TKE Developer Guide

Version: 2.9

Author: Trevor Williams

Table of Contents

Plugin Development

This document is written for anyone who is interested in writing plugins for TKE.

TKE contains a plugin framework that allows external Tcl/Tk code to be included and executed in a TKE application. Plugins can be attached to the GUI via various menus, text bindings, the command launcher, and events. This document aims to document the plugin framework, how it works, and, most importantly, how to create new plugins using TKE's plugin API.

Plugin Framework

Starting up

When TKE is started, one of the startup tasks is to read the file contents contained in the TKE plugin directory. This directory contains all of the TKE plugin bundles.

The plugin directory exists in the TKE installation directory under the "plugins" directory. Only bundles in this directory that are properly structured (as described later on in this chapter) are considered for plugin access.

Each plugin bundle is a directory that should contain at least the following files:

File	Required	Description
header.tkedat	Yes	Contains plugin information that describes the plugin and is used by the plugin installer.
main.tcl	Yes	The main Tcl file that is sourced by the plugin installer. This file must contain a call to the api::register procedure. In addition, this file should either contain the plugin namespace and action procedures or source one or more other files in the plugin bundle that contain the action code.
README.md	No	Optional file that should contain usage information about the plugin. This file is displayed as a read-only file in an editing buffer when the user selects an installed plugin with the "Plugin / Show Installed Plugins…" menu option.

After both files are found and the header file is properly parsed, the header contents are stored in a Tcl array. If a plugin bundle does not parse correctly, it is ignored and not made available for usage.

Once this process has completed, the TKE plugin configuration file is read. This file is located at ~/.tke/plugins.tkedat. If this file does not exist, TKE continues without error and its default values take effect. If this file is found, the contents of this file are stored in a Tcl array within TKE. Information stored in this file include which plugins the user has previously selected to use and whether the user has granted the plugin trust (note: trusted plugins are allowed to view and modify the file system and execute system commands) when the plugin was installed.

If a plugin was previously selected by a previous TKE session, the plugin file is included into a separate Tcl interpreter via the Tcl “source” command. If there were any Tcl syntax errors in a given plugin file that are detectable with this source execution, the plugin is marked to be in error. If no syntax errors are found in the file, the plugin registers itself with the plugin framework at this time.

The plugin registration is performed with the api::register procedure. This procedure call associates the plugin with its stored header information. All of the plugin action types associated with the plugin are stored for later retrieval by the plugin framework.

Once all of the selected plugins have been registered, TKE continues on, building the GUI interface and performing other startup actions.

Plugin management

At any time after initial startup time, the user may install/uninstall plugins via the Plugins menu or the command launcher. If a plugin is installed, the associated plugin file is sourced. If there are any errors in a newly sourced plugin, the plugin will remain in the uninstalled state. If the plugin is uninstalled, its associated namespace is deleted from memory and any hooks into the UI are removed.

Once a plugin is installed or uninstalled, the status of all of the plugins is immediately saved to the plugin configuration file (if no plugin configuration file exists in the current directory, it is created).

Interpreter Creation

Two types of plugin interpreters are available. The first interpreter is a “safe” interpreter that restricts a plugin’s ability to view and modify the file system and eliminates the ability to execute subprocesses (i.e., exec and network calls). It essentially provides a clean “sandboxed” environment for the plugin to run in. By default, all plugins are run in this mode of operation. If the plugin requires extended functionality, it can mark its *trust_required* header option to a value of “yes”. If a plugin has this attribute set, when the plugin is installed it will notify the user that the plugin requires extended functionality. The user can then decide whether to grant the plugin trust or reject the request. If trust is granted, the plugin will be installed in an interpreter that will have the full Tcl command library available to do what the plugin requires. If trust is rejected, the plugin will not be installed.

It is preferable that all plugins are written with the intention of running in sandboxed mode, if at all possible.

Safe (Untrusted) Interpreter Description

Safe interpreters all their plugins to view and modify files within three directories:

- ~/.tke/plugins/_plugin_name_
- *installation_directory*/plugins/_plugin_name_
- *installation_directory*/plugins/images

Where *installation_directory* is the pathname to the directory where TKE is installed and *plugin_name* is the name of the plugin. The filenames of these directories are managed in such a way that the “~/.tke/plugins” and “_installation_directory_/plugins” pathnames are hidden encoded is such a way that they cannot be discerned by the

plugin. Specifying any of these directories or files/subdirectories within these directories in any Tcl/Tk command that uses filenames will decode the full pathname within the TKE master interpreter and handle their usage in that interpreter.

The following table lists the differences in standard Tcl commands within a safe plugin interpreter to their standard counterparts.

Command	**Difference Description**
cd	This command is unavailable.
encoding	You may get the system encoding value, but you may not set the system encoding value. All other encoding subcommands are allowed.
exec	This command is unavailable.
exit	This command is unavailable.
fconfigure	This command is unavailable.
file atime, file attributes, file exists, file executable, file isdirectory, file isfile, file mtime, file owned, file readable, file size, file type, file writable	The name argument passed must be a file/directory that exists under one of the sandboxed directories.
file delete	Only names passed to the delete command that exist under one of the sandboxed directories will be deleted.
file dirname	If the resulting directory of this call is a directory under one of the sandboxed directories (or the a sandboxed directory itself), the name of the directory will be returned in encoded form.
file mkdir	Only names that exist under one of the sandboxed directories will be created.
file join, file extension, file rootname, file tail, file separator, file split	These can be called with any pathname since they neither operate on a file system directory/file nor require a valid directory/file for their operation to perform.
file channels, file copy, file link, file lstat, file nativename, file normalize, file pathtype, file readlink, file rename, file stat, file system, file volumes	These commands are not available.
glob	Only names that exist under one of the sandboxed directories (specified with the -directory or -path options) will be checked.
	The requested file, a shared object file, is dynamically loaded into the safe

load	interpreter if it is found. The filename exist in one of the sandboxed directories. Additionally, the shared object file must contain a safe entry point; see the manual page for the load command for more details.
open	You may only open files that exist under one of the three sandboxed directories.
pwd	This command is unavailable.
socket	This command is unavailable.
source	The given filename must exist under one of the sandboxed directories. Additionally, the name of the source file must not be longer than 14 characters, cannot contain more than one period (.) and must end in either .tcl or be named tclIndex.
unload	This command is unavailable.

Plugin GUI Element Creation

Menus

When a menu is about to be displayed to the user (i.e., when the user clicks on a menu entry that creates a menu window to be displayed), the menu is first recursively cleared of all current elements. After everything has been deleted from the menu, the menu is repopulated with the TKE core menu elements (if there are any). Once these elements have been added to the menu, the plugin framework searches for any menu plugin action types in the selected plugin list. When it finds a plugin action type that needs to be added to this menu, the element is added to the menu, creating any cascading menus that are needed to store the menu element (menus can contain a submenu hierarchy so that menu items can be intelligently grouped). Once all of the missing cascading menus have been created (if needed), the menu action command is added to the last menu and it's "-command" option is setup to call the plugin's "do" procedure. The "do" procedure for a plugin action type performs the main action of the plugin action type (which can basically be anything). Once the command has been added to the window, the current plugin's associated "handle_state" procedure is called. The purpose of the "handle_state" procedure is to determine whether the menu item should be enabled (by returning a value of 1) or disabled (by returning a value of 0).

This process is repeated for each menu plugin action type. Once all of the menu items have been added to the menu window, the window is displayed to the user. If the user selects a menu item that corresponds to a plugin action, the "do" procedure for that action is invoked, allowing the plugin to do something meaningful. If the menu item is associated with a table GUI element, the Tk reference to the table is given to the "do" procedure along with the row within the table that the user right-clicked in. If the menu item is associated with a text GUI element, the Tk reference to the text widget is given to the "do" procedure.

