N** it returns a statically−allocated string corresponding to *anchor*. If *anchor* isn‘t a legal anchor value, then ‘‘unknown anchor position‘‘ is returned.

KEYWORDS

anchor position

NAME

Tk_GetBitmap, Tk_DefineBitmap, Tk_NameOfBitmap, Tk_SizeOfBitmap, Tk_FreeBitmap, Tk_GetBitmapFromData − maintain database of single−plane pixmaps

=for category C Programming

SYNOPSIS

#include <tk.h>

Pixmap **Tk_GetBitmap(***interp, tkwin, id***)**

int **Tk_DefineBitmap(***interp, nameId, source, width, height***)**

Tk_Uid **Tk_NameOfBitmap(***display, bitmap***)**

Tk_SizeOfBitmap(*display, bitmap, widthPtr, heightPtr***)**

Tk_FreeBitmap(*display, bitmap***)**

ARGUMENTS

Tcl_Interp *interp (in)

Interpreter to use for error reporting.

Tk_Window tkwin (in)

Token for window in which the bitmap will be used.

Tk_Uid id (in)

Description of bitmap; see below for possible values.

Tk_Uid nameId (in)

Name for new bitmap to be defined.

char *source (in)

Data for bitmap, in standard bitmap format. Must be stored in static memory whose value will never change.

"int" width (in)

Width of bitmap.

"int" height (in)

Height of bitmap.

"int" *widthPtr (out)

Pointer to word to fill in with *bitmap*'s width.

"int" *heightPtr (out)

Pointer to word to fill in with *bitmap*'s height.

Display *display (in)

Display for which *bitmap* was allocated.

Pixmap bitmap (in)

Identifier for a bitmap allocated by **Tk_GetBitmap**.

DESCRIPTION

These procedures manage a collection of bitmaps (one−plane pixmaps) being used by an application. The procedures allow bitmaps to be re−used efficiently, thereby avoiding server overhead, and also allow bitmaps to be named with character strings.

Tk_GetBitmap takes as argument a Tk_Uid describing a bitmap. It returns a Pixmap identifier for a bitmap corresponding to the description. It re−uses an existing bitmap, if possible, and creates a new one otherwise.

At present, *id* must have one of the following forms:

@fileName

FileName must be the name of a file containing a bitmap description in the standard X11 or X10 format.

name

Name must be the name of a bitmap defined previously with a call to **Tk_DefineBitmap**. The following names are pre−defined by Tk:

error

The international "don't" symbol: a circle with a diagonal line across it.

gray75

75% gray: a checkerboard pattern where three out of four bits are on.

gray50

50% gray: a checkerboard pattern where every other bit is on.

gray25

25% gray: a checkerboard pattern where one out of every four bits is on.

gray12

12.5% gray: a pattern where one−eighth of the bits are on, consisting of every fourth pixel in every other row.

hourglass

An hourglass symbol.

info A large letter ''i''.

questhead

The silhouette of a human head, with a question mark in it.

question

A large question−mark.

warning

A large exclamation point.

In addition, the following pre−defined names are available only on the **Macintosh** platform:

document

A generic document.

stationery

Document stationery.

edition

The *edition* symbol.

application

Generic application icon.

accessory

A desk accessory.

folder

Generic folder icon.

pfolder
A locked folder.

trash
A trash can.

floppy
A floppy disk.

ramdisk
A floppy disk with chip.

cdrom
A cd disk icon.

preferences
A folder with prefs symbol.

querydoc
A database document icon.

stop
A stop sign.

note
A face with ballon words.

caution
A triangle with an exclamation point.

Under normal conditions, **Tk_GetBitmap** returns an identifier for the requested bitmap. If an error occurs in creating the bitmap, such as when *id* refers to a non−existent file, then **None** is returned and an error message is left in *interp−>result*.

Tk_DefineBitmap associates a name with in−memory bitmap data so that the name can be used in later calls to **Tk_GetBitmap**. The *nameId* argument gives a name for the bitmap; it must not previously have been used in a call to **Tk_DefineBitmap**. The arguments *source*, *width*, and *height* describe the bitmap. **Tk_DefineBitmap** normally returns TCL_OK; if an error occurs (e.g. a bitmap named *nameId* has already been defined) then TCL_ERROR is returned and an error message is left in *interp−>result*. Note: **Tk_DefineBitmap** expects the memory pointed to by *source* to be static: **Tk_DefineBitmap** doesn't make a private copy of this memory, but uses the bytes pointed to by *source* later in calls to **Tk_GetBitmap**.

Typically **Tk_DefineBitmap** is used by **#include**−ing a bitmap file directly into a C program and then referencing the variables defined by the file. For example, suppose there exists a file **stip.bitmap**, which was created by the **bitmap** program and contains a stipple pattern. The following code uses **Tk_DefineBitmap** to define a new bitmap named **foo**:

```
Pixmap bitmap;
#include "stip.bitmap"
Tk_DefineBitmap(interp, Tk_GetUid("foo"), stip_bits,
       stip_width, stip_height);
  ...
bitmap = Tk_GetBitmap(interp, tkwin, Tk_GetUid("foo"));
```

This code causes the bitmap file to be read at compile−time and incorporates the bitmap information into the program's executable image. The same bitmap file could be read at run−time using **Tk_GetBitmap**:

```
Pixmap bitmap;
bitmap = Tk_GetBitmap(interp, tkwin, Tk_GetUid("@stip.bitmap"));
```

The second form is a bit more flexible (the file could be modified after the program has been compiled, or a different string could be provided to read a different file), but it is a little slower and requires the bitmap file to exist separately from the program.

Tk_GetBitmap maintains a database of all the bitmaps that are currently in use. Whenever possible, it will return an existing bitmap rather than creating a new one. This approach can substantially reduce server overhead, so **Tk_GetBitmap** should generally be used in preference to Xlib procedures like **XReadBitmapFile**.

The bitmaps returned by **Tk_GetBitmap** are shared, so callers should never modify them. If a bitmap must be modified dynamically, then it should be created by calling Xlib procedures such as **XReadBitmapFile** or **XCreatePixmap** directly.

The procedure **Tk_NameOfBitmap** is roughly the inverse of **Tk_GetBitmap**. Given an X Pixmap argument, it returns the *id* that was passed to **Tk_GetBitmap** when the bitmap was created. *Bitmap* must have been the return value from a previous call to **Tk_GetBitmap**.

Tk_SizeOfBitmap returns the dimensions of its *bitmap* argument in the words pointed to by the *widthPtr* and *heightPtr* arguments. As with **Tk_NameOfBitmap**, *bitmap* must have been created by **Tk_GetBitmap**.

When a bitmap returned by **Tk_GetBitmap** is no longer needed, **Tk_FreeBitmap** should be called to release it. There should be exactly one call to **Tk_FreeBitmap** for each call to **Tk_GetBitmap**. When a bitmap is no longer in use anywhere (i.e. it has been freed as many times as it has been gotten) **Tk_FreeBitmap** will release it to the X server and delete it from the database.

BUGS

In determining whether an existing bitmap can be used to satisfy a new request, **Tk_GetBitmap** considers only the immediate value of its *id* argument. For example, when a file name is passed to **Tk_GetBitmap**, **Tk_GetBitmap** will assume it is safe to re−use an existing bitmap created from the same file name: it will not check to see whether the file itself has changed, or whether the current directory has changed, thereby causing the name to refer to a different file.

KEYWORDS

bitmap, pixmap

NAME

Tk_GetCapStyle, Tk_NameOfCapStyle − translate between strings and cap styles

=for category C Programming

SYNOPSIS

#include <tk.h>

int **Tk_GetCapStyle(***interp, string, capPtr***)**

char * **Tk_NameOfCapStyle(***cap***)**

ARGUMENTS

Tcl_Interp *interp (in)

Interpreter to use for error reporting.

char *string (in)

String containing name of cap style: one of ‘‘butt’’, ‘‘projecting’’, or ‘‘round’’.

int *capPtr (out)

Pointer to location in which to store X cap style corresponding to *string*.

int cap (in)

Cap style: one of **CapButt**, **CapProjecting**, or **CapRound**.

DESCRIPTION

Tk_GetCapStyle places in **capPtr* the X cap style corresponding to *string*. This will be one of the values **CapButt**, **CapProjecting**, or **CapRound**. Cap styles are typically used in X graphics contexts to indicate how the end−points of lines should be capped. See the X documentation for information on what each style implies.

Under normal circumstances the return value is **TCL_OK** and *interp* is unused. If *string* doesn‘t contain a valid cap style or an abbreviation of one of these names, then an error message is stored in *interp−>result*, **TCL_ERROR** is returned, and **capPtr* is unmodified.

Tk_NameOfCapStyle is the logical inverse of **Tk_GetCapStyle**. Given a cap style such as **CapButt** it returns a statically−allocated string corresponding to *cap*. If *cap* isn‘t a legal cap style, then ‘‘unknown cap style’’ is returned.

KEYWORDS

butt, cap style, projecting, round

NAME

Tk_GetColormap, Tk_FreeColormap − allocate and free colormaps

=for category C Programming

SYNOPSIS

#include <tk.h>

Colormap **Tk_GetColormap(***interp, tkwin, string***)**

Tk_FreeColormap(*display, colormap***)**

ARGUMENTS

Tcl_Interp *interp (in)

Interpreter to use for error reporting.

Tk_Window tkwin (in)

Token for window in which colormap will be used.

char *string (in)

Selects a colormap: either **new** or the name of a window with the same screen and visual as *tkwin*.

Display *display (in)

Display for which *colormap* was allocated.

Colormap colormap (in)

Colormap to free; must have been returned by a previous call to **Tk_GetColormap** or **Tk_GetVisual**.

DESCRIPTION

These procedures are used to manage colormaps. **Tk_GetColormap** returns a colormap suitable for use in *tkwin*. If its *string* argument is **new** then a new colormap is created; otherwise *string* must be the name of another window with the same screen and visual as *tkwin*, and the colormap from that window is returned. If *string* doesn't make sense, or if it refers to a window on a different screen from *tkwin* or with a different visual than *tkwin*, then **Tk_GetColormap** returns **None** and leaves an error message in *interp−>result*.

Tk_FreeColormap should be called when a colormap returned by **Tk_GetColormap** is no longer needed. Tk maintains a reference count for each colormap returned by **Tk_GetColormap**, so there should eventually be one call to **Tk_FreeColormap** for each call to **Tk_GetColormap**. When a colormap's reference count becomes zero, Tk releases the X colormap.

Tk_GetVisual and **Tk_GetColormap** work together, in that a new colormap created by **Tk_GetVisual** may later be returned by **Tk_GetColormap**. The reference counting mechanism for colormaps includes both procedures, so callers of **Tk_GetVisual** must also call **Tk_FreeColormap** to release the colormap. If **Tk_GetColormap** is called with a *string* value of **new** then the resulting colormap will never be returned by **Tk_GetVisual**; however, it can be used in other windows by calling **Tk_GetColormap** with the original window's name as *string*.

KEYWORDS

colormap

NAME

Tk_GetColor, Tk_GetColorByValue, Tk_NameOfColor, Tk_FreeColor − maintain database of colors

=for category C Programming

SYNOPSIS

#include <tk.h>

XColor * **Tk_GetColor**(*interp, tkwin, nameId*)

XColor * **Tk_GetColorByValue**(*tkwin, prefPtr*)

char * **Tk_NameOfColor**(*colorPtr*)

GC **Tk_GCForColor**(*colorPtr, drawable*)

Tk_FreeColor(*colorPtr*)

ARGUMENTS

Tcl_Interp *interp (in)

Interpreter to use for error reporting.

Tk_Window tkwin (in)

Token for window in which color will be used.

Tk_Uid nameId (in)

Textual description of desired color.

XColor *prefPtr (in)

Indicates red, green, and blue intensities of desired color.

XColor *colorPtr (in)

Pointer to X color information. Must have been allocated by previous call to **Tk_GetColor** or **Tk_GetColorByValue**, except when passed to **Tk_NameOfColor**.

Drawable drawable (in)

Drawable in which the result graphics context will be used. Must have same screen and depth as the window for which the color was allocated.

DESCRIPTION

The **Tk_GetColor** and **Tk_GetColorByValue** procedures locate pixel values that may be used to render particular colors in the window given by *tkwin*. In **Tk_GetColor** the desired color is specified with a Tk_Uid (*nameId*), which may have any of the following forms:

colorname

Any of the valid textual names for a color defined in the server's color database file, such as **red** or **PeachPuff**.

#*RGB*
#*RRGGBB*
#*RRRGGGBBB*
#*RRRRGGGGBBBB*

A numeric specification of the red, green, and blue intensities to use to display the color. Each *R*, *G*, or *B* represents a single hexadecimal digit. The four forms permit colors to be specified with 4−bit, 8−bit, 12−bit or 16−bit values. When fewer than 16 bits are provided for each color, they represent the most significant bits of the color. For example, #3a7 is the same as #3000a0007000.

In **Tk_GetColorByValue**, the desired color is indicated with the *red*, *green*, and *blue* fields of the structure pointed to by *colorPtr*.

If **Tk_GetColor** or **Tk_GetColorByValue** is successful in allocating the desired color, then it returns a pointer to an XColor structure; the structure indicates the exact intensities of the allocated color (which may differ slightly from those requested, depending on the limitations of the screen) and a pixel value that may be used to draw in the color. If the colormap for *tkwin* is full, **Tk_GetColor** and **Tk_GetColorByValue** will use the closest existing color in the colormap. If **Tk_GetColor** encounters an error while allocating the color (such as an unknown color name) then NULL is returned and an error message is stored in *interp*−>*result*; **Tk_GetColorByValue** never returns an error.

Tk_GetColor and **Tk_GetColorByValue** maintain a database of all the colors currently in use. If the same *nameId* is requested multiple times from **Tk_GetColor** (e.g. by different windows), or if the same intensities are requested multiple times from **Tk_GetColorByValue**, then existing pixel values will be re−used. Re−using an existing pixel avoids any interaction with the X server, which makes the allocation much more efficient. For this reason, you should generally use **Tk_GetColor** or **Tk_GetColorByValue** instead of Xlib procedures like **XAllocColor**, **XAllocNamedColor**, or **XParseColor**.

Since different calls to **Tk_GetColor** or **Tk_GetColorByValue** may return the same shared pixel value, callers should never change the color of a pixel returned by the procedures. If you need to change a color value dynamically, you should use **XAllocColorCells** to allocate the pixel value for the color.

The procedure **Tk_NameOfColor** is roughly the inverse of **Tk_GetColor**. If its *colorPtr* argument was created by **Tk_GetColor**, then the return value is the *nameId* string that was passed to **Tk_GetColor** to create the color. If *colorPtr* was created by a call to **Tk_GetColorByValue**, or by any other mechanism, then the return value is a string that could be passed to **Tk_GetColor** to return the same color. Note: the string returned by **Tk_NameOfColor** is only guaranteed to persist until the next call to **Tk_NameOfColor**.

Tk_GCForColor returns a graphics context whose **Foreground** field is the pixel allocated for *colorPtr* and whose other fields all have default values. This provides an easy way to do basic drawing with a color. The graphics context is cached with the color and will exist only as long as *colorPtr* exists; it is freed when the last reference to *colorPtr* is freed by calling **Tk_FreeColor**.

When a pixel value returned by **Tk_GetColor** or **Tk_GetColorByValue** is no longer needed, **Tk_FreeColor** should be called to release the color. There should be exactly one call to **Tk_FreeColor** for each call to **Tk_GetColor** or **Tk_GetColorByValue**. When a pixel value is no longer in use anywhere (i.e. it has been freed as many times as it has been gotten) **Tk_FreeColor** will release it to the X server and delete it from the database.