Text Bindings

When a new file is opened in the editor, the editor UI is created and TKE core bindings are added to the text widget that contains the file text. After TKE core bindings have been applied, the plugin framework is invoked to find any text binding actions used within plugins. If a plugin has text bindings to add, the plugin framework creates a binding tag that is unique for the plugin and inserts the new tag into the tag binding list for the text widget in one of two places

(depending on what has been specified in the action registration). If the action specifies the “pretext” location, the binding is added prior to any TKE core bindtags. It is important to note that any changes to the text widget will not be visible to the commands that are bound at this point. If the action specifies the “posttext” location, the binding is added immediately after the text bind command is executed. Any commands run at this point will be able to see the changes to the text widget (if any exist).

Take a look at the text_binding example plugin in the plugin installation directory for an example of how text bindings can be used.

Tk Windows

The creation and manipulation of Tk windows (widgets) by a plugin is actually handled within the master. When the plugin interpreter needs to create a Tk widget, the widget is specified just as though the widget was being created in the plugin interpreter. For example, to create a top-level window with a single button in the window, the plugin would perform the following:

```
toplevel .mywin
ttk::button .mywin.b –text “Click Me” –command { foo::click_me }
pack .mywin.b
```

In this example, a single button will be created in a new toplevel window with the text “Click Me”. If the button is clicked, the procedure foo::click_me (which would exist in the plugin interpreter) would be executed. Since all Tk commands are run in the master, a restricted set of the Tk command set is provided. However, all widgets will be themed and configured to match the look and feel of the rest of the Tk window (and they will change to match the UI theme if the user changes the UI theme). The following table lists the available Tk commands and any usage differences between the plugin Tk command set and the standard Tk command set.

Command	Difference Description
canvas, listbox, menu, text, toplevel, ttk::button, ttk::checkbutton, ttk::combobox, ttk::entry, ttk::frame, ttk::label, ttk::labelframe, ttk::menubutton, ttk::notebook, ttk::panedwindow, ttk::progressbar, ttk::radiobutton, ttk::scale, ttk::scrollbar, ttk::separator, ttk::spinbox, ttk::treeview	None. All commands are executed in the plugin interpreter and any variables referenced are variables which exist in the plugin interpreter. Any Tk widgets that are created on behalf of the plugin are destroyable by that plugin. Any other widgets that are passed to the plugin may not be destroyed by the plugin.
clipboard, event, focus, font, grid, pack, place, tk_messageBox	None. All commands are executed in the plugin interpreter and any variables referenced are variables which exist in the plugin interpreter. Any Tk widgets that are created on behalf of the plugin are destroyable by that plugin. Any other widgets that are passed to the plugin may not be destroyed by the plugin.

destroy	Any Tk widgets created by the plugin are destroyed; however, any plugins not created by the plugin are not destroyable. An error will be returned instead.
bind	All commands specified are executed in the plugin interpreter.
winfo atom, winfo atomname, winfo cells, winfo children, winfo class, winfo colormapfull, winfo depth, winfo exists, winfo fpixels, winfo geometry, winfo height, winfo id, winfo ismapped, winfo manager, winfo name, winfo pixels, winfo pointerx, winfo pointerxy, winfo pointery, winfo reqheight, winfo reqwidth, winfo rgb, winfo rootx, winfo rooty, winfo screen, winfo screencells, winfo screendepth, winfo screenheight, winfo screenmmheight, winfo screenmmwidth, winfo screenvisual, winfo screenwidth, winfo viewable, winfo visual, winfo visualsavailable, winfo vrootheight, winfo vrootwidth, winfo vrootx, winfo vrooty, winfo width, winfo x, winfo y	Only Tk widgets created by the plugin may be interrogated via the winfo command.
winfo containing, winfo parent, winfo pathname, winfo toplevel	If the result from executing this command is the name of a window which was created by the plugin, a valid result will be returned; otherwise, an error will be returned.
wm	If the passed window was created by the plugin, the wm command may be executed; otherwise, an error will be returned.
image	If the -file or -maskfile options are specified, the image command will allow these files to be read if the file exists in a directory/subdirectory of one of the trusted directories. If a file is specified with options outside of an trusted directory, an error will be returned. Only images created by the plugin will be deletable by the plugin. If the plugin is uninstalled, the images created by the plugin will be automatically destroyed.
tk::TextSetCursor, tk::TextUpDownLine	

Creating Launcher Commands

TKE has a powerful launcher capability that allows the user to interact with the GUI via keyboard commands. This functionality is also available to plugins via the plugin launcher registration procedure. This procedure is called once for each plugin command that is available. To register a launcher command, call the following procedure from within one of the “do” style procedures.

`api::register_launcher description command`

The *description* argument is a short description of the launcher command. This string is displayed in the launcher results. The *command* argument is the Tcl command to execute when the user selects the launcher entry. The contents of this command can be anything.

Here is a brief example of how to use this command:

```
namespace eval foobar {
  ...
  proc launcher_command {} {
      puts "FOOBAR"
  }
  proc do {} {
      api::register_launcher "Print FOOBAR” foobar::launcher_command
  }
  ...
}
```

The above code will create a launcher that will print the string “FOOBAR” to standard output when invoked in the command launcher.

To unregister a previously registered command launcher command, call the following:

`api::unregister_launcher description`

The value of *description* must match the string passed to the api::register_launcher command to properly unregister the launcher command.

Plugin Bundle Structure

As stated previously, all plugin bundles must reside in the TKE installation’s “plugins” directory, must contain the header.tkedat and main.tcl files (with the required elements). Optionally, the directory can also contain a README.md (Markdown formatted) file which should contain any plugin usage information for the user. These elements are described in detail in this section.

header.tkedat

Every plugin bundle must contain a valid header.tkedat file which is a specially formatted in a tkedat format. The header file can contain comments (ignored by the parser) by specifying the “#” character at the beginning of a line. Any other lines must be specified as follows:

Name

`name {value}`

The value of *value* must be the name of the plugin. This name should match the name of the bundle directory and it must match the name used in the plugin::register procedure call (more about this later). The name of the plugin must be a valid variable name (i.e., no spaces, symbols, etc.).

Author

`author {name}`

The value of *name* should be the name of the user who originally created the plugin.

Email

`email {email_address}`

The value of *email_address* should be the e-mail address of the user who original created the plugin.

Version

`version {version}`

The value of *version* is a numbering system in the format of “major.minor”.

Include

`include {value}`

The value of *value* is either “yes” or “no”. This line specifies whether this plugin should be included in the list of available plugins that user’s can install. Typically this value should be set to the value “yes” which will allow the plugin to be used by users; however, setting this value to “no” allows a plugin which is incomplete or currently not working to be temporarily disabled.

Trust Required

`trust_required {value}`

The value of *value* is either “yes” or “no”. If the value is set to “no” (the default value if this option is not specified in the header file), the plugin will not ask the user to grant it trust and the plugin will be run in “safe” or “untrusted” mode (see the “Safe Interpreter Description” section for details). If the value is set to “yes”, the user will be prompted to grant the plugin trust to operate. If trust is granted, the plugin will be installed and the plugin will be given the full Tcl command set to use. If trust is rejected, the plugin will not be installed.

Description

`description {paragraph}`

The value of paragraph should be a paragraph (multi-lined and formatted) which describes what this plugin does.

The following is an example of what a plugin header might look like:

```
name           {p4_filelog}
author         {John Smith}
email          {jsmith@bubba.com}
version        {1.0}
include        {yes}
trust_required {no}
description    {Adds a function to the sidebar menu popup for
files that, when selected, displays the entire
Perforce filelog history for that file in a
new editor tab.}
```

Registration

Each plugin needs to register itself with the plugin architecture by calling the api::register procedure (from the main.tcl bundle file) which has the following call structure:

```
api::register name action_type_list
```

The value of *name* must match the plugin name in the plugin header. As such, the name must be a valid variable name.

The *action_type_list* is a Tcl list that contains all of the plugin action types used by this plugin. Each plugin action type is a Tcl list that contains information about the plugin action item. Every plugin must contain at least one plugin action type. The contents that make up a plugin action type list depend on the type of plugin action type, though the first element of the list is always a string which names the action type. Appendix A describes each of the plugin action types.