KEYWORDS

color, intensity, pixel value

NAME

Tk_GetCursor, Tk_GetCursorFromData, Tk_NameOfCursor, Tk_FreeCursor − maintain database of cursors

=for category C Programming

SYNOPSIS

#include <tk.h>

Tk_Cursor **Tk_GetCursor(***interp, tkwin, nameId***)**

Tk_Cursor **Tk_GetCursorFromData(***interp, tkwin, source, mask, width, height, xHot, yHot, fg, bg***)**

char * **Tk_NameOfCursor(***display, cursor***)**

Tk_FreeCursor(*display, cursor***)**

ARGUMENTS

Tcl_Interp *interp (in)
: Interpreter to use for error reporting.

Tk_Window tkwin (in)
: Token for window in which the cursor will be used.

Tk_Uid nameId (in)
: Description of cursor; see below for possible values.

char *source (in)
: Data for cursor bitmap, in standard bitmap format.

char *mask (in)
: Data for mask bitmap, in standard bitmap format.

"int" width (in)
: Width of *source* and *mask*.

"int" height (in)
: Height of *source* and *mask*.

"int" xHot (in)
: X−location of cursor hot−spot.

"int" yHot (in)
: Y−location of cursor hot−spot.

Tk_Uid fg (in)
: Textual description of foreground color for cursor.

Tk_Uid bg (in)
: Textual description of background color for cursor.

Display *display (in)
: Display for which *cursor* was allocated.

Tk_Cursor cursor (in)
: Opaque Tk identifier for cursor. If passed to**Tk_FreeCursor**, must have been returned by some previous call to **Tk_GetCursor** or **Tk_GetCursorFromData**.

DESCRIPTION

These procedures manage a collection of cursors being used by an application. The procedures allow cursors to be re−used efficiently, thereby avoiding server overhead, and also allow cursors to be named with

character strings (actually Tk_Uids).

Tk_GetCursor takes as argument a Tk_Uid describing a cursor, and returns an opaque Tk identifier for a cursor corresponding to the description. It re−uses an existing cursor if possible and creates a new one otherwise. *NameId* must be a standard Tcl list with one of the following forms:

name [*fgColor* [*bgColor*]]

Name is the name of a cursor in the standard X cursor font, i.e., any of the names defined in **cursorfont.h**, without the **XC_**. Some example values are **X_cursor**, **hand2**, or **left_ptr**. Appendix B of ''The X Window System'' by Scheifler & Gettys has illustrations showing what each of these cursors looks like. If *fgColor* and *bgColor* are both specified, they give the foreground and background colors to use for the cursor (any of the forms acceptable to **Tk_GetColor** may be used). If only *fgColor* is specified, then there will be no background color: the background will be transparent. If no colors are specified, then the cursor will use black for its foreground color and white for its background color. The Macintosh version of Tk also supports all of the X cursors. Tk on the Mac will also accept any of the standard Mac cursors including **ibeam**, **crosshair**, **watch**, **plus**, and **arrow**. In addition, Tk will load Macintosh cursor resources of the types **crsr** (color) and **CURS** (black and white) by the name of the of the resource. The application and all its open dynamic library's resource files will be searched for the named cursor. If there are conflicts color cursors will always be loaded in preference to black and white cursors.

@sourceName maskName fgColor bgColor

In this form, *sourceName* and *maskName* are the names of files describing bitmaps for the cursor's source bits and mask. Each file must be in standard X11 or X10 bitmap format. *FgColor* and *bgColor* indicate the colors to use for the cursor, in any of the forms acceptable to **Tk_GetColor**. This form of the command will not work on Macintosh or Windows computers.

@sourceName fgColor

This form is similar to the one above, except that the source is used as mask also. This means that the cursor's background is transparent. This form of the command will not work on Macintosh or Windows computers.

Tk_GetCursorFromData allows cursors to be created from in−memory descriptions of their source and mask bitmaps. *Source* points to standard bitmap data for the cursor's source bits, and *mask* points to standard bitmap data describing which pixels of *source* are to be drawn and which are to be considered transparent. *Width* and *height* give the dimensions of the cursor, *xHot* and *yHot* indicate the location of the cursor's hot−spot (the point that is reported when an event occurs), and *fg* and *bg* describe the cursor's foreground and background colors textually (any of the forms suitable for **Tk_GetColor** may be used). Typically, the arguments to **Tk_GetCursorFromData** are created by including a cursor file directly into the source code for a program, as in the following example:

```
Tk_Cursor cursor;
#include "source.cursor"
#include "mask.cursor"
cursor = Tk_GetCursorFromData(interp, tkwin, source_bits,
       mask_bits, source_width, source_height, source_x_hot,
       source_y_hot, Tk_GetUid("red"), Tk_GetUid("blue"));
```

Under normal conditions, **Tk_GetCursor** and **Tk_GetCursorFromData** will return an identifier for the requested cursor. If an error occurs in creating the cursor, such as when *nameId* refers to a non−existent file, then **None** is returned and an error message will be stored in *interp−>result*.

Tk_GetCursor and **Tk_GetCursorFromData** maintain a database of all the cursors they have created. Whenever possible, a call to **Tk_GetCursor** or **Tk_GetCursorFromData** will return an existing cursor rather than creating a new one. This approach can substantially reduce server overhead, so the Tk procedures should generally be used in preference to Xlib procedures like **XCreateFontCursor** or **XCreatePixmapCursor**, which create a new cursor on each call.

The procedure **Tk_NameOfCursor** is roughly the inverse of **Tk_GetCursor**. If its *cursor* argument was created by **Tk_GetCursor**, then the return value is the *nameId* argument that was passed to **Tk_GetCursor** to create the cursor. If *cursor* was created by a call to **Tk_GetCursorFromData**, or by any other mechanism, then the return value is a hexadecimal string giving the X identifier for the cursor. Note: the string returned by **Tk_NameOfCursor** is only guaranteed to persist until the next call to **Tk_NameOfCursor**. Also, this call is not portable except for cursors returned by **Tk_GetCursor**.

When a cursor returned by **Tk_GetCursor** or **Tk_GetCursorFromData** is no longer needed, **Tk_FreeCursor** should be called to release it. There should be exactly one call to **Tk_FreeCursor** for each call to **Tk_GetCursor** or **Tk_GetCursorFromData**. When a cursor is no longer in use anywhere (i.e. it has been freed as many times as it has been gotten) **Tk_FreeCursor** will release it to the X server and remove it from the database.

BUGS

In determining whether an existing cursor can be used to satisfy a new request, **Tk_GetCursor** and **Tk_GetCursorFromData** consider only the immediate values of their arguments. For example, when a file name is passed to **Tk_GetCursor**, **Tk_GetCursor** will assume it is safe to re−use an existing cursor created from the same file name: it will not check to see whether the file itself has changed, or whether the current directory has changed, thereby causing the name to refer to a different file. Similarly, **Tk_GetCursorFromData** assumes that if the same *source* pointer is used in two different calls, then the pointers refer to the same data; it does not check to see if the actual data values have changed.

KEYWORDS

cursor

NAME

Tk_GetFont, Tk_NameOfFont, Tk_FreeFont − maintain database of fonts

=for category C Programming

SYNOPSIS

#include <tk.h>

Tk_Font **Tk_GetFont(***interp, tkwin, string***)**

char * **Tk_NameOfFont(***tkfont***)**

void **Tk_FreeFont(***tkfont***)**

ARGUMENTS

"Tcl_Interp" *interp (in)

Interpreter to use for error reporting.

Tk_Window tkwin (in)

Token for window on the display in which font will be used.

"const char" *string (in)

Name or description of desired font. See documentation for the **font** command for details on acceptable formats.

Tk_Font tkfont (in)

Opaque font token.

DESCRIPTION

Tk_GetFont finds the font indicated by *string* and returns a token that represents the font. The return value can be used in subsequent calls to procedures such as **Tk_FontMetrics**, **Tk_MeasureChars**, and **Tk_FreeFont**. The token returned by **Tk_GetFont** will remain valid until **Tk_FreeFont** is called to release it. *String* can be either a symbolic name or a font description; see the documentation for the **font** command for a description of the valid formats. If **Tk_GetFont** is unsuccessful (because, for example, *string* was not a valid font specification) then it returns **NULL** and stores an error message in *interp−>result*.

Tk_GetFont maintains a database of all fonts it has allocated. If the same *string* is requested multiple times (e.g. by different windows or for different purposes), then additional calls for the same *string* will be handled without involving the platform−specific graphics server.

The procedure **Tk_NameOfFont** is roughly the inverse of **Tk_GetFont**. Given a *tkfont* that was created by **Tk_GetFont**, the return value is the *string* argument that was passed to **Tk_GetFont** to create the font. The string returned by **Tk_NameOfFont** is only guaranteed to persist until the *tkfont* is deleted. The caller must not modify this string.

When a font returned by **Tk_GetFont** is no longer needed, **Tk_FreeFont** should be called to release it. There should be exactly one call to **Tk_FreeFont** for each call to **Tk_GetFont**. When a font is no longer in use anywhere (i.e. it has been freed as many times as it has been gotten) **Tk_FreeFont** will release any platform−specific storage and delete it from the database.

KEYWORDS

font

NAME

Tk_GetFontStruct, Tk_NameOfFontStruct, Tk_FreeFontStruct − maintain database of fonts

=for category C Programming

SYNOPSIS

#include <tk.h>

XFontStruct * **Tk_GetFontStruct(***interp, tkwin, nameId***)**

char * **Tk_NameOfFontStruct(***fontStructPtr***)**

Tk_FreeFontStruct(*fontStructPtr***)**

ARGUMENTS

Tcl_Interp *interp (in)

Interpreter to use for error reporting.

Tk_Window tkwin (in)

Token for window in which font will be used.

Tk_Uid nameId (in)

Name of desired font.

XFontStruct *fontStructPtr (in)

Font structure to return name for or delete.

DESCRIPTION

Tk_GetFont loads the font indicated by *nameId* and returns a pointer to information about the font. The pointer returned by **Tk_GetFont** will remain valid until **Tk_FreeFont** is called to release it. *NameId* can be either a font name or pattern; any value that could be passed to **XLoadQueryFont** may be passed to **Tk_GetFont**. If **Tk_GetFont** is unsuccessful (because, for example, there is no font corresponding to *nameId*) then it returns **NULL** and stores an error message in *interp−>result*.

Tk_GetFont maintains a database of all fonts it has allocated. If the same *nameId* is requested multiple times (e.g. by different windows or for different purposes), then additional calls for the same *nameId* will be handled very quickly, without involving the X server. For this reason, it is generally better to use **Tk_GetFont** in place of X library procedures like **XLoadQueryFont**.

The procedure **Tk_NameOfFontStruct** is roughly the inverse of **Tk_GetFontStruct**. If its *fontStructPtr* argument was created by **Tk_GetFontStruct**, then the return value is the *nameId* argument that was passed to **Tk_GetFontStruct** to create the font. If *fontStructPtr* was not created by a call to **Tk_GetFontStruct**, then the return value is a hexadecimal string giving the X identifier for the associated font. Note: the string returned by **Tk_NameOfFontStruct** is only guaranteed to persist until the next call to **Tk_NameOfFontStruct**.

When a font returned by **Tk_GetFont** is no longer needed, **Tk_FreeFont** should be called to release it. There should be exactly one call to **Tk_FreeFont** for each call to **Tk_GetFont**. When a font is no longer in use anywhere (i.e. it has been freed as many times as it has been gotten) **Tk_FreeFont** will release it to the X server and delete it from the database.

KEYWORDS

font

NAME

Tk_GetGC, Tk_FreeGC − maintain database of read−only graphics contexts

=for category C Programming

SYNOPSIS

#include <tk.h>

GC **Tk_GetGC**(*tkwin, valueMask, valuePtr*)

Tk_FreeGC(*display, gc*)

ARGUMENTS

Tk_Window tkwin (in)

Token for window in which the graphics context will be used.

"unsigned long" valueMask (in)

Mask of bits (such as **GCForeground** or **GCStipple**) indicating which fields of **valuePtr* are valid.

XGCValues *valuePtr (in)

Pointer to structure describing the desired values for the graphics context.

Display *display (in)

Display for which *gc* was allocated.

GC gc (in)

X identifier for graphics context that is no longer needed. Must have been allocated by **Tk_GetGC**.

DESCRIPTION

Tk_GetGC and **Tk_FreeGC** manage a collection of graphics contexts being used by an application. The procedures allow graphics contexts to be shared, thereby avoiding the server overhead that would be incurred if a separate GC were created for each use. **Tk_GetGC** takes arguments describing the desired graphics context and returns an X identifier for a GC that fits the description. The graphics context that is returned will have default values in all of the fields not specified explicitly by *valueMask* and *valuePtr*.

Tk_GetGC maintains a database of all the graphics contexts it has created. Whenever possible, a call to **Tk_GetGC** will return an existing graphics context rather than creating a new one. This approach can substantially reduce server overhead, so **Tk_GetGC** should generally be used in preference to the Xlib procedure **XCreateGC**, which creates a new graphics context on each call.

Since the return values of **Tk_GetGC** are shared, callers should never modify the graphics contexts returned by **Tk_GetGC**. If a graphics context must be modified dynamically, then it should be created by calling **XCreateGC** instead of **Tk_GetGC**.

When a graphics context is no longer needed, **Tk_FreeGC** should be called to release it. There should be exactly one call to **Tk_FreeGC** for each call to **Tk_GetGC**. When a graphics context is no longer in use anywhere (i.e. it has been freed as many times as it has been gotten) **Tk_FreeGC** will release it to the X server and delete it from the database.

KEYWORDS

graphics context

NAME

Tk_GetImage, Tk_RedrawImage, Tk_SizeOfImage, Tk_FreeImage − use an image in a widget

=for category C Programming

SYNOPSIS

#include <tk.h>

Tk_Image **Tk_GetImage**(*interp, tkwin, name, changeProc, clientData*)

Tk_RedrawImage(*image, imageX, imageY, width, height, drawable, drawableX, drawableY*)

Tk_SizeOfImage(*image, widthPtr, heightPtr*)

Tk_FreeImage(*image*)

ARGUMENTS

Tcl_Interp *interp (in)

Place to leave error message.

Tk_Window tkwin (in)

Window in which image will be used.

char *name (in)

Name of image.

Tk_ImageChangedProc *changeProc (in)

Procedure for Tk to invoke whenever image content or size changes.

ClientData clientData (in)

One−word value for Tk to pass to *changeProc*.

Tk_Image image (in)

Token for image instance; must have been returned by a previous call to **Tk_GetImage**.

int imageX (in)

X−coordinate of upper−left corner of region of image to redisplay (measured in pixels from the image's upper−left corner).

int imageY (in)

Y−coordinate of upper−left corner of region of image to redisplay (measured in pixels from the image's upper−left corner).

"int" width ((in))

Width of region of image to redisplay.

"int" height ((in))

Height of region of image to redisplay.

Drawable drawable (in)

Where to display image. Must either be window specified to **Tk_GetImage** or a pixmap compatible with that window.

int drawableX (in)

Where to display image in *drawable*: this is the x−coordinate in *drawable* where x−coordinate *imageX* of the image should be displayed.

int drawableY (in)

Where to display image in *drawable*: this is the y−coordinate in *drawable* where y−coordinate *imageY* of the image should be displayed.

"int" widthPtr (out)

Store width of *image* (in pixels) here.

"int" heightPtr (out)

Store height of *image* (in pixels) here.

DESCRIPTION

These procedures are invoked by widgets that wish to display images. **Tk_GetImage** is invoked by a widget when it first decides to display an image. *name* gives the name of the desired image and *tkwin* identifies the window where the image will be displayed. **Tk_GetImage** looks up the image in the table of existing images and returns a token for a new instance of the image. If the image doesn't exist then **Tk_GetImage** returns NULL and leaves an error message in *interp->result*.

When a widget wishes to actually display an image it must call **Tk_RedrawWidget**, identifying the image (*image*), a region within the image to redisplay (*imageX*, *imageY*, *width*, and *height*), and a place to display the image (*drawable*, *drawableX*, and *drawableY*). Tk will then invoke the appropriate image manager, which will display the requested portion of the image before returning.

A widget can find out the dimensions of an image by calling **Tk_SizeOfImage**: the width and height will be stored in the locations given by *widthPtr* and *heightPtr*, respectively.

When a widget is finished with an image (e.g., the widget is being deleted or it is going to use a different image instead of the current one), it must call **Tk_FreeImage** to release the image instance. The widget should never again use the image token after passing it to **Tk_FreeImage**. There must be exactly one call to **Tk_FreeImage** for each call to **Tk_GetImage**.

If the contents or size of an image changes, then any widgets using the image will need to find out about the changes so that they can redisplay themselves. The *changeProc* and *clientData* arguments to **Tk_GetImage** are used for this purpose. *changeProc* will be called by Tk whenever a change occurs in the image; it must match the following prototype:

```
typedef void Tk_ImageChangedProc(
        ClientData clientData,
        int x,
        int y,
        int width,
        int height,
        int imageWidth,
        int imageHeight);
```

The *clientData* argument to *changeProc* is the same as the *clientData* argument to **Tk_GetImage**. It is usually a pointer to the widget record for the widget or some other data structure managed by the widget. The arguments *x*, *y*, *width*, and *height* identify a region within the image that must be redisplayed; they are specified in pixels measured from the upper–left corner of the image. The arguments *imageWidth* and *imageHeight* give the image's (new) size.

SEE ALSO

Tk::CrtImgType

KEYWORDS

images, redisplay

NAME

Tk_GetJoinStyle, Tk_NameOfJoinStyle − translate between strings and join styles

=for category C Programming

SYNOPSIS

#include <tk.h>

int **Tk_GetJoinStyle(***interp, string, joinPtr***)**

char * **Tk_NameOfJoinStyle(***join***)**

ARGUMENTS

Tcl_Interp *interp (in)

Interpreter to use for error reporting.

char *string (in)

String containing name of join style: one of ‘‘bevel’’, ‘‘miter’’, or ‘‘round’’.

int *joinPtr (out)

Pointer to location in which to store X join style corresponding to *string*.

int join (in)

Join style: one of **JoinBevel**, **JoinMiter**, **JoinRound**.

DESCRIPTION

Tk_GetJoinStyle places in **joinPtr* the X join style corresponding to *string*, which will be one of **JoinBevel**, **JoinMiter**, or **JoinRound**. Join styles are typically used in X graphics contexts to indicate how adjacent line segments should be joined together. See the X documentation for information on what each style implies.

Under normal circumstances the return value is **TCL_OK** and *interp* is unused. If *string* doesn‘t contain a valid join style or an abbreviation of one of these names, then an error message is stored in *interp−>result*, **TCL_ERROR** is returned, and **joinPtr* is unmodified.

Tk_NameOfJoinStyle is the logical inverse of **Tk_GetJoinStyle**. Given a join style such as **JoinBevel** it returns a statically−allocated string corresponding to *join*. If *join* isn‘t a legal join style, then ‘‘unknown join style’’ is returned.

KEYWORDS

bevel, join style, miter, round

NAME

Tk_GetJustify, Tk_NameOfJustify − translate between strings and justification styles

=for category C Programming

SYNOPSIS

#include <tk.h>

Tk_Justify **Tk_GetJustify(***interp, string, justifyPtr***)**

char * **Tk_NameOfJustify(***justify***)**

ARGUMENTS

Tcl_Interp *interp (in)

Interpreter to use for error reporting.

char *string (in)

String containing name of justification style (‘‘left’’, ‘‘right’’, or ‘‘center’’).

int *justifyPtr (out)

Pointer to location in which to store justify value corresponding to *string*.

Tk_Justify justify (in)

Justification style (one of the values listed below).

DESCRIPTION

Tk_GetJustify places in **justifyPtr* the justify value corresponding to *string*. This value will be one of the following:

TK_JUSTIFY_LEFT

Means that the text on each line should start at the left edge of the line; as a result, the right edges of lines may be ragged.

TK_JUSTIFY_RIGHT

Means that the text on each line should end at the right edge of the line; as a result, the left edges of lines may be ragged.

TK_JUSTIFY_CENTER

Means that the text on each line should be centered; as a result, both the left and right edges of lines may be ragged.

Under normal circumstances the return value is **TCL_OK** and *interp* is unused. If *string* doesn‘t contain a valid justification style or an abbreviation of one of these names, then an error message is stored in *interp−>result*, **TCL_ERROR** is returned, and **justifyPtr* is unmodified.

Tk_NameOfJustify is the logical inverse of **Tk_GetJustify**. Given a justify value it returns a statically−allocated string corresponding to *justify*. If *justify* isn‘t a legal justify value, then ‘‘unknown justification style’’ is returned.

KEYWORDS

center, fill, justification, string

NAME

Tk_GetOption − retrieve an option from the option database

=for category C Programming

SYNOPSIS

#include <tk.h>

Tk_Uid **Tk_GetOption**(*tkwin, name, class*)

ARGUMENTS

Tk_Window tkwin (in)

Token for window.

char *name (in)

Name of desired option.

char *class (in)

Class of desired option. Null means there is no class for this option; do lookup based on name only.

DESCRIPTION

This procedure is invoked to retrieve an option from the database associated with *tkwin*'s main window. If there is an option for *tkwin* that matches the given *name* or *class*, then it is returned in the form of a Tk_Uid. If multiple options match *name* and *class*, then the highest−priority one is returned. If no option matches, then NULL is returned.

Tk_GetOption caches options related to *tkwin* so that successive calls for the same *tkwin* will execute much more quickly than successive calls for different windows.

KEYWORDS

class, name, option, retrieve

NAME

Tk::pTk::GetPixels, Tk::pTk::GetScreenMM − translate between strings and screen units

=for category C Programming

SYNOPSIS

#include <tk.h>

int **Tk_GetPixels(***interp, tkwin, string, intPtr***)**

int **Tk_GetScreenMM(***interp, tkwin, string, doublePtr***)**

ARGUMENTS

Tcl_Interp *interp (in)

Interpreter to use for error reporting.

Tk_Window tkwin (in)

Window whose screen geometry determines the conversion between absolute units and pixels.

char *string (in)

String that specifies a distance on the screen.

int *intPtr (out)

Pointer to location in which to store converted distance in pixels.

double *doublePtr (out)

Pointer to location in which to store converted distance in millimeters.

DESCRIPTION

These two procedures take as argument a specification of distance on the screen (*string*) and compute the corresponding distance either in integer pixels or floating−point millimeters. In either case, *string* specifies a screen distance as a floating−point number followed by one of the following characters that indicates units:

<none>

The number specifies a distance in pixels.

c The number specifies a distance in centimeters on the screen.

i The number specifies a distance in inches on the screen.

m The number specifies a distance in millimeters on the screen.

p The number specifies a distance in printer‘s points (1/72 inch) on the screen.

Tk_GetPixels converts *string* to the nearest even number of pixels and stores that value at **intPtr*. **Tk_GetScreenMM** converts *string* to millimeters and stores the double−precision floating−point result at **doublePtr*.

Both procedures return **TCL_OK** under normal circumstances. If an error occurs (e.g. *string* contains a number followed by a character that isn‘t one of the ones above) then **TCL_ERROR** is returned and an error message is left in *interp−>result*.

KEYWORDS

centimeters, convert, inches, millimeters, pixels, points, screen units

NAME

Tk_GetPixmap, Tk_FreePixmap − allocate and free pixmaps

=for category C Programming

SYNOPSIS

#include <tk.h>

Pixmap **Tk_GetPixmap**(*display, d, width, height, depth*)

Tk_FreePixmap(*display, pixmap*)

ARGUMENTS

Display *display (in)

X display for the pixmap.

Drawable d (in)

Pixmap or window where the new pixmap will be used for drawing.

"int" width (in)

Width of pixmap.

"int" height (in)

Height of pixmap.

"int" depth (in)

Number of bits per pixel in pixmap.

Pixmap pixmap (in)

Pixmap to destroy.

DESCRIPTION

These procedures are identical to the Xlib procedures **XCreatePixmap** and **XFreePixmap**, except that they have extra code to manage X resource identifiers so that identifiers for deleted pixmaps can be reused in the future. It is important for Tk applications to use these procedures rather than **XCreatePixmap** and **XFreePixmap**; otherwise long−running applications may run out of resource identifiers.

Tk_GetPixmap creates a pixmap suitable for drawing in *d*, with dimensions given by *width*, *height*, and *depth*, and returns its identifier. **Tk_FreePixmap** destroys the pixmap given by *pixmap* and makes its resource identifier available for reuse.

KEYWORDS

pixmap, resource identifier

NAME

Tk_GetRelief, Tk_NameOfRelief − translate between strings and relief values

=for category C Programming

SYNOPSIS

#include <tk.h>

int **Tk_GetRelief**(*interp, name, reliefPtr*)

char * **Tk_NameOfRelief**(*relief*)

ARGUMENTS

Tcl_Interp *interp (in)

Interpreter to use for error reporting.

char *name (in)

String containing relief name (one of ‘‘flat’’, ‘‘groove’’, ‘‘raised’’, ‘‘ridge’’, ‘‘solid’’, or ‘‘sunken’’).

int *reliefPtr (out)

Pointer to location in which to store relief value corresponding to *name*.

int relief (in)

Relief value (one of TK_RELIEF_FLAT, TK_RELIEF_RAISED, TK_RELIEF_SUNKEN, TK_RELIEF_GROOVE, TK_RELIEF_SOLID, or TK_RELIEF_RIDGE).

DESCRIPTION

Tk_GetRelief places in **reliefPtr* the relief value corresponding to *name*. This value will be one of TK_RELIEF_FLAT, TK_RELIEF_RAISED, TK_RELIEF_SUNKEN, TK_RELIEF_GROOVE, TK_RELIEF_SOLID, or TK_RELIEF_RIDGE. Under normal circumstances the return value is TCL_OK and *interp* is unused. If *name* doesn‘t contain one of the valid relief names or an abbreviation of one of them, then an error message is stored in *interp−>result*, TCL_ERROR is returned, and **reliefPtr* is unmodified.

Tk_NameOfRelief is the logical inverse of **Tk_GetRelief**. Given a relief value it returns the corresponding string (‘‘flat’’, ‘‘raised’’, ‘‘sunken’’, ‘‘groove’’, ‘‘solid’’, or ‘‘ridge’’). If *relief* isn‘t a legal relief value, then ‘‘unknown relief’’ is returned.

KEYWORDS

name, relief, string

NAME

Tk_GetRootCoords – Compute root–window coordinates of window

=for category C Programming

SYNOPSIS

#include <tk.h>

Tk_GetRootCoords(*tkwin, xPtr, yPtr*)

ARGUMENTS

Tk_Window tkwin (in)

Token for window.

int *xPtr (out)

Pointer to location in which to store root–window x–coordinate corresponding to left edge of *tkwin*‘s border.

int *yPtr (out)

Pointer to location in which to store root–window y–coordinate corresponding to top edge of *tkwin*‘s border.

DESCRIPTION

This procedure scans through the structural information maintained by Tk to compute the root–window coordinates corresponding to the upper–left corner of *tkwin*‘s border. If *tkwin* has no border, then **Tk_GetRootCoords** returns the root–window coordinates corresponding to location (0,0) in *tkwin*. **Tk_GetRootCoords** is relatively efficient, since it doesn‘t have to communicate with the X server.

KEYWORDS

coordinates, root window

NAME

Tk_GetScrollInfo − parse arguments for scrolling commands

=for category C Programming

SYNOPSIS

#include <tk.h>

int **Tk_GetScrollInfo(***interp, argc, argv, dblPtr, intPtr***)**

ARGUMENTS

Tcl_Interp *interp (in)

Interpreter to use for error reporting.

int argc (in)

Number of strings in *argv* array.

char *argv[] (in)

Argument strings. These represent the entire method, of which the first word is typically the widget name and the second word is typically **xview** or **yview**. This procedure parses arguments starting with *argv*[2].

double *dblPtr (out)

Filled in with fraction from **moveto** option, if any.

int *intPtr (out)

Filled in with line or page count from **scroll** option, if any. The value may be negative.

DESCRIPTION

Tk_GetScrollInfo parses the arguments expected by widget scrolling commands such as **xview** and **yview**. It receives the entire list of words that make up a method and parses the words starting with *argv*[2]. The words starting with *argv*[2] must have one of the following forms:

```
moveto fraction
scroll number units
scroll number pages
```

Any of the **moveto**, **scroll**, **units**, and **pages** keywords may be abbreviated. If *argv* has the **moveto** form, **TK_SCROLL_MOVETO** is returned as result and **dblPtr* is filled in with the *fraction* argument to the command, which must be a proper real value. If *argv* has the **scroll** form, **TK_SCROLL_UNITS** or **TK_SCROLL_PAGES** is returned and **intPtr* is filled in with the *number* value, which must be a proper integer. If an error occurs in parsing the arguments, **TK_SCROLL_ERROR** is returned and an error message is left in *interp−>result*.

KEYWORDS

parse, scrollbar, scrolling command, xview, yview

NAME

Tk_GetSelection − retrieve the contents of a selection

=for category C Programming

SYNOPSIS

#include <tk.h>

int **Tk_GetSelection**(*interp, tkwin, selection, target, proc, clientData*)

ARGUMENTS

Tcl_Interp *interp (in)

Interpreter to use for reporting errors.

Tk_Window tkwin (in)

Window on whose behalf to retrieve the selection (determines display from which to retrieve).

Atom selection (in)

The name of the selection to be retrieved.

Atom target (in)

Form in which to retrieve selection.

Tk_GetSelProc *proc (in)

Procedure to invoke to process pieces of the selection as they are retrieved.

ClientData clientData (in)

Arbitrary one−word value to pass to *proc*.

DESCRIPTION

Tk_GetSelection retrieves the selection specified by the atom *selection* in the format specified by *target*. The selection may actually be retrieved in several pieces; as each piece is retrieved, *proc* is called to process the piece. *Proc* should have arguments and result that match the type **Tk_GetSelProc**:

```
typedef int Tk_GetSelProc(
        ClientData clientData,
        Tcl_Interp *interp,
        char *portion);
```

The *clientData* and *interp* parameters to *proc* will be copies of the corresponding arguments to **Tk_GetSelection**. *Portion* will be a pointer to a string containing part or all of the selection. For large selections, *proc* will be called several times with successive portions of the selection. The X Inter−Client Communication Conventions Manual allows a selection to be returned in formats other than strings, e.g. as an array of atoms or integers. If this happens, Tk converts the selection back into a string before calling *proc*. If a selection is returned as an array of atoms, Tk converts it to a string containing the atom names separated by white space. For any other format besides string, Tk converts a selection to a string containing hexadecimal values separated by white space.

Tk_GetSelection returns to its caller when the selection has been completely retrieved and processed by *proc*, or when a fatal error has occurred (e.g. the selection owner didn‘t respond promptly).
Tk_GetSelection normally returns TCL_OK; if an error occurs, it returns TCL_ERROR and leaves an error message in *interp−>result*. *Proc* should also return either TCL_OK or TCL_ERROR. If *proc* encounters an error in dealing with the selection, it should leave an error message in *interp−>result* and return TCL_ERROR; this will abort the selection retrieval.

KEYWORDS

format, get, selection retrieval

NAME

Tk_GetUid, Tk_Uid – convert from string to unique identifier

=for category C Programming

SYNOPSIS

#include <tk.h>

#typedef char *Tk_Uid

Tk_Uid **Tk_GetUid**(*string*)

ARGUMENTS

char *string (in)

String for which the corresponding unique identifier is desired.