As an example of what a call to the api::register procedure looks like, consider the following example. This example shows what a fairly complex plugin can do.

```
api::register word_count {
  {menu command "Display word count"
      word_count::menu_do
      word_count::menu_handle_state}
}
```

This plugin's purpose is going to display the number of words the exist in the current text widget in the information bar. The menu command will be available in the "Plugins" menu. The menu element is a command type where the word_count::menu_do will be run when the command is selected. The word_count::menu_handle_state call be executed to set the menu state to disabled if no text widget currently is displayed or it will be enabled if there is a current text widget displayed.

Plugin Action Namespace and Procedures

The third required group of elements within a plugin file is the plugin namespace and namespace procedures that are called out in the action type list within the plugin. Every plugin must contain a namespace that matches the name of the plugin in the header. Within this namespace are all of the variables and procedures used for the plugin. It is important that no global variables get created within the plugin to avoid naming collisions.

The makeup and usage of the namespace procedures are fully described in Appendix A.

Other Elements

In addition to the required three elements of a plugin file, the user may include any other procedures, variables, packages, etc. that are needed for the plugin within the file. It is important to note that all plugin variables and procedures reside within the plugin namespace.

Creating a New Plugin Template

Creating a new plugin file template is a straightforward process. First, you must open a new terminal and set the TKE development environment variable to a value of 1 as follows:

```
setenv TKE_DEVEL 1
```

After this command has been entered, run TKE from that same shell. When the application is ready, go to the “Plugins / Create…” menu command (the “Create…” command will be missing from the Plugins menu if TKE is started without the TKE_DEVEL environment variable set).

This will display an entry field at the bottom of the window, prompting you to enter the name of the plugin being created. This name must be a legal variable name (i.e., no whitespace, symbols, etc.) Once a name has been provided and the RETURN key pressed, a new plugin bundle will be created (in the TKE installation’s “plugins” directory) which is named the same as the entered name. Within the bundle, TKE will create a partially filled out template for both the header.tkedat and main.tcl, and these files will displayed within two editor tabs so that the developer can start coding the new plugin behavior.

Supposing that we entered a plugin name of “foobar”, the resulting directory (bundle) “foobar” would be created. The following files will exist in the directory:

header.tkedat

```
name           {foobar}
author         {}
email          {}
version        {1.0}
include        {yes}
trust_required {no}
description    {}
```

main.tcl

```
namespace eval foobar {
}

api::register foobar {
}
```

It is advisable for you to also create a file called README.md in the directory as well which should primarily contain plugin usage information. A user can display the contents of this file within TKE in a read-only buffer by using the Plugins / Show Installed… menu option and selecting one of the installed plugins from the resulting window.

You may, optionally, place any other files that are needed by your plugin within the plugin bundle, including, but not limited to, other Tcl source files, packages, data files, and images.

You cannot use any content that requires a compilation on the installation machine.

Syntax Handling

TKE comes equipped with syntax highlighting support for several popular languages. However, adding support for other languages is supported through the application's syntax description files. All syntax files exist in the installation directory under the data/syntax directory and have a special ".snippet" extension. The base name of the file is the name of the language being supported (ex., the language file to support C++ is called "C++.syntax").

File Format

The format of the syntax file is essentially a Tcl list containing key-value pairs. As such, all values need to be surrounded by curly brackets (i.e, {..}). Tcl command calls are not allowed in the file (i.e., no evaluations or substitutions are performed).

The following subsections describe the individual components of this file along with examples.

filepatterns

The filepatterns value is a list of file extension patterns that are used to automatically identify the type of file to associate the syntax rules to. Whenever a file is opened in the editor, the file's extension is compared against all of the syntax extensions. A match causes the associated language syntax highlighting rules to be used. If a syntax cannot be found, the default "<None>" syntax is used (essentially no syntax highlighting is applied to the file). The format of this list should look as follows:

```
filepatterns {extension …}
```

Each extension value must contain a PERIOD (.) followed by a legal filesystem extension (ex., ".cc" ".tcl" ".php", etc.) Zero or more extension values are allowed in the extension list.

vimsyntax

The vimsyntax value is a list of one or more names that match the corresponding *.vim syntax file (this can be found in the /usr/share/vim/vim_version_/syntax directory of your system, minus the .vim extension) that can also be used for syntax highlighting. This value is compared to any "syntax=name" Vim modeline information to determine which syntax highlighting language to use for the given file.

```
vimsyntax {name …}
```

embedded

The embedded value is used to describe one or more language syntaxes that are either embedded in the language syntax between starting/ending syntax (ex., PHP within HTML) or are mixed in with the current language syntax (ex., C within C++). The value is made up of a list of embedded language descriptions where each language description contains either one or three elements as follows:

```
embedded {
  {language ?start_expression end_expression?}+
}
```

The specified language must be an existing language syntax that is provided natively with TKE or is provided via a plugin. The start_expression and end_expression values are regular expressions that describe the syntax for the start and end of the language syntax (if the language is embedded in the parent language as a block). If the embedded language is intermixed in syntax with the parent language (as is the case with C/C++), then no start_expression and end_expression values should be specified for the language description. For examples of embedded language description, see the HTML.syntax and C++.syntax files in the data/syntax installation directory.

matchcharsallowed

Specifies a list of characters that will be automatically, smartly inserted into or deleted from the editing buffer when its counterpart is inserted/deleted.

`matchcharsallowed { ?value …? }`

The following is the list of legal values:

Value	Description
curly	Left curly bracket ({) insertion/deletion will cause the right curly bracket (}) to be added/removed.
square	Left square bracket ([) insertion/deletion will cause the right square bracket (]) to be added/removed.
paren	Left parenthesis (() insertion/deletion will cause the right parenthesis ()) to be added/removed.
angled	Left angled bracket (<) insertion/deletion will cause the right angled bracket (>) to be added/removed.
double	Double quote (“) insertion/deletion will cause another double quote to be added/removed.
single	Single quote (‘) insertion/deletion will cause another single quote to be added/removed.
btick	Backtick (`) insertion/deletion will cause another backtick to be added/removed.

escapes

The escapes value is used to indicate to the syntax highlighter whether or not the escape character (\) should be treated as a C escape character (i.e., the character immediately following the escape character should not be considered its normal value) or as just another character in the syntax. A value of 1 is the default (consider the escape as in C).

`escapes {0|1}`

tabsallowed

The tabsallowed value is used to determine whether any TAB characters entered in the editor should be inserted as a TAB or should have the TAB substituted as space characters. It is recommended that unless the file type requires TAB characters in the syntax (ex., Makefiles) that this value should be set to false (0). This value should be either 0 (false) or 1 (true) and should be specified as follows:

```
tabsallowed {0|1}
```

casesensitive

The casesensitive value specifies if the language is case sensitive (1) or not (0). If the language is not case sensitive, TKE will perform any keyword/expression matching using a case insensitive method. This value should be either 0 or 1 and should be specified as follows:

```
casesensitive {0|1}
```

delimiters

The delimiters value allows a syntax specification to provide a custom regular expression that is used for determining word boundaries. This value is optional as a default delimiter expression will be used if this value is not specified. The default expression is as follows:

```
[^\s\(\{\[\}\]\)\.\t\n\r;:=\”'\|,<>]+
```

The following specifies the syntax for this element:

```
delimiters {regular_expression}
```

indent

The indent value is a list of language syntax elements that should be used to cause a level of indentation to be automatically added when a newline character is inserted after a syntax match occurs. Each element in the list should be surrounded by curly brackets (ex., {…}) with whitespace added between elements. Any curly brackets used within an element should be escaped with the BACKSPACE (\) character (ex., \{).

Any Tcl regular expressions can be specified for an indent element.

The following specifies the syntax for this element:

```
indent {{indentation_expression} *}
```

unindent

The unindent value is a list of language syntax elements that should be used to cause a level of indentation to be automatically removed when a matching syntax is found. Each element in the list should be surrounded by curly brackets (ex., {…}) with whitespace added between elements. Any curly brackets used within an element should be escaped with the BACKSPACE (\) character (ex. \}}).

Any Tcl regular expressions can be specified for an unindent element.