DESCRIPTION

Tk_GetUid returns the unique identifier corresponding to *string*. Unique identifiers are similar to atoms in Lisp, and are used in Tk to speed up comparisons and searches. A unique identifier (type Tk_Uid) is a string pointer and may be used anywhere that a variable of type ‘‘char *’’ could be used. However, there is guaranteed to be exactly one unique identifier for any given string value. If **Tk_GetUid** is called twice, once with string *a* and once with string *b*, and if *a* and *b* have the same string value (strcmp(a, b) == 0), then **Tk_GetUid** will return exactly the same Tk_Uid value for each call (Tk_GetUid(a) == Tk_GetUid(b)). This means that variables of type Tk_Uid may be compared directly (x == y) without having to call **strcmp**. In addition, the return value from **Tk_GetUid** will have the same string value as its argument (strcmp(Tk_GetUid(a), a) == 0).

KEYWORDS

atom, unique identifier

NAME

Tk_GetVisual − translate from string to visual

=for category C Programming

SYNOPSIS

#include <tk.h>

Visual * **Tk_GetVisual**(*interp, tkwin, string, depthPtr, colormapPtr*)

ARGUMENTS

Tcl_Interp *interp (in)

Interpreter to use for error reporting.

Tk_Window tkwin (in)

Token for window in which the visual will be used.

char *string (in)

String that identifies the desired visual. See below for valid formats.

int *depthPtr (out)

Depth of returned visual gets stored here.

Colormap *colormapPtr (out)

If non−NULL then a suitable colormap for visual is found and its identifier is stored here.

DESCRIPTION

Tk_GetVisual takes a string description of a visual and finds a suitable X Visual for use in *tkwin*, if there is one. It returns a pointer to the X Visual structure for the visual and stores the number of bits per pixel for it at **depthPtr*. If *string* is unrecognizable or if no suitable visual could be found, then NULL is returned and **Tk_GetVisual** leaves an error message in *interp−>result*. If *colormap* is non−NULL then **Tk_GetVisual** also locates an appropriate colormap for use with the result visual and stores its X identifier at **colormapPtr*.

The *string* argument specifies the desired visual in one of the following ways:

class depth

The string consists of a class name followed by an integer depth, with any amount of white space (including none) in between. *class* selects what sort of visual is desired and must be one of **directcolor**, **grayscale**, **greyscale**, **pseudocolor**, **staticcolor**, **staticgray**, **staticgrey**, or **truecolor**, or a unique abbreviation. *depth* specifies how many bits per pixel are needed for the visual. If possible, **Tk_GetVisual** will return a visual with this depth; if there is no visual of the desired depth then **Tk_GetVisual** looks first for a visual with greater depth, then one with less depth.

default

Use the default visual for *tkwin*'s screen.

`$widget`

Use the visual for the window given by `$widget`. `$widget` must be the name of a window on the same screen as *tkwin*.

number

Use the visual whose X identifier is *number*.

best ?*depth*?

Choose the ''best possible'' visual, using the following rules, in decreasing order of priority: (a) a visual that has exactly the desired depth is best, followed by a visual with greater depth than requested (but as little extra as possible), followed by a visual with less depth than requested (but as great a depth

as possible); (b) if no *depth* is specified, then the deepest available visual is chosen; (c) **pseudocolor** is better than **truecolor** or **directcolor**, which are better than **staticcolor**, which is better than **staticgray** or **grayscale**; (d) the default visual for the screen is better than any other visual.

CREDITS

The idea for **Tk_GetVisual**, and the first implementation, came from Paul Mackerras.

KEYWORDS

colormap, screen, visual

NAME

Tk_GetVRootGeometry − Get location and size of virtual root for window

=for category C Programming

SYNOPSIS

#include <tk.h>

Tk_GetVRootGeometry(*tkwin, xPtr, yPtr, widthPtr, heightPtr***)**

ARGUMENTS

Tk_Window tkwin (in)

Token for window whose virtual root is to be queried.

int xPtr (out)

Points to word in which to store x−offset of virtual root.

int yPtr (out)

Points to word in which to store y−offset of virtual root.

"int" widthPtr (out)

Points to word in which to store width of virtual root.

"int" heightPtr (out)

Points to word in which to store height of virtual root.

DESCRIPTION

TkGetVRootGeometry returns geometry information about the virtual root window associated with *tkwin*. The ‘‘associated‘‘ virtual root is the one in which *tkwin*‘s nearest top−level ancestor (or *tkwin* itself if it is a top−level window) has been reparented by the window manager. This window is identified by a **__SWM_ROOT** or **__WM_ROOT** property placed on the top−level window by the window manager. If *tkwin* is not associated with a virtual root (e.g. because the window manager doesn‘t use virtual roots) then **xPtr* and **yPtr* will be set to 0 and **widthPtr* and **heightPtr* will be set to the dimensions of the screen containing *tkwin*.

KEYWORDS

geometry, height, location, virtual root, width, window manager

NAME

Tk_HandleEvent − invoke event handlers for window system events

=for category C Programming

SYNOPSIS

#include <tk.h>

Tk_HandleEvent(*eventPtr*)

ARGUMENTS

XEvent *eventPtr (in)

Pointer to X event to dispatch to relevant handler(s).

DESCRIPTION

Tk_HandleEvent is a lower−level procedure that deals with window events. It is called by **Tk_ServiceEvent** (and indirectly by **Tk_DoOneEvent**), and in a few other cases within Tk. It makes callbacks to any window event handlers (created by calls to **Tk_CreateEventHandler**) that match *eventPtr* and then returns. In some cases it may be useful for an application to bypass the Tk event queue and call **Tk_HandleEvent** directly instead of calling **Tk_QueueEvent** followed by **Tk_ServiceEvent**.

This procedure may be invoked recursively. For example, it is possible to invoke **Tk_HandleEvent** recursively from a handler called by **Tk_HandleEvent**. This sort of operation is useful in some modal situations, such as when a notifier has been popped up and an application wishes to wait for the user to click a button in the notifier before doing anything else.

KEYWORDS

callback, event, handler, window

NAME

Tk_IdToWindow – Find Tk's window information for an X window

=for category C Programming

SYNOPSIS

#include <tk.h>

Tk_Window **Tk_IdToWindow**(*display, window*)

ARGUMENTS

Display *display (in)

X display containing the window.

Window window (in)

X id for window.

DESCRIPTION

Given an X window identifier and the X display it corresponds to, this procedure returns the corresponding Tk_Window handle. If there is no Tk_Window corresponding to `$widget` then NULL is returned.

KEYWORDS

X window id

NAME

Tk_ImageChanged – notify widgets that image needs to be redrawn

=for category C Programming

SYNOPSIS

#include <tk.h>

Tk_ImageChanged(*imageMaster, x, y, width, height, imageWidth, imageHeight*)

ARGUMENTS

Tk_ImageMaster imageMaster (in)

Token for image, which was passed to image's *createProc* when the image was created.

int x (in)

X–coordinate of upper–left corner of region that needs redisplay (measured from upper–left corner of image).

int y (in)

Y–coordinate of upper–left corner of region that needs redisplay (measured from upper–left corner of image).

"int" width (in)

Width of region that needs to be redrawn, in pixels.

"int" height (in)

Height of region that needs to be redrawn, in pixels.

"int" imageWidth (in)

Current width of image, in pixels.

"int" imageHeight (in)

Current height of image, in pixels.

DESCRIPTION

An image manager calls **Tk_ImageChanged** for an image whenever anything happens that requires the image to be redrawn. As a result of calling **Tk_ImageChanged**, any widgets using the image are notified so that they can redisplay themselves appropriately. The *imageMaster* argument identifies the image, and *x*, *y*, *width*, and *height* specify a rectangular region within the image that needs to be redrawn. *imageWidth* and *imageHeight* specify the image's (new) size.

An image manager should call **Tk_ImageChanged** during its *createProc* to specify the image's initial size and to force redisplay if there are existing instances for the image. If any of the pixel values in the image should change later on, **Tk_ImageChanged** should be called again with *x*, *y*, *width*, and *height* values that cover all the pixels that changed. If the size of the image should change, then **Tk_ImageChanged** must be called to indicate the new size, even if no pixels need to be redisplayed.

SEE ALSO

Tk::CrtImgType

KEYWORDS

images, redisplay, image size changes

NAME

Tk_InternAtom, Tk_GetAtomName − manage cache of X atoms

=for category C Programming

SYNOPSIS

#include <tk.h>

Atom **Tk_InternAtom(***tkwin, name*)

char * **Tk_GetAtomName(***tkwin, atom*)

ARGUMENTS

Tk_Window tkwin (in)

Token for window. Used to map atom or name relative to a particular display.

char *name (in)

String name for which atom is desired.

Atom atom (in)

Atom for which corresponding string name is desired.

DESCRIPTION

These procedures are similar to the Xlib procedures **XInternAtom** and **XGetAtomName**. **Tk_InternAtom** returns the atom identifier associated with string given by *name*; the atom identifier is only valid for the display associated with *tkwin*. **Tk_GetAtomName** returns the string associated with *atom* on *tkwin*‘s display. The string returned by **Tk_GetAtomName** is in Tk‘s storage: the caller need not free this space when finished with the string, and the caller should not modify the contents of the returned string. If there is no atom *atom* on *tkwin*‘s display, then **Tk_GetAtomName** returns the string ‘‘?bad atom?’’.

Tk caches the information returned by **Tk_InternAtom** and **Tk_GetAtomName** so that future calls for the same information can be serviced from the cache without contacting the server. Thus **Tk_InternAtom** and **Tk_GetAtomName** are generally much faster than their Xlib counterparts, and they should be used in place of the Xlib procedures.

KEYWORDS

atom, cache, display

NAME

Tk_MainLoop – loop for events until all windows are deleted

=for category C Programming

SYNOPSIS

#include <tk.h>

`Tk_MainLoop``()`

DESCRIPTION

Tk_MainLoop is a procedure that loops repeatedly calling **Tcl_DoOneEvent**. It returns only when there are no applications left in this process (i.e. no main windows exist anymore). Most windowing applications will call **Tk_MainLoop** after initialization; the main execution of the application will consist entirely of callbacks invoked via **Tcl_DoOneEvent**.

KEYWORDS

application, event, main loop

NAME

Tk_MaintainGeometry, Tk_UnmaintainGeometry − maintain geometry of one window relative to another

=for category C Programming

SYNOPSIS

#include <tk.h>

Tk_MaintainGeometry(*slave, master, x, y, width, height*)

Tk_UnmaintainGeometry(*slave, master*)

ARGUMENTS

Tk_Window slave (in)

Window whose geometry is to be controlled.

Tk_Window master (in)

Window relative to which *slave*‘s geometry will be controlled.

int x (in)

Desired x−coordinate of *slave* in *master*, measured in pixels from the inside of *master*‘s left border to the outside of *slave*‘s left border.

int y (in)

Desired y−coordinate of *slave* in *master*, measured in pixels from the inside of *master*‘s top border to the outside of *slave*‘s top border.

int width (in)

Desired width for *slave*, in pixels.

int height (in)

Desired height for *slave*, in pixels.

DESCRIPTION

Tk_MaintainGeometry and **Tk_UnmaintainGeometry** make it easier for geometry managers to deal with slaves whose masters are not their parents. Three problems arise if the master for a slave is not its parent:

[1] The x− and y−position of the slave must be translated from the coordinate system of the master to that of the parent before positioning the slave.

[2] If the master window, or any of its ancestors up to the slave‘s parent, is moved, then the slave must be repositioned within its parent in order to maintain the correct position relative to the master.

[3] If the master or one of its ancestors is mapped or unmapped, then the slave must be mapped or unmapped to correspond.

None of these problems is an issue if the parent and master are the same. For example, if the master or one of its ancestors is unmapped, the slave is automatically removed by the screen by X.

Tk_MaintainGeometry deals with these problems for slaves whose masters aren‘t their parents. **Tk_MaintainGeometry** is typically called by a window manager once it has decided where a slave should be positioned relative to its master. **Tk_MaintainGeometry** translates the coordinates to the coordinate system of *slave*‘s parent and then moves and resizes the slave appropriately. Furthermore, it remembers the desired position and creates event handlers to monitor the master and all of its ancestors up to (but not including) the slave‘s parent. If any of these windows is moved, mapped, or unmapped, the slave will be adjusted so that it is mapped only when the master is mapped and its geometry relative to the master remains as specified by *x*, *y*, *width*, and *height*.

When a window manager relinquishes control over a window, or if it decides that it does not want the window to appear on the screen under any conditions, it calls **Tk_UnmaintainGeometry**.

Tk_UnmaintainGeometry unmaps the window and cancels any previous calls to **Tk_MaintainGeometry** for the *master–slave* pair, so that the slave's geometry and mapped state are no longer maintained automatically. **Tk_UnmaintainGeometry** need not be called by a geometry manager if the slave, the master, or any of the master's ancestors is destroyed: Tk will call it automatically.

If **Tk_MaintainGeometry** is called repeatedly for the same *master–slave* pair, the information from the most recent call supersedes any older information. If **Tk_UnmaintainGeometry** is called for a *master–slave* pair that is isn't currently managed, the call has no effect.

KEYWORDS

geometry manager, map, master, parent, position, slave, unmap

NAME

Tk_MainWindow − find the main window for an application

=for category C Programming

SYNOPSIS

#include <tk.h>

Tk_Window **Tk_MainWindow**(*interp*)

ARGUMENTS

Tcl_Interp *interp (in/out)

Interpreter associated with the application.

DESCRIPTION

If *interp* is associated with a Tk application then **Tk_MainWindow** returns the application's main window. If there is no Tk application associated with *interp* then **Tk_MainWindow** returns NULL and leaves an error message in *interp−>result*.

KEYWORDS

application, main window

NAME

Tk_ManageGeometry − arrange to handle geometry requests for a window

=for category C Programming

SYNOPSIS

#include <tk.h>

Tk_ManageGeometry(*tkwin, mgrPtr, clientData*)

ARGUMENTS

Tk_Window tkwin (in)

Token for window to be managed.

Tk_GeomMgr *mgrPtr (in)

Pointer to data structure containing information about the geometry manager, or NULL to indicate that *tkwin*'s geometry shouldn't be managed anymore. The data structure pointed to by *mgrPtr* must be static: Tk keeps a reference to it as long as the window is managed.

ClientData clientData (in)

Arbitrary one−word value to pass to geometry manager callbacks.

DESCRIPTION

Tk_ManageGeometry arranges for a particular geometry manager, described by the *mgrPtr* argument, to control the geometry of a particular slave window, given by *tkwin*. If *tkwin* was previously managed by some other geometry manager, the previous manager loses control in favor of the new one. If *mgrPtr* is NULL, geometry management is cancelled for *tkwin*.

The structure pointed to by *mgrPtr* contains information about the geometry manager:

```
typedef struct {
        char *name;
        Tk_GeomRequestProc *requestProc;
        Tk_GeomLostSlaveProc *lostSlaveProc;
} Tk_GeomMgr;
```

The *name* field is the textual name for the geometry manager, such as **pack** or **place**; this value will be returned by the command **winfo manager**.

requestProc is a procedure in the geometry manager that will be invoked whenever **Tk_GeometryRequest** is called by the slave to change its desired geometry. *requestProc* should have arguments and results that match the type **Tk_GeomRequestProc**:

```
typedef void Tk_GeomRequestProc(
        ClientData clientData,
        Tk_Window tkwin);
```

The parameters to *requestProc* will be identical to the corresponding parameters passed to **Tk_ManageGeometry**. *clientData* usually points to a data structure containing application−specific information about how to manage *tkwin*'s geometry.