The following specifies the syntax for this element:

```
unindent {{unindentation_expression} *}
```

reindent

The reindent value is a list of language syntax elements that may cause both an unindent followed by an indent such as the case of C/C++ switch..case syntax. Each element of the list consists of a starting regexp element that starts the sequence of indentations. Each element in the list after it are potential reindent syntax where the first occurrence of the reindent element will not be unindented but all occurrences of one of the reindent elements afterwards (but in the same statement block as the the first occurrence) will be unindented.

This feature allows for proper automatic indentation in the syntax like C/C++ switch and C++ classes (code following “public”, “private” and “protected” lines will be indented) as the following example code shows:

```
switch( a ) {
  case 0 :
… // This code will be auto-indented properly
break;
  case 1 :  // ‘case’ will be unindented automatically
… // This line will be indented
break;
}
```

See the C++.syntax file in the <TKE>/data/syntax directory for an example of what this code would look like to handle a switch/class case using reindent.

The following specifies the syntax for this element:

```
reindent {{start_expression expression …} *}
```

icomment

The icomment value is a list of one or two elements that represent the character string to insert a comment when the user selects the “Text / Comment” menu item. If the list contains one character sequence, the sequence is assumed to be used as a line comment (i.e., it is inserted before each line of selected text at the beginning of the line). If the list contains two character sequences, the sequences are treated as the beginning and end of a block comment (i.e., the first character sequence will be inserted before a block of selected text while the second sequence will be inserted after a block of selected text). Note that the “Text / Uncomment” menu item will not use these values for removing comment characters, instead it uses a combination of the “lcomments” and “bcomments” regular expressions for comment parsing.

lcomments

The lcomments value is a list of language syntax elements that indicate a line comment. Whenever a match in the file occurs, the syntax and all other syntax after it until the newline character is found is syntax highlighted as a comment. Each element in the list should be surrounded by curly brackets (ex., {…}) with whitespace characters added between elements. Any curly brackets used within an element should be escaped with the BACKSPACE (\) character (ex., \{{).

Any Tcl regular expressions can be specified for an lcomments element.

The following specifies the syntax for this element:

```
lcomments {{element} *}
```

bcomments

The bcomments value is a list of language syntax element pairs that indicate the starting syntax and ending syntax elements to define a block comment. All text between these syntax elements are highlighted as comments. Each pair in the list should be surrounded by curly brackets (ex., {…}) as well as each element in the pair. All elements must contain whitespace between them and any curly brackets used within an element should be escaped with the BACKSPACE () character (ex., \{{).

Any Tcl regular expressions can be specified for elements in each bcomments pair.

The following specified the syntax for this element:

```
bcomments {{{start_element} {end_element}} *}
```

strings

The strings value is a list of language syntax elements that indicate the start and end of a string. All text found between two occurrences of an element will be highlighted as a string. Each element in the list should be surrounded by curly brackets (ex., {…}) with whitespace characters added between elements.

Any Tcl regular expressions can be specified for an strings element.

The following specifies the syntax for this element:

```
strings {{element} *}
```

keywords

The keywords value is a list of syntax keywords supported by the language. Each keyword must be a literal value (no regular expressions can be specified) and must be parseable as a word. All elements in the list must be separated by whitespace.

The following specifies the syntax for this element:

```
keywords {{keyword} *}
```

symbols

The symbols value is a list of syntax keywords and/or regular expressions that represent special markers in the code. The name of the symbol is the first word following this keyword/expression. The user can find all symbols within the language and jump to them in the source code by specifying the ‘@‘ symbol in the command launcher and typing in the name of the symbol to search for.

For example, to make all Tcl procedures a symbol, a value of “proc” would be specified in the symbol keyword list. The list of symbols would then be the name of all procedures in the source code.

Whitespace must be used to separate all symbol values in the list.

The following specifies the syntax for this element:

```
symbols {
  {HighlightKeywords {symbol_keyword *}} *
  {HighlightClassForRegexp {regular_expression}} *
}
```

The value of *symbol_keyword* must be a literal value. The value of *regular_expression* must be a valid Tcl regular expression. You can have any number of HighlightClass and/or HighlightClassForRegexp lists in the symbols list.

numbers

The numbers value is a list of regular expressions that represent all valid numbers in the syntax. Any text matching one of these regular expressions will be highlighted with the number syntax color. Whitespace must be used to separate all number expressions in the list.

The following specifies the syntax for this element:

```
numbers {
  {HighlightClassForRegexp {regular_expression}} *
}
```

punctuation

The punctuation value is a list of regular expressions that represent all valid punctuation in the syntax. Any text matching one of these regular expressions will be highlighted with the punctuation syntax color. Whitespace must be used to separate all regular expressions in the list.

The following specifies the syntax for this element:

```
punctuation {
  {HighlightClassForRegexp {regular_expression}} *
}
```

precompile

The precompile value is a list of regular expressions that represent all valid precompiler syntax in the language (if the language supports it). Any text matching one of these regular expressions will be highlighted with the precompile syntax color. Whitespace must be used to separate all regular expressions in the list.

The following specifies the syntax for this element:

```
precompile {
  {HighlightClassForRegexp {regular_expression}} *
  {HighlightClassStartWithChar {character}} *
}
```

The *regular_expression* value must be a valid Tcl expression. The character value must be a single keyboard character. The HighlightClassStartWithChar is a special case regular expression that finds a non-whitespace list of characters that starts with the given character and highlights it. From a performance perspective, it is faster to use this call than a regular expression if your situation can take advantage of it.

miscellaneous1, miscellaneous2, miscellaneous3

The miscellaneous values are a list of literal keyword values or regular expressions that either don't fit in with any of the categories above or an additional color is desired. Each miscellaneous group is associated with its own color. Any values and/or regular expressions that match these values will be highlighted with the corresponding color. Whitespace is required between all values in this list.

The following specifies the syntax for this element:

```
miscellaneous {
  {HighlightKeywords {{keyword} *}}
  {HighlightClassForRegexp {regular_expression}} *
  {HighlightClassStartsWithChar {character}} *
}
```

highlight

The highlight section allows text to be syntax highlighted by colorizing the background color instead of the foreground color. The foreground color of this text will be the same as the background color of the editing window.

The following specifies the syntax for this element:

```
highlighter {
  {HighlightClassForRegexp {regular_expression}} *
  {HighlightClassStartWithChar {character}} *
}
```

meta

The meta section allows text to be syntax highlighted with a color that matches the warning width and line numbers. Any text matched with this type has the special ability to be shown or hidden by the user based on the setting of the View menu option. An example of where this is used is in marking up Markdown characters used for formatting purposes. Since the formatted text is viewable with Markdown, the formatting characters can be hidden to help make the document even easier to read.

The following specifies the syntax for this element:

```
meta {
  {HighlightClassForRegexp {regular_expression}} *
  {HighlightClassStartWithChar {character}} *
}
```

advanced

The advanced section allows for more complex language parsing scenarios (beyond what can be handled with a regular expression only) and allows the user to change the font rendering (i.e., bold, italics, underline, overstrike, superscript, subscript, and font size) and handle mouse clicks.

The advanced section is comprised of two parts. The first part is a list of highlight classes that are user-defined. A highlight class is defined using the following syntax:

`HighlightClass class_name syntax_key {render_options}`

where *class_name* is a user-defined name that will be rendered with the color of the *syntax_key* and the options associated with *render_options*. The value of *syntax_key* can be any of the highlight classes for a syntax file (i.e., strings, keywords, symbols, numbers, punctuation, precompile, miscellaneous1, miscellaneous2, miscellaneous3). The list of *render_options* is a space-separated list of any of the following values:

Value(s)	Description
bold	Any text tagged with the associated *class_name* will be emboldened.
italics	Any text tagged with the associated *class_name* will be italicized.
underline	Any text tagged with the associated *class_name* will be underlined.
h1, h2, h3, h4, h5, h6	Any text tagged with the associated *class_name* will have its font rendered the the specified font size where a value of h1 is the largest font while h6 is a font size one point size greater than the normal font size used in the editor.
overstrike	Any text tagged with the associated *class_name* will be overstriken.
superscript	Any text tagged with the associated *class_name* will be written in superscript.
subscript	Any text tagged with the associated *class_name* will be written in subscript.
click	Any text tagged with the associated *class_name* will be clickable. Any left-clicks associated with the text will call a specified Tcl procedure.