The *lostSlaveProc* field of *mgrPtr* points to another procedure in the geometry manager. Tk will invoke *lostSlaveProc* if some other manager calls **Tk_ManageGeometry** to claim *tkwin* away from the current geometry manager. *lostSlaveProc* is not invoked if **Tk_ManageGeometry** is called with a NULL value for *mgrPtr* (presumably the current geometry manager has made this call, so it already knows that the window is no longer managed), nor is it called if *mgrPtr* is the same as the window's current geometry manager. *lostSlaveProc* should have arguments and results that match the following prototype:

```
typedef void Tk_GeomLostSlaveProc(
        ClientData clientData,
```

```
        Tk_Window tkwin);
```

The parameters to *lostSlaveProc* will be identical to the corresponding parameters passed to **Tk_ManageGeometry**.

KEYWORDS

callback, geometry, managed, request, unmanaged

NAME

Tk_MapWindow, Tk_UnmapWindow − map or unmap a window

=for category C Programming

SYNOPSIS

#include <tk.h>

Tk_Window **Tk_MapWindow**(*tkwin*)

Tk_UnmapWindow(*tkwin*)

ARGUMENTS

Tk_Window tkwin (in)

Token for window.

DESCRIPTION

These procedures may be used to map and unmap windows managed by Tk. **Tk_MapWindow** maps the window given by *tkwin*, and also creates an X window corresponding to *tkwin* if it doesn't already exist. See the **Tk_CreateWindow** manual entry for information on deferred window creation. **Tk_UnmapWindow** unmaps *tkwin*'s window from the screen.

If *tkwin* is a child window (i.e. **Tk_CreateChildWindow** was used to create it), then event handlers interested in map and unmap events are invoked immediately. If *tkwin* isn't an internal window, then the event handlers will be invoked later, after X has seen the request and returned an event for it.

These procedures should be used in place of the X procedures **XMapWindow** and **XUnmapWindow**, since they update Tk's local data structure for *tkwin*. Applications using Tk should not invoke **XMapWindow** and **XUnmapWindow** directly.

KEYWORDS

map, unmap, window

NAME

Tk_MeasureChars, Tk_TextWidth, Tk_DrawChars, Tk_UnderlineChars − routines to measure and display simple single−line strings.

=for category C Programming

SYNOPSIS

#include <tk.h>

int **Tk_MeasureChars(***tkfont, string, maxChars, maxPixels, flags, lengthPtr***)**

int **Tk_TextWidth(***tkfont, string, numChars***)**

void **Tk_DrawChars(***display, drawable, gc, tkfont, string, numChars, x, y***)**

void **Tk_UnderlineChars(***display, drawable, gc, tkfont, string, x, y, firstChar, lastChar***)**

ARGUMENTS

Tk_Font tkfont (in)

Token for font in which text is to be drawn or measured. Must have been returned by a previous call to **Tk_GetFont**.

"const char" *string (in)

Text to be measured or displayed. Need not be null terminated. Any non−printing meta−characters in the string (such as tabs, newlines, and other control characters) will be measured or displayed in a platform−dependent manner.

int maxChars (in)

The maximum number of characters to consider when measuring *string*. Must be greater than or equal to 0.

int maxPixels (in)

If *maxPixels* is greater than 0, it specifies the longest permissible line length in pixels. Characters from *string* are processed only until this many pixels have been covered. If *maxPixels* is <= 0, then the line length is unbounded and the *flags* argument is ignored.

int flags (in)

Various flag bits OR−ed together: TK_PARTIAL_OK means include a character as long as any part of it fits in the length given by *maxPixels*; otherwise, a character must fit completely to be considered. TK_WHOLE_WORDS means stop on a word boundary, if possible. If TK_AT_LEAST_ONE is set, it means return at least one character even if no characters could fit in the length given by *maxPixels*. If TK_AT_LEAST_ONE is set and TK_WHOLE_WORDS is also set, it means that if not even one word fits on the line, return the first few letters of the word that did fit; if not even one letter of the word fit, then the first letter will still be returned.

int *lengthPtr (out)

Filled with the number of pixels occupied by the number of characters returned as the result of **Tk_MeasureChars**.

int numChars (in)

The total number of characters to measure or draw from *string*. Must be greater than or equal to 0.

Display *display (in)

Display on which to draw.

Drawable drawable (in)

Window or pixmap in which to draw.

GC gc (in)

Graphics context for drawing characters. The font selected into this GC must be the same as the *tkfont*.

int "x, y" (in)

Coordinates at which to place the left edge of the baseline when displaying *string*.

int firstChar (in)

The index of the first character to underline in the *string*. Underlining begins at the left edge of this character.

int lastChar (in)

The index of the last character up to which the underline will be drawn. The character specified by *lastChar* will not itself be underlined.

DESCRIPTION

These routines are for measuring and displaying simple single−font, single−line, strings. To measure and display single−font, multi−line, justified text, refer to the documentation for **Tk_ComputeTextLayout**. There is no programming interface in the core of Tk that supports multi−font, multi−line text; support for that behavior must be built on top of simpler layers.

A glyph is the displayable picture of a letter, number, or some other symbol. Not all character codes in a given font have a glyph. Characters such as tabs, newlines/returns, and control characters that have no glyph are measured and displayed by these procedures in a platform−dependent manner; under X, they are replaced with backslashed escape sequences, while under Windows and Macintosh hollow or solid boxes may be substituted. Refer to the documentation for **Tk_ComputeTextLayout** for a programming interface that supports the platform−independent expansion of tab characters into columns and newlines/returns into multi−line text.

Tk_MeasureChars is used both to compute the length of a given string and to compute how many characters from a string fit in a given amount of space. The return value is the number of characters from *string* that fit in the space specified by *maxPixels* subject to the conditions described by *flags*. If all characters fit, the return value will be *maxChars*. **lengthPtr* is filled with the computed width, in pixels, of the portion of the string that was measured. For example, if the return value is 5, then **lengthPtr* is filled with the distance between the left edge of *string*[0] and the right edge of *string*[4].

Tk_TextWidth is a wrapper function that provides a simpler interface to the **Tk_MeasureChars** function. The return value is how much space in pixels the given *string* needs.

Tk_DrawChars draws the *string* at the given location in the given *drawable*.

Tk_UnderlineChars underlines the given range of characters in the given *string*. It doesn't draw the characters (which are assumed to have been displayed previously by **Tk_DrawChars**); it just draws the underline. This procedure is used to underline a few characters without having to construct an underlined font. To produce natively underlined text, the appropriate underlined font should be constructed and used.

KEYWORDS

font

NAME

Tk_MoveToplevelWindow − Adjust the position of a top−level window

=for category C Programming

SYNOPSIS

#include <tk.h>

Tk_MoveToplevelWindow(*tkwin, x, y*)

ARGUMENTS

Tk_Window tkwin (in)

Token for top−level window to move.

int x (in)

New x−coordinate for the top−left pixel of *tkwin*'s border, or the top−left pixel of the decorative border supplied for *tkwin* by the window manager, if there is one.

int y (in)

New y−coordinate for the top−left pixel of *tkwin*'s border, or the top−left pixel of the decorative border supplied for *tkwin* by the window manager, if there is one.

DESCRIPTION

In general, a window should never set its own position; this should be done only by the geometry manger that is responsible for the window. For top−level windows the window manager is effectively the geometry manager; Tk provides interface code between the application and the window manager to convey the application's desires to the geometry manager. The desired size for a top−level window is conveyed using the usual **Tk_GeometryRequest** mechanism. The procedure **Tk_MoveToplevelWindow** may be used by an application to request a particular position for a top−level window; this procedure is similar in function to the **wm geometry** Tcl command except that negative offsets cannot be specified. It is invoked by widgets such as menus that want to appear at a particular place on the screen.

When **Tk_MoveToplevelWindow** is called it doesn't immediately pass on the new desired location to the window manager; it defers this action until all other outstanding work has been completed, using the **Tk_DoWhenIdle** mechanism.

KEYWORDS

position, top−level window, window manager

NAME

Tk_Name, Tk_PathName, Tk_NameToWindow − convert between names and window tokens

=for category C Programming

SYNOPSIS

#include <tk.h>

Tk_Uid **Tk_Name**(*tkwin*)

char * **Tk_PathName**(*tkwin*)

Tk_Window **Tk_NameToWindow**(*interp, pathName, tkwin*)

ARGUMENTS

Tk_Window tkwin (in)

Token for window.

Tcl_Interp *interp (out)

Interpreter to use for error reporting.

char *pathName (in)

Character string containing path name of window.

DESCRIPTION

Each window managed by Tk has two names, a short name that identifies a window among children of the same parent, and a path name that identifies the window uniquely among all the windows belonging to the same main window. The path name is used more often in Tk than the short name; many commands, like **bind**, expect path names as arguments.

The **Tk_Name** macro returns a window's short name, which is the same as the *name* argument passed to **Tk_CreateWindow** when the window was created. The value is returned as a Tk_Uid, which may be used just like a string pointer but also has the properties of a unique identifier (see the the documentation for **Tk_GetUid** for details).

The **Tk_PathName** macro returns a hierarchical name for *tkwin*. Path names have a structure similar to file names in Unix but with dots between elements instead of slashes: the main window for an application has the path name ''.''; its children have names like ''.a'' and ''.b''; their children have names like ''.a.aa'' and ''.b.bb''; and so on. A window is considered to be be a child of another window for naming purposes if the second window was named as the first window's *parent* when the first window was created. This is not always the same as the X window hierarchy. For example, a pop−up is created as a child of the root window, but its logical parent will usually be a window within the application.

The procedure **Tk_NameToWindow** returns the token for a window given its path name (the `$widget` argument) and another window belonging to the same main window (*tkwin*). It normally returns a token for the named window, but if no such window exists **Tk_NameToWindow** leaves an error message in *interp−>result* and returns NULL. The *tkwin* argument to **Tk_NameToWindow** is needed because path names are only unique within a single application hierarchy. If, for example, a single process has opened two main windows, each will have a separate naming hierarchy and the same path name might appear in each of the hierarchies. Normally *tkwin* is the main window of the desired hierarchy, but this need not be the case: any window in the desired hierarchy may be used.

KEYWORDS

name, path name, token, window

NAME

Tk_NameOfImage – Return name of image.

=for category C Programming

SYNOPSIS

#include <tk.h>

char * **Tk_NameOfImage**(*typePtr*)

ARGUMENTS

Tk_ImageMaster *masterPtr (in)

Token for image, which was passed to image manager's *createProc* when the image was created.

DESCRIPTION

This procedure is invoked by image managers to find out the name of an image. Given the token for the image, it returns the string name for the image.

KEYWORDS

image manager, image name

NAME

Tk_OwnSelection − make a window the owner of the primary selection

=for category C Programming

SYNOPSIS

#include <tk.h>

Tk_OwnSelection(*tkwin, selection, proc, clientData*)

ARGUMENTS

Tk_Window tkwin (in)

Window that is to become new selection owner.

Atom selection (in)

The name of the selection to be owned, such as XA_PRIMARY.

Tk_LostSelProc *proc (in)

Procedure to invoke when *tkwin* loses selection ownership later.

ClientData clientData (in)

Arbitrary one−word value to pass to *proc*.

DESCRIPTION

Tk_OwnSelection arranges for *tkwin* to become the new owner of the selection specified by the atom *selection*. After this call completes, future requests for the selection will be directed to handlers created for *tkwin* using **Tk_CreateSelHandler**. When *tkwin* eventually loses the selection ownership, *proc* will be invoked so that the window can clean itself up (e.g. by unhighlighting the selection). *Proc* should have arguments and result that match the type **Tk_LostSelProc**:

```
typedef void Tk_LostSelProc(ClientData clientData);
```

The *clientData* parameter to *proc* is a copy of the *clientData* argument given to **Tk_OwnSelection**, and is usually a pointer to a data structure containing application−specific information about *tkwin*.

KEYWORDS

own, selection owner

NAME

Tk_ParseArgv − process command−line options

=for category C Programming

SYNOPSIS

#include <tk.h>

int **Tk_ParseArgv**(*interp, tkwin, argcPtr, argv, argTable, flags*)

ARGUMENTS

Tcl_Interp *interp (in)

Interpreter to use for returning error messages.

Tk_Window tkwin (in)

Window to use when arguments specify Tk options. If NULL, then no Tk options will be processed.

int argcPtr (in/out)

Pointer to number of arguments in argv; gets modified to hold number of unprocessed arguments that remain after the call.

char **argv (in/out)

Command line arguments passed to main program. Modified to hold unprocessed arguments that remain after the call.

Tk_ArgvInfo *argTable (in)

Array of argument descriptors, terminated by element with type TK_ARGV_END.

int flags (in)

If non−zero, then it specifies one or more flags that control the parsing of arguments. Different flags may be OR'ed together. The flags currently defined are TK_ARGV_DONT_SKIP_FIRST_ARG, TK_ARGV_NO_ABBREV, TK_ARGV_NO_LEFTOVERS, and TK_ARGV_NO_DEFAULTS.

DESCRIPTION

Tk_ParseArgv processes an array of command−line arguments according to a table describing the kinds of arguments that are expected. Each of the arguments in *argv* is processed in turn: if it matches one of the entries in *argTable*, the argument is processed according to that entry and discarded. The arguments that do not match anything in *argTable* are copied down to the beginning of *argv* (retaining their original order) and returned to the caller. At the end of the call **Tk_ParseArgv** sets **argcPtr* to hold the number of arguments that are left in *argv*, and *argv[*argcPtr]* will hold the value NULL. Normally, **Tk_ParseArgv** assumes that *argv[0]* is a command name, so it is treated like an argument that doesn't match *argTable* and returned to the caller; however, if the TK_ARGV_DONT_SKIP_FIRST_ARG bit is set in *flags* then *argv[0]* will be processed just like the other elements of *argv*.

Tk_ParseArgv normally returns the value TCL_OK. If an error occurs while parsing the arguments, then TCL_ERROR is returned and **Tk_ParseArgv** will leave an error message in *interp−>result* in the standard Tcl fashion. In the event of an error return, **argvPtr* will not have been modified, but *argv* could have been partially modified. The possible causes of errors are explained below.

The *argTable* array specifies the kinds of arguments that are expected; each of its entries has the following structure:

```
typedef struct {
        char *key;
        int type;
        char *src;
        char *dst;
        char *help;
```

```
} Tk_ArgvInfo;
```

The *key* field is a string such as ‘‘−display‘‘ or ‘‘−bg‘‘ that is compared with the values in *argv*. *Type* indicates how to process an argument that matches *key* (more on this below). *Src* and *dst* are additional values used in processing the argument. Their exact usage depends on *type*, but typically *src* indicates a value and *dst* indicates where to store the value. The **char *** declarations for *src* and *dst* are placeholders: the actual types may be different. Lastly, *help* is a string giving a brief description of this option; this string is printed when users ask for help about command−line options.

When processing an argument in *argv*, **Tk_ParseArgv** compares the argument to each of the *key*‘s in *argTable*. **Tk_ParseArgv** selects the first specifier whose *key* matches the argument exactly, if such a specifier exists. Otherwise **Tk_ParseArgv** selects a specifier for which the argument is a unique abbreviation. If the argument is a unique abbreviation for more than one specifier, then an error is returned. If there is no matching entry in *argTable*, then the argument is skipped and returned to the caller.

Once a matching argument specifier is found, **Tk_ParseArgv** processes the argument according to the *type* field of the specifier. The argument that matched *key* is called ‘‘the matching argument‘‘ in the descriptions below. As part of the processing, **Tk_ParseArgv** may also use the next argument in *argv* after the matching argument, which is called ‘‘the following argument‘‘. The legal values for *type*, and the processing that they cause, are as follows:

TK_ARGV_END

Marks the end of the table. The last entry in *argTable* must have this type; all of its other fields are ignored and it will never match any arguments.