The second section in the advanced section is a list of highlight calls, associating values/regular expressions with Tcl procedure calls that will be executed whenever text is found that matches the value/regular expression. The highlight calls are defined using the following syntax:

```
HighlightRegexp             {regular_expression} procedure_name
HighlightClassStartWithChar {character}          procedure_name
```

For user-created syntax files, the location of the Tcl procedures would be within the main.tcl plugin file. The purposes of the Tcl procedure is to take the matching contents of the text widget and return a list containing a list of tags, their starting positions in the text widget, their ending positions in the text widget, and any Tcl procedures to call (if the tag is clickable) along with an optional new starting position in the text widget to begin parsing (default is to start at the character just after the input matching text.

The following is a representation of this Tcl procedure:

```
proc foobar {txt start_pos end_pos ins} {
  return [list [list [list tag new_start new_end cmd]] “”]
}
```

where the value of tag is one of the user-defined class names within the syntax advanced section, the value of *new_start* and *new_end* is a legal index value for a Tcl text widget, and the value of cmd is a Tcl procedure along with any parameters to pass when it is called. The last element of the list is a legal Tcl text widget index value to begin parsing in the main syntax parser. If this value is the empty string, the parser will resume parsing at the *end_pos* character passed to this function.

The parameters of the procedure include *txt* which is the name of the text widget, *start_pos* and *end_pos* which indicate the range of text that matched the HighlightRegexp or HighlightClassStartWithChar, and *ins* indicates if the procedure callback was due to text being inserted (1) or not (0).

The body of the function should perform some sort of advanced parsing of the given text that ultimately produces the return list. Care should be taken in the body of this function to produce as efficient of code as possible as this procedure could be called often by the syntax parser.

For an example of how to write your own advanced parsing code, refer to the *markdown_color* plugin located in the installation directory (*installation_directory*/plugins/_markdown_color_).

Example

The following example code is taken right from the Tcl syntax file (data/syntax/Tcl.syntax). It can give you an idea about how to create your own syntax file. Feel free to also take a look at any of the other language syntax files in the directory as example code. It is important to note that if a syntax highlight class is not needed, it does not not need to specified in the syntax file.

```
filepatterns
{*.tcl *.msg}

vimsyntax
{tcl}
```

```
matchcharsallowed
{curly square paren double}

tabsallowed
{0}

casesensitive
{1}

indent
{\{}

unindent
{\}}

icomment {{#}}

lcomments {{^[ \t]*#} {;#}}

strings {{"}}

keywords
{
  after append apply array auto_execok auto_import auto_load auto_mkindex
  auto_mkindex_old auto_qualify auto_reset bgerror binary break catch cd chan clock
  close concat continue dde dict else elseif encoding eof error eval exec exit expr
  fblocked fconfigure fcopy file fileevent filename flush for foreach format gets glob global
  history http if incr info interp join lappend lassign lindex linsert list llength load lrange
  lrepeat lreplace levers lsearch lset lsort mathfunc mathop memory msgcat namespace
  open package parray pid pkg::create pkg_mkIndex platform platform::shell puts pwd
  read refchan regexp registry regsub rename return scan seek set socket source split
  string subst switch tcl_endOfWord tcl_findLibrary tcl_startOfNextWord
  tcl_startOfPreviousWord tcl_wordBreakAfter tcl_wordBreakBefore tcltest tell time tm
  trace unknown unload unset update uplevel upvar variable vwait while bind bindtags
}

symbols
{
  HighlightClass proc syntax::get_prefixed_symbol
}

numbers
{
  HighlightClassForRegexp {\m([0-9]+|[0-9]+\.[0-9]*|0x[0-9a-fA-F]+} {}
}

punctuation
{
  HighlightClassForRegexp {[][\{\}]} {}
}

miscellaneous1
{
  HighlightClass {
ctext button label text frame toplevel scrollbar checkbutton canvas
listbox menu menubar menubutton radiobutton scale entry message
tk_chooseDirectory tk_getSaveFile tk_getOpenFile tk_chooseColor tk_optionMenu
ttk::button ttk::checkbutton ttk::combobox ttk::entry ttk::frame ttk::label
ttk::labelframe ttk::menubutton ttk::notebook ttk::panedwindow
ttk::progressbar ttk::radiobutton ttk::scale ttk::scrollbar ttk::separator
ttk::sizegrip ttk::treeview
```

```
  } {}
}

miscellaneous2 {
  HighlightClass {
-text -command -yscrollcommand -xscrollcommand -background -foreground -fg
-bg -highlightbackground -y -x -highlightcolor -relief -width -height -wrap
-font -fill -side -outline -style -insertwidth  -textvariable -activebackground
-activeforeground -insertbackground -anchor -orient -troughcolor -nonewline
-expand -type -message -title -offset -in -after -yscroll -xscroll -forward
-regexp -count -exact -padx -ipadx -filetypes -all -from -to -label -value
-variable -regexp -backwards -forwards -bd -pady -ipady -state -row -column
-cursor -highlightcolors -linemap -menu -tearoff -displayof -cursor -underline
-tags -tag -weight -sticky -rowspan -columnspan
  } {}
}

miscellaneous3 {
  HighlightClassForRegexp {\m(\.[a-zA-Z0-9\_\-]+)+} {}
  HighlightClassWithOnlyCharStart \$ {}
}
```

Essentially this file is specifying the following about the Tcl language:

1. Any file that ends with .tcl or .msg should be parsed as a Tcl file.
2. If a Vim modeline is found with syntax=tcl, use this syntax highlighting information.
3. Auto-match curly brackets ({}), square brackets ([]), parenthesis (()) and double-quotes (“”).
4. Tab characters should not be used for indentation.
5. Use case sensitive matching for parsing purposes.
6. Whenever an open curly bracket is found, increase the indentation level, and whenever a closing curly bracket is found, decrease the indentation level.
7. Insert line comments with the HASH (#) character.
8. All comments start with the HASH (#) character.
9. All strings start and end with the QUOTE (“) character.
10. Apply keyword coloring to the list of keywords (ex., “after”, “bindtags”, “uplevel”, etc.)
11. Whenever a “proc” keyword is found, use the name of the proc as a searchable symbol in the file.
12. Highlight any integer values as numbers.
13. Highlight the], [, {, } characters as punctuation
14. There are no precompiler syntax to be colored.
15. Highlight Tk keywords in a different color than Tcl keywords.
16. Highlight Tcl/Tk option values in a different color than normal Tcl keywords.
17. Highlight Tk window pathnames in the miscellaneous3 color.
18. Highlight variables in the miscellaneous3 color.

Plugin Action Types

menu

Description

The menu plugin type allows the user to add a new menu command to the Plugins menu in the main application menubar. All menu plugins can optionally append any number of ".submenu" items to the menubar menu representing a cascading menu hierarchy within the menu that the command will be placed in. This allows the user to organize plugin menu items into groups, making the menu easier to find commands and easier to read/understand.

Tcl Registration

```
{menu command hierarchy do_procname handle_state_procname}
{menu {checkbutton variable} hierarchy do_procname handle_state_procname}
{menu {radiobutton variable value} hierarchy do_procname handle_state_procname}
{menu separator hierarchy}
```

The "menu command" type creates a menu command that, when clicked, runs the procedure called *do_procname*. The *hierarchy* value specifies the menu hierarchy (optional) and string text in the menu (joined with periods). The hierarchy will be created if it does not exist.

The "menu checkbutton" type creates a menu command has an on/off state associated with it. When the menu item is clicked, the state of the menu item is inverted and the *do_procname* procedure is called. The *variable* argument is the name of a variable containing the current on/off value associated with the menu item. The *hierarchy* value specifies the menu hierarchy (optional) and string text in the menu (joined with periods). The hierarchy will be created if it does not exist.

The "menu radiobutton" type creates a menu command that has an on/off state such that in a group of multiple menu items that share the same variable, only one is on at at time. When the menu item is clicked, the state of the menu item is set to on and the *do_procname* procedure is called. The *variable* argument is the name of a variable containing the menu item that is currently on. The *value* value specifies a unique identifier for this menu within the group. When the value of variable is set to value, this menu option will have the on state. The *hierarchy* value specifies the menu hierarchy (optional) and string text in the menu (joined with periods). The hierarchy will be created if it does not exist.

The "menu separator" type creates a horizontal separator in the menu which is useful for organizing menu options. The *hierarchy* value, in this case, only refers to the menu hierarchy to add the separator to (menu separators don't have text associated with them).

Tcl Procedures

The "do" Procedure

The "do" procedure contains the code that will be executed when the user invokes the menu item in the menubar.

Example:

```
proc foobar_menubar_do {} {
 puts "Foobar menu item has been clicked!"
}
```

The “handle_state” Procedure

The “handle_state” procedure is called when the Plugin menu is created (when the “Plugins” menubar item is clicked). This procedure is responsible for determining the state of the menu item to normal (1) or disabled (0) as deemed appropriate by the plugin creator.

Example:

```
proc foobar_menubar_handle_state {} {
 return $some_test_condition
}
```

tab_popup

Description

The tab_popup plugin type allows the user to add a new menu command to the popup menu in an editor tab. All tab_popup plugins can optionally append any number of “.submenu” items to the menubar menu representing a cascading menu hierarchy within the menu that the command will be placed in. This allows the user to organize plugin menu items into groups, making the menu easier to find commands and easier to read/understand.

Tcl Registration

```
{tab_popup command hierarchy do_procname handle_state_procname}
{tab_popup {checkbutton variable} hierarchy do_procname handle_state_procname}
{tab_popup {radiobutton variable value} hierarchy do_procname \
    handle_state_procname}
{tab_popup separator hierarchy}
```

The “tab_popup command” type creates a menu command that, when clicked, runs the procedure called *do_procname*. The *hierarchy* value specifies the menu hierarchy (optional) and string text in the menu (joined with periods). The hierarchy will be created if it does not exist.

The “tab_popup checkbutton” type creates a menu command has an on/off state associated with it. When the menu item is clicked, the state of the menu item is inverted and the *do_procname* procedure is called. The *variable* argument is the name of a variable containing the current on/off value associated with the menu item. The *hierarchy* value specifies the menu hierarchy (optional) and string text in the menu (joined with periods). The hierarchy will be created if it does not exist.

The “tab_popup radiobutton” type creates a menu command that has an on/off state such that in a group of multiple menu items that share the same variable, only one is on at at time. When the menu item is clicked, the state of the menu item is set to on and the *do_procname* procedure is called. The *variable* argument is the name of a variable containing the menu item that is currently on. The *value* value specifies a unique identifier for this menu within the group. When the value of variable is set to value, this menu option will have the on state. The *hierarchy* value specifies the menu hierarchy (optional) and string text in the menu (joined with periods). The hierarchy will be created if it does not exist.

The “tab_popup separator” type creates a horizontal separator in the menu which is useful for organizing menu options. The *hierarchy* value, in this case, only refers to the menu hierarchy to add the separator to (menu separators don’t have text associated with them).

Tcl Procedures

The “do” Procedure

The “do” procedure contains the code that will be executed when the user invokes the menu item in the menubar.

Example:

```
proc foobar_tab_popup_do {} {
 puts "Foobar tab popup item has been clicked!"
}
```

The “handle_state” Procedure

The “handle_state” procedure is called when the popup menu is created (when a right click occurs within a tab). This procedure is responsible for determining the state of the menu item of normal (1) or disabled (0) as deemed appropriate by the plugin creator.

Example:

```
proc foobar_menubar_handle_state {} {
  return $some_test_condition
}
```

root_popup

Description

The root_popup plugin type allows the user to add a new menu command to the popup menu in the sidebar when a root directory (i.e., a directory that doesn’t have a parent directory in the sidebar pane) is right-clicked. All root_popup plugins can optionally append any number of “_.submenu_” items to the menubar menu representing a cascading menu hierarchy within the menu that the command will be placed in. This allows the user to organize plugin menu items into groups, making the menu easier to find commands and easier to read/understand.

Tcl Registration

```
{root_popup command hierarchy do_procname handle_state_procname}
{root_popup {checkbutton variable} hierarchy do_procname handle_state_procname}
{root_popup {radiobutton variable value} hierarchy do_procname handle_state_procname}
{root_popup separator hierarchy}
```

The “root_popup command” type creates a menu command that, when clicked, runs the procedure called *do_procname*. The *hierarchy* value specifies the menu hierarchy (optional) and string text in the menu (joined with periods). The hierarchy will be created if it does not exist.

The “root_popup checkbutton” type creates a menu command has an on/off state associated with it. When the menu item is clicked, the state of the menu item is inverted and the *do_procname* procedure is called. The *variable* argument is the name of a variable containing the current on/off value associated with the menu item. The *hierarchy* value specifies the menu hierarchy (optional) and string text in the menu (joined with periods). The hierarchy will be created if it does not exist.

The “root_popup radiobutton” type creates a menu command that has an on/off state such that in a group of multiple menu items that share the same variable, only one is on at at time. When the menu item is clicked, the state of the menu item is set to on and the *do_procname* procedure is called. The *variable* argument is the name of a variable containing the menu item that is currently on. The *value* value specifies a unique identifier for this menu within the group. When the value of variable is set to value, this menu option will have the on state. The *hierarchy* value specifies the menu hierarchy (optional) and string text in the menu (joined with periods). The hierarchy will be created if it does not exist.

The “root_popup separator” type creates a horizontal separator in the menu which is useful for organizing menu options. The *hierarchy* value, in this case, only refers to the menu hierarchy to add the separator to (menu separators don’t have text associated with them).

Tcl Procedures

The “do” Procedure

The “do” procedure contains the code that will be executed when the user invokes the menu item in the menubar.

Example:

```
proc foobar_root_popup_do {} {
 puts "Foobar root popup item has been clicked!"
}
```

The “handle_state” Procedure

The “handle_state” procedure is called when the popup menu is created (when a right click occurs within a tab). This procedure is responsible for determining the state of the menu item of normal (1) or disabled (0) as deemed appropriate by the plugin creator.

Example:

```
proc foobar_root_popup_handle_state {} {
 return $some_test_condition
}
```

dir_popup

Description

The dir_popup plugin type allows the user to add a new menu command to the popup menu in the sidebar when any non-root directory is right-clicked. All root_popup plugins can optionally append any number of "_.submenu_" items to the menubar menu representing a cascading menu hierarchy within the menu that the command will be placed in. This allows the user to organize plugin menu items into groups, making the menu easier to find commands and easier to read/understand.

Tcl Registration

```
{dir_popup command hierarchy do_procname handle_state_procname}
{dir_popup {checkbutton variable} hierarchy do_procname handle_state_procname}
{dir_popup {radiobutton variable value} hierarchy do_procname handle_state_procname}
{dir_popup separator hierarchy}
```

The "dir_popup command" type creates a menu command that, when clicked, runs the procedure called *do_procname*. The *hierarchy* value specifies the menu hierarchy (optional) and string text in the menu (joined with periods). The hierarchy will be created if it does not exist.

The "dir_popup checkbutton" type creates a menu command has an on/off state associated with it. When the menu item is clicked, the state of the menu item is inverted and the *do_procname* procedure is called. The *variable* argument is the name of a variable containing the current on/off value associated with the menu item. The *hierarchy* value specifies the menu hierarchy (optional) and string text in the menu (joined with periods). The hierarchy will be created if it does not exist.