TK_ARGV_CONSTANT

Src is treated as an integer and *dst* is treated as a pointer to an integer. *Src* is stored at **dst*. The matching argument is discarded.

TK_ARGV_INT

The following argument must contain an integer string in the format accepted by **strtol** (e.g. ‘‘0‘‘ and ‘‘0x‘‘ prefixes may be used to specify octal or hexadecimal numbers, respectively). *Dst* is treated as a pointer to an integer; the following argument is converted to an integer value and stored at **dst*. *Src* is ignored. The matching and following arguments are discarded from *argv*.

TK_ARGV_FLOAT

The following argument must contain a floating−point number in the format accepted by **strtol**. *Dst* is treated as the address of an double−precision floating point value; the following argument is converted to a double−precision value and stored at **dst*. The matching and following arguments are discarded from *argv*.

TK_ARGV_STRING

In this form, *dst* is treated as a pointer to a (char *); **Tk_ParseArgv** stores at **dst* a pointer to the following argument, and discards the matching and following arguments from *argv*. *Src* is ignored.

TK_ARGV_UID

This form is similar to TK_ARGV_STRING, except that the argument is turned into a Tk_Uid by calling **Tk_GetUid**. *Dst* is treated as a pointer to a Tk_Uid; **Tk_ParseArgv** stores at **dst* the Tk_Uid corresponding to the following argument, and discards the matching and following arguments from *argv*. *Src* is ignored.

TK_ARGV_CONST_OPTION

This form causes a Tk option to be set (as if the **option** command had been invoked). The *src* field is treated as a pointer to a string giving the value of an option, and *dst* is treated as a pointer to the name of the option. The matching argument is discarded. If *tkwin* is NULL, then argument specifiers of this type are ignored (as if they did not exist).

TK_ARGV_OPTION_VALUE

This form is similar to TK_ARGV_CONST_OPTION, except that the value of the option is taken from the following argument instead of from *src*. *Dst* is used as the name of the option. *Src* is ignored. The matching and following arguments are discarded. If *tkwin* is NULL, then argument specifiers of this type are ignored (as if they did not exist).

TK_ARGV_OPTION_NAME_VALUE

In this case the following argument is taken as the name of a Tk option and the argument after that is taken as the value for that option. Both *src* and *dst* are ignored. All three arguments are discarded from *argv*. If *tkwin* is NULL, then argument specifiers of this type are ignored (as if they did not exist).

TK_ARGV_HELP

When this kind of option is encountered, **Tk_ParseArgv** uses the *help* fields of *argTable* to format a message describing all the valid arguments. The message is placed in *interp->result* and **Tk_ParseArgv** returns TCL_ERROR. When this happens, the caller normally prints the help message and aborts. If the *key* field of a TK_ARGV_HELP specifier is NULL, then the specifier will never match any arguments; in this case the specifier simply provides extra documentation, which will be included when some other TK_ARGV_HELP entry causes help information to be returned.

TK_ARGV_REST

This option is used by programs or commands that allow the last several of their options to be the name and/or options for some other program. If a **TK_ARGV_REST** argument is found, then **Tk_ParseArgv** doesn't process any of the remaining arguments; it returns them all at the beginning of *argv* (along with any other unprocessed arguments). In addition, **Tk_ParseArgv** treats *dst* as the address of an integer value, and stores at **dst* the index of the first of the **TK_ARGV_REST** options in the returned *argv*. This allows the program to distinguish the **TK_ARGV_REST** options from other unprocessed options that preceded the **TK_ARGV_REST**.

TK_ARGV_FUNC

For this kind of argument, *src* is treated as the address of a procedure, which is invoked to process the following argument. The procedure should have the following structure:

```
int
func(dst, key, nextArg)
        char *dst;
        char *key;
        char *nextArg;
{
}
```

The *dst* and *key* parameters will contain the corresponding fields from the *argTable* entry, and *nextArg* will point to the following argument from *argv* (or NULL if there aren't any more arguments left in *argv*). If *func* uses *nextArg* (so that **Tk_ParseArgv** should discard it), then it should return 1. Otherwise it should return 0 and **TkParseArgv** will process the following argument in the normal fashion. In either event the matching argument is discarded.

TK_ARGV_GENFUNC

This form provides a more general procedural escape. It treats *src* as the address of a procedure, and passes that procedure all of the remaining arguments. The procedure should have the following form:

```
int
genfunc(dst, interp, key, argc, argv)
        char *dst;
        Tcl_Interp *interp;
        char *key;
        int argc;
```

```
        char **argv;
{
}
```

The *dst* and *key* parameters will contain the corresponding fields from the *argTable* entry. *Interp* will be the same as the *interp* argument to **Tcl_ParseArgv**. *Argc* and *argv* refer to all of the options after the matching one. *Genfunc* should behave in a fashion similar to **Tk_ParseArgv**: parse as many of the remaining arguments as it can, then return any that are left by compacting them to the beginning of *argv* (starting at *argv*[0]). *Genfunc* should return a count of how many arguments are left in *argv*; **Tk_ParseArgv** will process them. If *genfunc* encounters an error then it should leave an error message in *interp->result*, in the usual Tcl fashion, and return -1; when this happens **Tk_ParseArgv** will abort its processing and return TCL_ERROR.

FLAGS

TK_ARGV_DONT_SKIP_FIRST_ARG

Tk_ParseArgv normally treats *argv[0]* as a program or command name, and returns it to the caller just as if it hadn't matched *argTable*. If this flag is given, then *argv[0]* is not given special treatment.

TK_ARGV_NO_ABBREV

Normally, **Tk_ParseArgv** accepts unique abbreviations for *key* values in *argTable*. If this flag is given then only exact matches will be acceptable.

TK_ARGV_NO_LEFTOVERS

Normally, **Tk_ParseArgv** returns unrecognized arguments to the caller. If this bit is set in *flags* then **Tk_ParseArgv** will return an error if it encounters any argument that doesn't match *argTable*. The only exception to this rule is *argv[0]*, which will be returned to the caller with no errors as long as TK_ARGV_DONT_SKIP_FIRST_ARG isn't specified.

TK_ARGV_NO_DEFAULTS

Normally, **Tk_ParseArgv** searches an internal table of standard argument specifiers in addition to *argTable*. If this bit is set in *flags*, then **Tk_ParseArgv** will use only *argTable* and not its default table.

EXAMPLE

Here is an example definition of an *argTable* and some sample command lines that use the options. Note the effect on *argc* and *argv*; arguments processed by **Tk_ParseArgv** are eliminated from *argv*, and *argc* is updated to reflect reduced number of arguments.

```
/*
 * Define and set default values for globals.
 */
int debugFlag = 0;
int numReps = 100;
char defaultFileName[] = "out";
char *fileName = defaultFileName;
Boolean exec = FALSE;

/*
 * Define option descriptions.
 */
Tk_ArgvInfo argTable[] = {
       {"-X", TK_ARGV_CONSTANT, (char *) 1, (char *) &debugFlag,
               "Turn on debugging printfs"},
       {"-N", TK_ARGV_INT, (char *) NULL, (char *) &numReps,
               "Number of repetitions"},
       {"-of", TK_ARGV_STRING, (char *) NULL, (char *) &fileName,
               "Name of file for output"},
```

```
        {"x", TK_ARGV_REST, (char *) NULL, (char *) &exec,
                "File to exec, followed by any arguments (must be last argument)."},
        {(char *) NULL, TK_ARGV_END, (char *) NULL, (char *) NULL,
            (char *) NULL}
};

main(argc, argv)
        int argc;
        char *argv[];
{
        ...

        if (Tk_ParseArgv(interp, tkwin, &argc, argv, argTable, 0) != TCL_OK) {
                fprintf(stderr, "%s\n", interp->result);
                exit(1);
        }

        /*
         * Remainder of the program.
         */
}
```

Note that default values can be assigned to variables named in *argTable*: the variables will only be overwritten if the particular arguments are present in *argv*. Here are some example command lines and their effects.

```
prog -N 200 infile              # just sets the numReps variable to 200
prog -of out200 infile          # sets fileName to reference "out200"
prog -XN 10 infile              # sets the debug flag, also sets numReps
```

In all of the above examples, *argc* will be set by **Tk_ParseArgv** to 2, *argv*[0] will be ‘‘prog‘‘, *argv*[1] will be ‘‘infile‘‘, and *argv*[2] will be NULL.

KEYWORDS

arguments, command line, options

NAME

Tk_Preserve, Tk_Release, Tk_EventuallyFree − avoid freeing storage while it‘s being used

=for category C Programming

SYNOPSIS

#include <tk.h>

Tk_Preserve(*clientData*)

Tk_Release(*clientData*)

Tk_EventuallyFree(*clientData, freeProc*)

ARGUMENTS

ClientData clientData (in)

Token describing structure to be freed or reallocated. Usually a pointer to memory for structure.

Tk_FreeProc *freeProc (in)

Procedure to invoke to free *clientData*.

DESCRIPTION

These three procedures help implement a simple reference count mechanism for managing storage. They are designed to solve a problem having to do with widget deletion. When a widget is deleted, its widget record (the structure holding information specific to the widget) must be returned to the storage allocator. However, it‘s possible that the widget record is in active use by one of the procedures on the stack at the time of the deletion. This can happen, for example, if the command associated with a button widget causes the button to be destroyed: an X event causes an event−handling C procedure in the button to be invoked, which in turn causes the button‘s associated Tcl command to be executed, which in turn causes the button to be deleted, which in turn causes the button‘s widget record to be de−allocated. Unfortunately, when the Tcl command returns, the button‘s event−handling procedure will need to reference the button‘s widget record. Because of this, the widget record must not be freed as part of the deletion, but must be retained until the event−handling procedure has finished with it. In other situations where the widget is deleted, it may be possible to free the widget record immediately.

Tk_Preserve and **Tk_Release** implement short−term reference counts for their *clientData* argument. The *clientData* argument identifies an object and usually consists of the address of a structure. The reference counts guarantee that an object will not be freed until each call to **Tk_Preserve** for the object has been matched by calls to **Tk_Release**. There may be any number of unmatched **Tk_Preserve** calls in effect at once.

Tk_EventuallyFree is invoked to free up its *clientData* argument. It checks to see if there are unmatched **Tk_Preserve** calls for the object. If not, then **Tk_EventuallyFree** calls *freeProc* immediately. Otherwise **Tk_EventuallyFree** records the fact that *clientData* needs eventually to be freed. When all calls to **Tk_Preserve** have been matched with calls to **Tk_Release** then *freeProc* will be called by **Tk_Release** to do the cleanup.

All the work of freeing the object is carried out by *freeProc*. *FreeProc* must have arguments and result that match the type **Tk_FreeProc**:

typedef void Tk_FreeProc(ClientData *clientData*);

The *clientData* argument to *freeProc* will be the same as the *clientData* argument to **Tk_EventuallyFree**.

This mechanism can be used to solve the problem described above by placing **Tk_Preserve** and **Tk_Release** calls around actions that may cause undesired storage re−allocation. The mechanism is intended only for short−term use (i.e. while procedures are pending on the stack); it will not work efficiently as a mechanism for long−term reference counts. The implementation does not depend in any way on the internal structure of the objects being freed; it keeps the reference counts in a separate structure.

KEYWORDS

free, reference count, storage

NAME

Tk_QueueWindowEvent − Add a window event to the Tcl event queue

=for category C Programming

SYNOPSIS

#include <tk.h>

Tk_QueueWindowEvent(*eventPtr, position*)

ARGUMENTS

XEvent *eventPtr (in)

An event to add to the event queue.

Tcl_QueuePosition position (in)

Where to add the new event in the queue: **TCL_QUEUE_TAIL**, **TCL_QUEUE_HEAD**, or **TCL_QUEUE_MARK**.

DESCRIPTION

This procedure places a window event on Tcl's internal event queue for eventual servicing. It creates a Tcl_Event structure, copies the event into that structure, and calls **Tcl_QueueEvent** to add the event to the queue. When the event is eventually removed from the queue it is processed just like all window events.

The *position* argument to **Tk_QueueWindowEvent** has the same significance as for **Tcl_QueueEvent**; see the documentation for **Tcl_QueueEvent** for details.

KEYWORDS

callback, clock, handler, modal timeout

NAME

Tk_RestackWindow – Change a window‘s position in the stacking order

=for category C Programming

SYNOPSIS

#include <tk.h>

int **Tk_RestackWindow**(*tkwin, aboveBelow, other*)

ARGUMENTS

Tk_Window tkwin (in)

Token for window to restack.

int aboveBelow (in)

Indicates new position of *tkwin* relative to *other*; must be **Above** or **Below**.

Tk_Window other (in)

Tkwin will be repositioned just above or below this window. Must be a sibling of *tkwin* or a descendant of a sibling. If NULL then *tkwin* is restacked above or below all siblings.

DESCRIPTION

Tk_RestackWindow changes the stacking order of `$widget` relative to its siblings. If *other* is specified as NULL then `$widget` is repositioned at the top or bottom of its stacking order, depending on whether *aboveBelow* is **Above** or **Below**. If *other* has a non–NULL value then `$widget` is repositioned just above or below *other*.

The *aboveBelow* argument must have one of the symbolic values **Above** or **Below**. Both of these values are defined by the include file <X11/Xlib.h>.

KEYWORDS

above, below, obscure, stacking order

NAME

Tk_RestrictEvents – filter and selectively delay X events

=for category C Programming

SYNOPSIS

#include <tk.h>

Tk_RestrictProc * **Tk_RestrictEvents**(*proc, clientData, prevClientDataPtr*)

ARGUMENTS

Tk_RestrictProc *proc (in)

Predicate procedure to call to filter incoming X events. NULL means do not restrict events at all.

ClientData clientData (in)

Arbitrary argument to pass to *proc*.

ClientData *prevClientDataPtr (out)

Pointer to place to save argument to previous restrict procedure.

DESCRIPTION

This procedure is useful in certain situations where applications are only prepared to receive certain X events. After **Tk_RestrictEvents** is called, **Tk_DoOneEvent** (and hence **Tk_MainLoop**) will filter X input events through *proc*. *Proc* indicates whether a given event is to be processed immediately, deferred until some later time (e.g. when the event restriction is lifted), or discarded. *Proc* is a procedure with arguments and result that match the type **Tk_RestrictProc**:

```
typedef Tk_RestrictAction Tk_RestrictProc(
        ClientData clientData,
        XEvent *eventPtr);
```

The *clientData* argument is a copy of the *clientData* passed to **Tk_RestrictEvents**; it may be used to provide *proc* with information it needs to filter events. The *eventPtr* points to an event under consideration. *Proc* returns a restrict action (enumerated type **Tk_RestrictAction**) that indicates what **Tk_DoOneEvent** should do with the event. If the return value is **TK_PROCESS_EVENT**, then the event will be handled immediately. If the return value is **TK_DEFER_EVENT**, then the event will be left on the event queue for later processing. If the return value is **TK_DISCARD_EVENT**, then the event will be removed from the event queue and discarded without being processed.

Tk_RestrictEvents uses its return value and *prevClientDataPtr* to return information about the current event restriction procedure (a NULL return value means there are currently no restrictions). These values may be used to restore the previous restriction state when there is no longer any need for the current restriction.

There are very few places where **Tk_RestrictEvents** is needed. In most cases, the best way to restrict events is by changing the bindings with the **bind** Tcl command or by calling **Tk_CreateEventHandler** and **Tk_DeleteEventHandler** from C. The main place where **Tk_RestrictEvents** must be used is when performing synchronous actions (for example, if you need to wait for a particular event to occur on a particular window but you don‘t want to invoke any handlers for any other events). The ‘‘obvious‘‘ solution in these situations is to call **XNextEvent** or **XWindowEvent**, but these procedures cannot be used because Tk keeps its own event queue that is separate from the X event queue. Instead, call **Tk_RestrictEvents** to set up a filter, then call **Tk_DoOneEvent** to retrieve the desired event(s).