The "dir_popup radiobutton" type creates a menu command that has an on/off state such that in a group of multiple menu items that share the same variable, only one is on at at time. When the menu item is clicked, the state of the menu item is set to on and the *do_procname* procedure is called. The *variable* argument is the name of a variable containing the menu item that is currently on. The *value* value specifies a unique identifier for this menu within the group. When the value of variable is set to value, this menu option will have the on state. The *hierarchy* value specifies the menu hierarchy (optional) and string text in the menu (joined with periods). The hierarchy will be created if it does not exist.

The "dir_popup separator" type creates a horizontal separator in the menu which is useful for organizing menu options. The *hierarchy* value, in this case, only refers to the menu hierarchy to add the separator to (menu separators don't have text associated with them).

Tcl Procedures

The “do” Procedure

The “do” procedure contains the code that will be executed when the user invokes the menu item in the menubar.

Example:

```
proc foobar_dir_popup_do {} {
 puts "Foobar directory popup item has been clicked!"
}
```

The “handle_state” Procedure

The “handle_state” procedure is called when the popup menu is created (when a right click occurs within a tab). This procedure is responsible for determining the state of the menu item of normal (1) or disabled (0) as deemed appropriate by the plugin creator.

Example:

```
proc foobar_dir_popup_handle_state {} {
 return $some_test_condition
}
```

file_popup

Description

The file_popup plugin type allows the user to add a new menu command to the popup menu in the sidebar when any file is right-clicked. All file_popup plugins can optionally append any number of “_.submenu_” items to the menubar menu representing a cascading menu hierarchy within the menu that the command will be placed in. This allows the user to organize plugin menu items into groups, making the menu easier to find commands and easier to read/understand.

Tcl Registration

```
{file_popup command hierarchy do_procname handle_state_procname}
{file_popup {checkbutton variable} hierarchy do_procname handle_state_procname}
{file_popup {radiobutton variable value} hierarchy do_procname handle_state_procname}
{file_popup separator hierarchy}
```

The “file_popup command” type creates a menu command that, when clicked, runs the procedure called *do_procname*. The *hierarchy* value specifies the menu hierarchy (optional) and string text in the menu (joined with periods). The hierarchy will be created if it does not exist.

The “file_popup checkbutton” type creates a menu command has an on/off state associated with it. When the menu item is clicked, the state of the menu item is inverted and the *do_procname* procedure is called. The *variable* argument is the name of a variable containing the current on/off value associated with the menu item. The *hierarchy* value specifies the menu hierarchy (optional) and string text in the menu (joined with periods). The hierarchy will be created if it does not exist.

The “file_popup radiobutton” type creates a menu command that has an on/off state such that in a group of multiple menu items that share the same variable, only one is on at at time. When the menu item is clicked, the state of the menu item is set to on and the *do_procname* procedure is called. The *variable* argument is the name of a variable containing the menu item that is currently on. The *value* value specifies a unique identifier for this menu within the group. When the value of variable is set to value, this menu option will have the on state. The *hierarchy* value specifies the menu hierarchy (optional) and string text in the menu (joined with periods). The hierarchy will be created if it does not exist.

The “file_popup separator” type creates a horizontal separator in the menu which is useful for organizing menu options. The *hierarchy* value, in this case, only refers to the menu hierarchy to add the separator to (menu separators don’t have text associated with them).

Tcl Procedures

The “do” Procedure

The “do” procedure contains the code that will be executed when the user invokes the menu item in the menubar.

Example:

```
proc foobar_file_popup_do {} {
 puts "Foobar file popup item has been clicked!"
}
```

The “handle_state” Procedure

The “handle_state” procedure is called when the popup menu is created (when a right click occurs within a tab). This procedure is responsible for determining the state of the menu item as normal (1) or disabled (0) as deemed appropriate by the plugin creator.

Example:

```
proc foobar_file_popup_handle_state {} {
 return $some_test_condition
}
```

text_binding

Description

The text_binding plugin action creates a unique bindtag to all of the text and Ctext pathnames in the entire UI based on the location and name suffix specified and calls a plugin provided procedure with the name of that binding. The procedure can then add whatever bindings are required on the given bindtag name. This allows a plugin to handle various keyboard, focus, configuration, mapping and mouse events that occur in any editor.

Tcl Registration

`{text_binding location bindtag_suffix bind_type do_procedure}`

The value of *location* is either the value “pretext” or “posttext”. If a value of “pretext” is specified, any bindings on the text widget will be called prior to the text/cursor being applied to the text widget. If a value of “posttext” is specified, any bindings on the text widget will be called after the text/cursor has been applied to the widget.

The value of *bindtag_suffix* is any unique name for the plugin (i.e., if the plugin specifies more than one text_binding action, each action must have a different value specified for *bindtag_suffix*).

The value of *bind_type* is either “all” or “only”. A value of all means that the text binding will be added to all text widgets in the window. A value of “only” means that the text binding will only be added to the text widgets that have a tag list containing the value of *bindtag_suffix* (see api::file::add_buffer or api::file::add_file procedure for details about the -tag option. Using a value of “only” can only be used if the same plugin adds a file/buffer of its own.

The value of *do_procedure* is the name of the procedure that is called when a text widget is adding bindtags and bindings to itself and the given bindtag name has not been previously created.

Tcl Procedures

The “do” Procedure

The “do” procedure is called when the associated text bindtag is being initially created. The name of the associated bindtag created and applied by the plugin framework is passed to the procedure. The return value of the procedure is ignored. This procedure should only add various text bindings to associate functionality with different events that will occur on the text widgets.

The following example allows any changes to the cursor to invoke the procedure called “update_line”.

```
namespace eval current_line {
 proc do_cline {btag} {
          bind $btag <<Modified>>    “after idle [list current_line::update_line %W]”
          bind $btag <ButtonPress-1> “after idle [list current_line::update_line %W]”
          bind $btag <B1-Motion>     “after idle [list current_line::update_line %W]”
          bind $btag <KeyPress>      “after idle [list current_line::update_line %W]”
 }
 proc update_line {txt} {
          …
 }
}

api::register current_line {
 {text_binding pretext cline all current_line::do_cline}
}
```

on_start

Description

The on_start plugin action is called when the application starts. More precisely, the following actions will take place prior to running procedures associated with this action type.

- Preferences are loaded
- Plugins are loaded
- Snippet contents are loaded
- Clipboard history is loaded
- Syntax highlighting information is loaded
- User interface components are built (but not yet displayed to the user)

At this point, any on_start action procedures are run. The following events occur after this occurs.

- Command-line files are added to the interface
- Last session information is restored to the interface

The action type allows plugins to initialize or make user interface modifications.

Tcl Registration

```
{on_start do_procname}
```

The value of *do_procname* is the name of the procedure to run when Tcl is started.

Tcl Procedures

The “do” Procedure

The “do” procedure contains the code that will be executed when the application starts. It is passed no options and has no return value. You can perform any type of initialization within this procedure.

on_open

Description

The on_open plugin action is called after a new tab has been created and after the file associated with the tab has been read and added to the editor in the editor pane.

Tcl Registration

```
{on_open do_procname}
```

The *do_procname* is the name of the procedure that is called for this action type.

Tcl Procedures

The “do” Procedure

The “do” procedure is called when the file has been added to the editor. The procedure takes a single argument, the file index of the added file. You can use this file index to get various pieces of information about the added file using the api::file::get_info API procedure. The return value is ignored.

The following example will display the full pathname of a file that was just added to the editor.

```
proc foobar_do {file_index} {
 set fname [api::file::get_info $file_index fname]
 puts “File $fname was just opened”
}
```

on_focusin

Description

The on_focusin action type is called whenever a text widget receives input focus (i.e., the text widget’s tab was selected, the file was viewed by clicking the filename in the sidebar, etc.)

Tcl Registration

`{on_focusin do_procname}`

The *do_procname* procedure is called whenever focus is given to a text widget.

Tcl Procedures

The “do” Procedure

The “do” procedure is called whenever focus is given to a text widget. It is passed a single argument, the file index of the file that was given focus. This value can be used to get information about the file. The return value is ignored.