KEYWORDS

delay, event, filter, restriction

NAME

Tk_SetAppName – Set the name of an application for ‘‘send‘‘ commands

=for category C Programming

SYNOPSIS

#include <tk.h>

char * **Tk_SetAppName**(*tkwin, name*)

ARGUMENTS

Tk_Window tkwin (in)

Token for window in application. Used only to select a particular application.

char *name (in)

Name under which to register the application.

DESCRIPTION

Tk_SetAppName associates a name with a given application and records that association on the display containing with the application‘s main window. After this procedure has been invoked, other applications on the display will be able to use the **send** command to invoke operations in the application. If *name* is already in use by some other application on the display, then a new name will be generated by appending ‘‘ **#2**‘‘ to *name*; if this name is also in use, the number will be incremented until an unused name is found. The return value from the procedure is a pointer to the name actually used.

If the application already has a name when **Tk_SetAppName** is called, then the new name replaces the old name.

Tk_SetAppName also adds a **send** command to the application‘s interpreter, which can be used to send commands from this application to others on any of the displays where the application has windows.

The application‘s name registration persists until the interpreter is deleted or the **send** command is deleted from *interp*, at which point the name is automatically unregistered and the application becomes inaccessible via **send**. The application can be made accessible again by calling **Tk_SetAppName**.

Tk_SetAppName is called automatically by **Tk_Init**, so applications don‘t normally need to call it explicitly.

The command **tk appname** provides Tcl–level access to the functionality of **Tk_SetAppName**.

KEYWORDS

application, name, register, send command

NAME

Tk_SetClass, Tk_Class − set or retrieve a window‘s class

=for category C Programming

SYNOPSIS

#include <tk.h>

Tk_SetClass(*tkwin, class*)

Tk_Uid **Tk_Class**(*tkwin*)

ARGUMENTS

Tk_Window tkwin (in)

Token for window.

char *class (in)

New class name for window.

DESCRIPTION

Tk_SetClass is called to associate a class with a particular window. The *class* string identifies the type of the window; all windows with the same general class of behavior (button, menu, etc.) should have the same class. By convention all class names start with a capital letter, and there exists a Tcl command with the same name as each class (except all in lower−case) which can be used to create and manipulate windows of that class. A window‘s class string is initialized to NULL when the window is created.

For main windows, Tk automatically propagates the name and class to the WM_CLASS property used by window managers. This happens either when a main window is actually created (e.g. in **Tk_MakeWindowExist**), or when **Tk_SetClass** is called, whichever occurs later. If a main window has not been assigned a class then Tk will not set the WM_CLASS property for the window.

Tk_Class is a macro that returns the current value of *tkwin*‘s class. The value is returned as a Tk_Uid, which may be used just like a string pointer but also has the properties of a unique identifier (see the the documentation for **Tk_GetUid** for details). If *tkwin* has not yet been given a class, then **Tk_Class** will return NULL.

KEYWORDS

class, unique identifier, window, window manager

NAME

Tk_SetGrid, Tk_UnsetGrid − control the grid for interactive resizing

=for category C Programming

SYNOPSIS

#include <tk.h>

Tk_SetGrid(*tkwin, reqWidth, reqHeight, widthInc, heightInc*)

Tk_UnsetGrid(*tkwin*)

ARGUMENTS

Tk_Window tkwin (in)

Token for window.

int reqWidth (in)

Width in grid units that corresponds to the pixel dimension *tkwin* has requested via **Tk_GeometryRequest**.

int reqHeight (in)

Height in grid units that corresponds to the pixel dimension *tkwin* has requested via **Tk_GeometryRequest**.

int widthInc (in)

Width of one grid unit, in pixels.

int heightInc (in)

Height of one grid unit, in pixels.

DESCRIPTION

Tk_SetGrid turns on gridded geometry management for *tkwin*'s toplevel window and specifies the geometry of the grid. **Tk_SetGrid** is typically invoked by a widget when its **setGrid** option is true. It restricts interactive resizing of *tkwin*'s toplevel window so that the space allocated to the toplevel is equal to its requested size plus or minus even multiples of *widthInc* and *heightInc*. Furthermore, the *reqWidth* and *reqHeight* values are passed to the window manager so that it can report the window's size in grid units during interactive resizes. If *tkwin*'s configuration changes (e.g., the size of a grid unit changes) then the widget should invoke **Tk_SetGrid** again with the new information.

Tk_UnsetGrid cancels gridded geometry management for *tkwin*'s toplevel window.

For each toplevel window there can be at most one internal window with gridding enabled. If **Tk_SetGrid** or **Tk_UnsetGrid** is invoked when some other window is already controlling gridding for *tkwin*'s toplevel, the calls for the new window have no effect.

See the **wm** documentation for additional information on gridded geometry management.

KEYWORDS

grid, window, window manager

NAME

Tk_SetWindowVisual − change visual characteristics of window

=for category C Programming

SYNOPSIS

#include <tk.h>

int **Tk_SetWindowVisual**(*tkwin, visual, depth, colormap*)

ARGUMENTS

Tk_Window tkwin (in)

Token for window.

Visual *visual (in)

New visual type to use for *tkwin*.

"int" depth (in)

Number of bits per pixel desired for *tkwin*.

Colormap colormap (in)

New colormap for *tkwin*, which must be compatible with *visual* and *depth*.

DESCRIPTION

When Tk creates a new window it assigns it the default visual characteristics (visual, depth, and colormap) for its screen. **Tk_SetWindowVisual** may be called to change them. **Tk_SetWindowVisual** must be called before the window has actually been created in X (e.g. before **Tk_MapWindow** or **Tk_MakeWindowExist** has been invoked for the window). The safest thing is to call **Tk_SetWindowVisual** immediately after calling **Tk_CreateWindow**. If *tkwin* has already been created before **Tk_SetWindowVisual** is called then it returns 0 and doesn't make any changes; otherwise it returns 1 to signify that the operation completed successfully.

Note: **Tk_SetWindowVisual** should not be called if you just want to change a window's colormap without changing its visual or depth; call **Tk_SetWindowColormap** instead.

KEYWORDS

colormap, depth, visual

NAME

Tk_Sleep – delay execution for a given number of milliseconds

=for category C Programming

SYNOPSIS

#include <tk.h>

Tk_Sleep(*ms*)

ARGUMENTS

int ms (in)

Number of milliseconds to sleep.

DESCRIPTION

This procedure delays the calling process by the number of milliseconds given by the *ms* parameter, and returns after that time has elapsed. It is typically used for things like flashing a button, where the delay is short and the application needn't do anything while it waits. For longer delays where the application needs to respond to other events during the delay, the procedure **Tk_CreateTimerHandler** should be used instead of **Tk_Sleep**.

KEYWORDS

sleep, time, wait

NAME

Tk_StrictMotif − Return value of tk_strictMotif variable

=for category C Programming

SYNOPSIS

#include <tk.h>

int **Tk_StrictMotif**(*tkwin*)

ARGUMENTS

Tk_Window tkwin (in)

Token for window.

DESCRIPTION

This procedure returns the current value of the **tk_strictMotif** variable in the interpreter associated with *tkwin*'s application. The value is returned as an integer that is either 0 or 1. 1 means that strict Motif compliance has been requested, so anything that is not part of the Motif specification should be avoided. 0 means that ''Motif−like'' is good enough, and extra features are welcome.

This procedure uses a link to the Tcl variable to provide much faster access to the variable's value than could be had by calling **Tcl_GetVar**.

KEYWORDS

Motif compliance, tk_strictMotif variable

NAME

Tk_ComputeTextLayout, Tk_FreeTextLayout, Tk_DrawTextLayout, Tk_UnderlineTextLayout, Tk_PointToChar, Tk_CharBbox, Tk_DistanceToTextLayout, Tk_IntersectTextLayout, Tk_TextLayoutToPostscript − routines to measure and display single−font, multi−line, justified text.

=for category C Programming

SYNOPSIS

#include <tk.h>

Tk_TextLayout **Tk_ComputeTextLayout(***tkfont, string, numChars, wrapLength, justify, flags, widthPtr, heightPtr***)**

void **Tk_FreeTextLayout(***layout***)**

void **Tk_DrawTextLayout(***display, drawable, gc, layout, x, y, firstChar, lastChar***)**

void **Tk_UnderlineTextLayout(***display, drawable, gc, layout, x, y, underline***)**

int **Tk_PointToChar(***layout, x, y***)**

int **Tk_CharBbox(***layout, index, xPtr, yPtr, widthPtr, heightPtr***)**

int **Tk_DistanceToTextLayout(***layout, x, y***)**

int **Tk_IntersectTextLayout(***layout, x, y, width, height***)**

void **Tk_TextLayoutToPostscript(***interp, layout***)**

ARGUMENTS

Tk_Font tkfont (in)

Font to use when constructing and displaying a text layout. The *tkfont* must remain valid for the lifetime of the text layout. Must have been returned by a previous call to **Tk_GetFont**.

"const char" *string (in)

Potentially multi−line string whose dimensions are to be computed and stored in the text layout. The *string* must remain valid for the lifetime of the text layout.

int numChars (in)

The number of characters to consider from *string*. If *numChars* is less than 0, then assumes *string* is null terminated and uses **strlen(***string***)**.

int wrapLength (in)

Longest permissible line length, in pixels. Lines in *string* will automatically be broken at word boundaries and wrapped when they reach this length. If *wrapLength* is too small for even a single character to fit on a line, it will be expanded to allow one character to fit on each line. If *wrapLength* is <= 0, there is no automatic wrapping; lines will get as long as they need to be and only wrap if a newline/return character is encountered.

Tk_Justify justify (in)

How to justify the lines in a multi−line text layout. Possible values are TK_JUSTIFY_LEFT, TK_JUSTIFY_CENTER, or TK_JUSTIFY_RIGHT. If the text layout only occupies a single line, then *justify* is irrelevant.

int flags (in)

Various flag bits OR−ed together. TK_IGNORE_TABS means that tab characters should not be expanded to the next tab stop. TK_IGNORE_NEWLINES means that newline/return characters should not cause a line break. If either tabs or newlines/returns are ignored, then they will be treated as regular characters, being measured and displayed in a platform−dependent manner as described in **Tk_MeasureChars**, and will not have any special behaviors.

int *widthPtr (out)

If non−NULL, filled with either the width, in pixels, of the widest line in the text layout, or the width, in pixels, of the bounding box for the character specified by *index*.

int *heightPtr (out)

If non−NULL, filled with either the total height, in pixels, of all the lines in the text layout, or the height, in pixels, of the bounding box for the character specified by *index*.

Tk_TextLayout layout (in)

A token that represents the cached layout information about the single−font, multi−line, justified piece of text. This token is returned by **Tk_ComputeTextLayout**.

Display *display (in)

Display on which to draw.

Drawable drawable (in)

Window or pixmap in which to draw.

GC gc (in)

Graphics context to use for drawing text layout. The font selected in this GC must correspond to the *tkfont* used when constructing the text layout.

int "x, y" (in)

Point, in pixels, at which to place the upper−left hand corner of the text layout when it is being drawn, or the coordinates of a point (with respect to the upper−left hand corner of the text layout) to check against the text layout.

int firstChar (in)

The index of the first character to draw from the given text layout. The number 0 means to draw from the beginning.

int lastChar (in)

The index of the last character up to which to draw. The character specified by *lastChar* itself will not be drawn. A number less than 0 means to draw all characters in the text layout.

int underline (in)

Index of the single character to underline in the text layout, or a number less than 0 for no underline.

int index (in)

The index of the character whose bounding box is desired. The bounding box is computed with respect to the upper−left hand corner of the text layout.

int "*xPtr, *yPtr" (out)

Filled with the upper−left hand corner, in pixels, of the bounding box for the character specified by *index*. Either or both *xPtr* and *yPtr* may be NULL, in which case the corresponding value is not calculated.

int "width, height" (in)

Specifies the width and height, in pixels, of the rectangular area to compare for intersection against the text layout.

Tcl_Interp *interp (out)

Postscript code that will print the text layout is appended to *interp−>result*.

DESCRIPTION

These routines are for measuring and displaying single−font, multi−line, justified text. To measure and display simple single−font, single−line strings, refer to the documentation for **Tk_MeasureChars**. There is no programming interface in the core of Tk that supports multi−font, multi−line text; support for that

behavior must be built on top of simpler layers.

The routines described here are built on top of the programming interface described in the **Tk_MeasureChars** documentation. Tab characters and newline/return characters may be treated specially by these procedures, but all other characters are passed through to the lower level.

Tk_ComputeTextLayout computes the layout information needed to display a single−font, multi−line, justified *string* of text and returns a Tk_TextLayout token that holds this information. This token is used in subsequent calls to procedures such as **Tk_DrawTextLayout**, **Tk_DistanceToTextLayout**, and **Tk_FreeTextLayout**. The *string* and *tkfont* used when computing the layout must remain valid for the lifetime of this token.

Tk_FreeTextLayout is called to release the storage associated with *layout* when it is no longer needed. A *layout* should not be used in any other text layout procedures once it has been released.

Tk_DrawTextLayout uses the information in *layout* to display a single−font, multi−line, justified string of text at the specified location.

Tk_UnderlineTextLayout uses the information in *layout* to display an underline below an individual character. This procedure does not draw the text, just the underline. To produce natively underlined text, an underlined font should be constructed and used. All characters, including tabs, newline/return characters, and spaces at the ends of lines, can be underlined using this method. However, the underline will never be drawn outside of the computed width of *layout*; the underline will stop at the edge for any character that would extend partially outside of *layout*, and the underline will not be visible at all for any character that would be located completely outside of the layout.

Tk_PointToChar uses the information in *layout* to determine the character closest to the given point. The point is specified with respect to the upper−left hand corner of the *layout*, which is considered to be located at (0, 0). Any point whose *y*−value is less that 0 will be considered closest to the first character in the text layout; any point whose *y*−value is greater than the height of the text layout will be considered closest to the last character in the text layout. Any point whose *x*−value is less than 0 will be considered closest to the first character on that line; any point whose *x*−value is greater than the width of the text layout will be considered closest to the last character on that line. The return value is the index of the character that was closest to the point. Given a *layout* with no characters, the value 0 will always be returned, referring to a hypothetical zero−width placeholder character.

Tk_CharBBox uses the information in *layout* to return the bounding box for the character specified by *index*. The width of the bounding box is the advance width of the character, and does not include any left or right bearing. Any character that extends partially outside of *layout* is considered to be truncated at the edge. Any character that would be located completely outside of *layout* is considered to be zero−width and pegged against the edge. The height of the bounding box is the line height for this font, extending from the top of the ascent to the bottom of the descent; information about the actual height of individual letters is not available. For measurement purposes, a *layout* that contains no characters is considered to contain a single zero−width placeholder character at index 0. If *index* was not a valid character index, the return value is 0 and **xPtr*, **yPtr*, **widthPtr*, and **heightPtr* are unmodified. Otherwise, if *index* did specify a valid, the return value is non−zero, and **xPtr*, **yPtr*, **widthPtr*, and **heightPtr* are filled with the bounding box information for the character. If any of *xPtr*, *yPtr*, *widthPtr*, or *heightPtr* are NULL, the corresponding value is not calculated or stored.

Tk_DistanceToTextLayout computes the shortest distance in pixels from the given point (*x, y*) to the characters in *layout*. Newline/return characters and non−displaying space characters that occur at the end of individual lines in the text layout are ignored for hit detection purposes, but tab characters are not. The return value is 0 if the point actually hits the *layout*. If the point didn't hit the *layout* then the return value is the distance in pixels from the point to the *layout*.

Tk_IntersectTextLayout determines whether a *layout* lies entirely inside, entirely outside, or overlaps a given rectangle. Newline/return characters and non−displaying space characters that occur at the end of individual lines in the *layout* are ignored for intersection calculations. The return value is −1 if the *layout* is entirely outside of the rectangle, 0 if it overlaps, and 1 if it is entirely inside of the rectangle.

Tk_TextLayoutToPostscript outputs code consisting of a Postscript array of strings that represent the individual lines in *layout*. It is the responsibility of the caller to take the Postscript array of strings and add some Postscript function operate on the array to render each of the lines. The code that represents the Postscript array of strings is appended to *interp->result*.

DISPLAY MODEL

When measuring a text layout, space characters that occur at the end of a line are ignored. The space characters still exist and the insertion point can be positioned amongst them, but their additional width is ignored when justifying lines or returning the total width of a text layout. All end-of-line space characters are considered to be attached to the right edge of the line; this behavior is logical for left-justified text and reasonable for center-justified text, but not very useful when editing right-justified text. Spaces are considered variable width characters; the first space that extends past the edge of the text layout is clipped to the edge, and any subsequent spaces on the line are considered zero width and pegged against the edge. Space characters that occur in the middle of a line of text are not suppressed and occupy their normal space width.

Tab characters are not ignored for measurement calculations. If wrapping is turned on and there are enough tabs on a line, the next tab will wrap to the beginning of the next line. There are some possible strange interactions between tabs and justification; tab positions are calculated and the line length computed in a left-justified world, and then the whole resulting line is shifted so it is centered or right-justified, causing the tab columns not to align any more.

When wrapping is turned on, lines may wrap at word breaks (space or tab characters) or newline/returns. A dash or hyphen character in the middle of a word is not considered a word break. **Tk_ComputeTextLayout** always attempts to place at least one word on each line. If it cannot because the *wrapLength* is too small, the word will be broken and as much as fits placed on the line and the rest on subsequent line(s). If *wrapLength* is so small that not even one character can fit on a given line, the *wrapLength* is ignored for that line and one character will be placed on the line anyhow. When wrapping is turned off, only newline/return characters may cause a line break.

When a text layout has been created using an underlined *tkfont*, then any space characters that occur at the end of individual lines, newlines/returns, and tabs will not be displayed underlined when **Tk_DrawTextLayout** is called, because those characters are never actually drawn – they are merely placeholders maintained in the *layout*.

KEYWORDS

font

NAME

Tk_CreateTimerHandler, Tk_DeleteTimerHandler − call a procedure at a given time

=for category C Programming

SYNOPSIS

#include <tk.h>

Tk_TimerToken **Tk_CreateTimerHandler**(*milliseconds, proc, clientData*)

Tk_DeleteTimerHandler(*token*)

ARGUMENTS

int milliseconds (in)

How many milliseconds to wait before invoking *proc*.

Tk_TimerProc *proc (in)

Procedure to invoke after *milliseconds* have elapsed.

ClientData clientData (in)

Arbitrary one−word value to pass to *proc*.

Tk_TimerToken token (in)

Token for previously−created timer handler (the return value from some previous call to **Tk_CreateTimerHandler**).

DESCRIPTION

Tk_CreateTimerHandler arranges for *proc* to be invoked at a time *milliseconds* milliseconds in the future. The callback to *proc* will be made by **Tk_DoOneEvent**, so **Tk_CreateTimerHandler** is only useful in programs that dispatch events through **Tk_DoOneEvent** or through other Tk procedures that call **Tk_DoOneEvent**, such as **Tk_MainLoop**. The call to *proc* may not be made at the exact time given by *milliseconds*: it will be made at the next opportunity after that time. For example, if **Tk_DoOneEvent** isn't called until long after the time has elapsed, or if there are other pending events to process before the call to *proc*, then the call to *proc* will be delayed.

Proc should have arguments and return value that match the type **Tk_TimerProc**:

typedef void Tk_TimerProc(ClientData *clientData*);

The *clientData* parameter to *proc* is a copy of the *clientData* argument given to **Tcl_CreateTimerHandler** when the callback was created. Typically, *clientData* points to a data structure containing application−specific information about what to do in *proc*.

Tk_DeleteTimerHandler may be called to delete a previously−created timer handler. It deletes the handler indicated by *token* so that no call to *proc* will be made; if that handler no longer exists (e.g. because the time period has already elapsed and *proc* has been invoked) then **Tk_DeleteTimerHandler** does nothing.

KEYWORDS

callback, clock, handler, timer

NAME

Tk_Init − add Tk to an interpreter and make a new Tk application.

=for category C Programming

SYNOPSIS

#include <tk.h>

int **Tk_Init**(*interp*)

ARGUMENTS

Tcl_Interp *interp (in)

Interpreter in which to load Tk. Tk should not already be loaded in this interpreter.

DESCRIPTION

Tk_Init is the package initialization procedure for Tk. It is normally invoked by the **Tcl_AppInit** procedure for an application or by the **load** command. **Tk_Init** adds all of Tk's commands to *interp* and creates a new Tk application, including its main window. If the initialization is successful **Tk_Init** returns **TCL_OK**; if there is an error it returns **TCL_ERROR**. **Tk_Init** also leaves a result or error message in *interp*−>*result*.

If there is a variable **argv** in *interp*, **Tk_Init** treats the contents of this variable as a list of options for the new Tk application. The options may have any of the forms documented for the **wish** application (in fact, **wish** uses Tk_Init to process its command−line arguments).

KEYWORDS

application, initialization, load, main window

NAME

Tk_WindowId, Tk_Parent, Tk_Display, Tk_DisplayName, Tk_ScreenNumber, Tk_Screen, Tk_X, Tk_Y, Tk_Width, Tk_Height, Tk_Changes, Tk_Attributes, Tk_IsMapped, Tk_IsTopLevel, Tk_ReqWidth, Tk_ReqHeight, Tk_InternalBorderWidth, Tk_Visual, Tk_Depth, Tk_Colormap − retrieve information from Tk's local data structure

=for category C Programming

SYNOPSIS

#include <tk.h>

Window **Tk_WindowId**(*tkwin*)

Tk_Window **Tk_Parent**(*tkwin*)

Display * **Tk_Display**(*tkwin*)

char * **Tk_DisplayName**(*tkwin*)

int **Tk_ScreenNumber**(*tkwin*)

Screen * **Tk_Screen**(*tkwin*)

int **Tk_X**(*tkwin*)

int **Tk_Y**(*tkwin*)

int **Tk_Width**(*tkwin*)

int **Tk_Height**(*tkwin*)

XWindowChanges * **Tk_Changes**(*tkwin*)

XSetWindowAttributes * **Tk_Attributes**(*tkwin*)

int **Tk_IsMapped**(*tkwin*)

int **Tk_IsTopLevel**(*tkwin*)

int **Tk_ReqWidth**(*tkwin*)

int **Tk_ReqHeight**(*tkwin*)

int **Tk_InternalBorderWidth**(*tkwin*)

Visual * **Tk_Visual**(*tkwin*)

int **Tk_Depth**(*tkwin*)

Colormap **Tk_Colormap**(*tkwin*)

ARGUMENTS

Tk_Window tkwin (in)

Token for window.

DESCRIPTION

Tk_WindowID and the other names listed above are all macros that return fields from Tk's local data structure for *tkwin*. None of these macros requires any interaction with the server; it is safe to assume that all are fast.

Tk_WindowId returns the X identifier for *tkwin*, or **NULL** if no X window has been created for *tkwin* yet.

Tk_Parent returns Tk's token for the logical parent of *tkwin*. The parent is the token that was specified when *tkwin* was created, or NULL for main windows.

Tk_Display returns a pointer to the Xlib display structure corresponding to *tkwin*. **Tk_DisplayName**

returns an ASCII string identifying *tkwin*‘s display. **Tk_ScreenNumber** returns the index of *tkwin*‘s screen among all the screens of *tkwin*‘s display. **Tk_Screen** returns a pointer to the Xlib structure corresponding to *tkwin*‘s screen.

Tk_X, **Tk_Y**, **Tk_Width**, and **Tk_Height** return information about *tkwin‘s* location within its parent and its size. The location information refers to the upper−left pixel in the window, or its border if there is one. The width and height information refers to the interior size of the window, not including any border. **Tk_Changes** returns a pointer to a structure containing all of the above information plus a few other fields. **Tk_Attributes** returns a pointer to an XSetWindowAttributes structure describing all of the attributes of the *tkwin*‘s window, such as background pixmap, event mask, and so on (Tk keeps track of all this information as it is changed by the application). Note: it is essential that applications use Tk procedures like **Tk_ResizeWindow** instead of X procedures like **XResizeWindow**, so that Tk can keep its data structures up−to−date.

Tk_IsMapped returns a non−zero value if *tkwin* is mapped and zero if *tkwin* isn‘t mapped.

Tk_IsTopLevel returns a non−zero value if *tkwin* is a top−level window (its X parent is the root window of the screen) and zero if *tkwin* isn‘t a top−level window.

Tk_ReqWidth and **Tk_ReqHeight** return information about the window‘s requested size. These values correspond to the last call to **Tk_GeometryRequest** for *tkwin*.

Tk_InternalBorderWidth returns the width of internal border that has been requested for *tkwin*, or 0 if no internal border was requested. The return value is simply the last value passed to **Tk_SetInternalBorder** for *tkwin*.

Tk_Visual, **Tk_Depth**, and **Tk_Colormap** return information about the visual characteristics of a window. **Tk_Visual** returns the visual type for the window, **Tk_Depth** returns the number of bits per pixel, and **Tk_Colormap** returns the current colormap for the window. The visual characteristics are normally set from the defaults for the window‘s screen, but they may be overridden by calling **Tk_SetWindowVisual**.

KEYWORDS

attributes, colormap, depth, display, height, geometry manager, identifier, mapped, requested size, screen, top−level, visual, width, window, x, y

NAME

Tk::Dialog – Perl/Tk Dialog widget

=for pm Tk/Dialog.pm

=for category Popups and Dialogs

SYNOPSIS

```
require Tk::Dialog;

$DialogRef = $widget->Dialog(
    -title          => $title,
    -text           => $text,
    -bitmap         => $bitmap,
    -default_button => $default_button,
    -buttons        => [@button_labels],
);

$selected = $DialogRef->Show(?-global?);
```

DESCRIPTION

This is an OO implementation of 'tk_dialog'. First, create all your **Dialog** objects during program initialization. When it's time to use a dialog, invoke the `Show` method on a dialog object; the method then displays the dialog, waits for a button to be invoked, and returns the text label of the selected button.

A Dialog object essentially consists of two subwidgets: a Label widget for the bitmap and a Label wigdet for the text of the dialog. If required, you can invoke the 'configure' method to change any characteristic of these subwidgets.

Because a Dialog object is a Toplevel widget all the 'composite' base class methods are available to you.

Advertised widgets: bitmap, message.

1) Call the constructor to create the dialog object, which in turn returns a blessed reference to the new composite widget:

```
require Tk::Dialog;

$DialogRef = $widget->Dialog(
    -title          => $title,
    -text           => $text,
    -bitmap         => $bitmap,
    -default_button => $default_button,
    -buttons        => [@button_labels],
);
```

- mw a widget reference, usually the result of a `MainWindow->new` call.
- title Title to display in the dialog's decorative frame.
- text Message to display in the dialog widget.
- bitmap

  Bitmap to display in the dialog.
- default_button

  Text label of the button that is to display the default ring (''signifies no default button).
- button_labels

  A reference to a list of one or more strings to display in buttons across the bottom of the dialog.

2) Invoke the `Show` method on a dialog object

```
$button_label = $DialogRef->Show;
```

This returns the text label of the selected button.

(Note: you can request a global grab by passing the string `-global` to the `Show` method.)

SEE ALSO

Tk::DialogBox

KEYWORDS

window, dialog, dialogbox

AUTHOR

Stephen O. Lidie, Lehigh University Computing Center. 94/12/27 lusol@Lehigh.EDU (based on John Stoffel's idea).

NAME

Tk::DropSite – Receive side of Drag & Drop abstraction

=for category User Interaction

SYNOPSIS

```
use Tk::DropSite qw(...);
$widget->DropSite(-entercommand => ...,
                  -dropcommand  => ...,
                  -motioncommand  => ...,
                  -dropcommand  => ...,
                 );
```

DESCRIPTION

`DropSite` creates an object which represents a site on which things may be "Dropped".

A DropSite provides the following methods:

`$site-`Enter($token,$event)
`$site-`Leave($token,$event)
`$site-`Motion($token,$event)
`$site-`Drop($token,$event)

NAME

Tk/compile – making executables from perl/Tk scripts

=for category Other Modules and Languages

SYNOPSIS

```
cd .../Tk800.013/compile
make PROG=...path/to/script
```

DESCRIPTION

This directory is for experimenting with compiling Tk apps. to executables. You need perl5.00556 (or later?) to have the right hooks in perl itself.

This has only been tested on Solaris 2.6 (but I doubt version makes much difference), and RedHat Linux 5.1 both with gcc. As far as I am aware other C compilers should work if 'Makefile' is tweaked appropriately. The Makefile is for GNU make as it uses `$(shell ...)`. The Makefile could be converted to a script (contributions welcome), but having a makefile makes it easier for me to debug the stages.

Instructions

build and install perl5.005_56 (suggest not as your _main_ perl). build and install Tk800.013 with perl5.005_56

cd to this directory Copy the script you want to compile to trial.pl (make PROG=.../script will do that) type 'make'

wait ...

type 'trial' to test the executable.

copy 'trial' to your path.

How it Works

The script is "invoked" with two modules ahead of it on the command line and with perl's –c flag to "just compile". The most important of the modules is Malcolm Beatie's –MO=C "Compile" module. This installs an END {} block so that when perl has finished parsing the script the "C backend" gets control.

The other command line module is –MPreLoad (in this directory) this supplies another END {} block (which is run before Malcolm's), and overrides AutoLoader::import to pass the END {} block a list of modules that "use AutoLoader". The END {} block then loads _all_ the subs that module could load.

When Malcolm's END {} block gets control it then writes C code to build the same data structures that perl has just built. (This takes a long time and writes a large file – ../tiny takes 1:20 minutes and builds 3.5M of C). Lots of messages get written at the start – expect 'bootstrap' to be redefined a few times.

This C code (with version of B::C module in 5.00556) has one function which is very large. This stresses the C compiler, so makefile runs a perl script "splitfunc" which breaks that function down into smaller ones.

Then C code is compiled to an object file. This takes a long time (tiny takes 3:07 minutes). Beware enabling optimization or debug on this compile, it will take even longer.

Another perl script is run to find the shared objects which where loaded by DynaLoader (B::C module has listed them as comments in the C file). We also hunt down the DynaLoader.a library (just in case script will load something at run time).

Then all the above i.e. object files, shared objects, dynaloader are linked with libperl to produce the final executable.

tiny builds a 2.7M executable (stripped). Note though that due to fact that B::C and PreLoad include "everything" from modules, larger apps don't grow in proportion (ptked is 5M stripped).

Restrictions on Scripts

What gets compiled into the script is what gets built by perl up to, but stopping short of running the main script. This means that body of "use‘d" modules is run, but "required" modules are not. The built executable is still (supposed to be) capable of loading other code (including binary modules), but will use "compiled–in" paths to do so.

So ‘use Tk::widgets (...)’ to load the widgets your script needs.

Only AutoLoader is handled by PreLoad. In particular SelfLoader will not work as DATA handle is lost between compile and running executable.

B::C‘s handing of initialized variables is still patchy. Thus if modules set global variables at "compile" time their values may be lost by time executable is run.

Showing Off

This file is being edited by compiled ptked on Linux.

TABLE OF CONTENTS