The following example displays the read-only status of the currently selected file.

```
proc focus_do {file_index} {
  if {[api::file::get_info $file_index readonly]} {
          puts “Selected file is readonly”
  } else {
          puts “Selected file is read/write”
  }
}
```

on_close

Description

The on_update plugin type allows the user to add an action to take whenever the contents of an editor is automatically updated by the application. This event is triggered when a file that is loaded into the editor is updated outside of the editor and focus is given back to the editor. If the file content within the editor is not in the modified state, TKE will automatically load the new file content and trigger this event. If the file content is in the modified state, a popup window will be presented to the user, letting them know that the file content has changed and asking them if they would like to accept the update or ignore it. If the user accepts the update request, the file content will be updated and this event will be triggered.

Tcl Registration

`{on_update do_procname}`

The value of *do_procname* is the name of the procedure that will be called after the file content has been updated in the editor.

Tcl Procedures

The “do” Procedure

The “do” procedure is called after a file is updated in the UI. It is passed a single argument, the file index of the file being closed. This argument value can be used to get information about the associated file. The return value is ignored.

The following example displays the name of the file that was updated.

```
proc foobar_do {file_index} {
 set fname [api::file::get_info $file_index fname]
 puts “File $fname has been updated”
}
```

on_quit

Description

The on_quit plugin type allows the user to add an action to take just before the tkdv session is quit. This can be used to perform file cleanup or other types of cleanup.

Tcl Registration

`{on_quit do_procname}`

The value of *do_procname* is the name of the procedure that will be called prior to the application quitting.

Tcl Procedures

The “do” Procedure

The “do” procedure contains the code that will be executed when the tkdv session is quit.

Example:

```
proc foobar_on_quit_do {} {
 file delete -force foobar.txt
}
```

on_reload

Description

The on_reload plugin type allows the user to store/restore internal data when the user performs a plugin reload operation. In the event that a plugin is reloaded, any internal data structures/state will be lost when the plugin is reloaded (re-sourced). The plugin may choose to store any internal data/state to non-corruptible memory within the plugin architecture just prior to the plugin being resourced and then restore that memory back into the plugin after it has been re-sourced.
Tcl Registration

`{on_reload store_procname restore_procname}`
The value of *store_procname* is the name of a procedure which will be called just prior to the reload operation taking place. The value of *restore_procname* is the name of a procedure which will be called after the reload operation has occurred.

Tcl Procedures

The “store” Procedure

The “store” procedure contains code that saves any necessary internal data/state to non-corruptible memory. It is called just prior to the plugin being re-sourced by the plugin architecture. The TKE API contains a procedure that can be called to safely store a variable along with its value such that the variable name and value can be restored properly.

Example:

```
proc foobar_on_reload_store {index} {
  variable some_data
  # Save the value of some_data to non-corruptible memory
  api::plugin::save_variable $index "some_data" $some_data
  # Save the geometry of a plugin window if it exists
  if {[winfo exists .mywindow]} {
     api::plugin::save_variable $index "mywindow_geometry" [winfo geometry .mywindow]
     destroy .mywindow
  }
```

```
}
```

In this example, we have a local namespace variable called “some_data” that contains some information that we want to preserve during a plugin reload. The example uses a user-available procedure within the plugin architecture called “api::plugin::save_variable” which takes three arguments: the unique identifier for the plugin (which is the value of the parameter called “index“), the string name of the value that we want to save, and the value to save. Note that the value must be given in a ”pass by value" format. The example also saves the geometry of a plugin window if it currently exists.

The “restore” Procedure

The “restore” procedure contains code that restores any previously saved information from the “store” procedure from non-corruptible memory back to the plugin memory. It is called immediately after the plugin has been re-sourced.

Example:

```
proc foobar_on_reload_restore {index} {
  variable some_data
  # Retrieve the value of some_data and save it to the
  # internal variable
  set some_data [api::plugin::load_variable $index "some_data"]
  # Get the plugin window dimensions if it previously existed
   set geometry [api::plugin::load_variable $index “mywindow_geometry”]
   if {$geometry ne ""} {
      create_mywindow
      wm geometry .mywindow $geometry
   }
}
```

In this example, we restore the value of some_data by calling the plugin architecture’s built-in “api::plugin::load_variable” procedure which takes two parameters (the unique index value for the plugin and the name of the variable that was previously stored) and returns the stored value (the procedure also removes the stored data from its internal memory). If the index/name combination was not previously stored, a value of empty string is returned. The example also checks to see if the mywindow geometry was saved. It it was it means that the window previously existed, so the restore will recreate the window (with an internal procedure called “create_mywindow” in this case) and the sets the geometry of the window to the saved value.

on_save

Description

The on_save action calls a procedure when a file is saved in the editor pane. Specifically, the action is called after the file is given a save name (the “fname” attribute of the file will be set) but before the file is actually written to the save file.

Tcl Registration

`{on_save do_procname}`

The *do_procname* value is a procedure that is called when this event occurs.

Tcl Procedures

The “do” Procedure

The “do” procedure is called when this event occurs. It is passed a single parameter value, the file index of the file being save. This value can be used in calls to the api::file::get_info to get information about the saved file. The return value is ignored.

The following example displays the name of the file being saved.

```
proc foobar_do {file_index} {
  set fname [api::file::get_info $file_index fname]
  puts “File $fname is being saved”
}
```

on_uninstall

Description

The on_uninstall action calls a procedure when the associated plugin is uninstalled by the user. Because uninstalling does not cause the application to quit (i.e., the UI remains in view), this plugin action allows the plugin writer to cleanup the UI that might have been affected by this plugin.

Tcl Registration

`{on_uninstall do_procname}`

The *do_procname* value is a procedure that is called when this event occurs.

Tcl Procedures

The “do” Procedure

The “do” procedure is called when this event occurs. No arguments are passed and the return value is ignored by the calling code. The body of this procedure should only be used to clean up any UI changes that this plugin may have previously made.

The following example removes a text tag called “foobar” that was previously added.

```
proc foobar_do {} {
  variable txt
  $txt tag delete foobar
}
```

on_rename

Description

The on_rename action calls a procedure when a file or directory is renamed within the sidebar. Specifically, this procedure will be called just prior to the rename being performed to allow the plugin to take any necessary actions on the given file/directory.

Tcl Registration

`{on_rename do_procname}`

The *do_procname* value is a procedure that is called when this event occurs.

Tcl Procedures

The “do” Procedure

The “do” procedure is called when this event occurs. Two arguments are passed. The first parameter is the full original pathname. The second parameter is the full new pathname. The return value is ignored.

The following example displays the original and new filenames.

```
proc foobar_do {old_name new_name} {
  puts “File $old_name has been renamed to $new_name”
}
```

on_duplicate

Description

The on_duplicate action calls a procedure immediately after a file or directory is duplicated within the sidebar.

Tcl Registration

`{on_duplicate do_procname}`

The *do_procname* value is a procedure that is called when this event occurs.

Tcl Procedures

The “do” Procedure

The “do” procedure is called when this event occurs. Two arguments are passed. The first parameter is the full original pathname. The second parameter is the full pathname of the duplicated file. The return value is ignored.

The following example displays the new filename.

```
proc foobar_do {orig_name new_name} {
  puts “File $orig_name has been duplicated ($new_name)”
}
```

on_delete

Description

The on_delete action calls a procedure just before a file is deleted from the file system within the sidebar.

Tcl Registration

`{on_delete do_procname}`

The *do_procname* value is a procedure that is called when this event occurs.

Tcl Procedures

The “do” Procedure

The “do” procedure is called when this event occurs. One parameter is passed — the full pathname of the file being deleted. The return value is ignored.

The following example displays the deleted filename.

```
proc foobar_do {name} {
  puts “File $name is deleted”
}
```

on_trash

Description

The on_trash action calls a procedure that is called prior to moving a file/folder to the trash.

Tcl Registration

`{on_trash do_procname}`

The *do_procname* value is a procedure that is called when this event occurs.

Tcl Procedures

The “do” Procedure

The “do” procedure is called when this event occurs. One parameter is passed — the full pathname of the file being moved to the trash. The return value is ignored.

The following example displays the trashed filename.

```
proc foobar_do {name} {
  puts “File $name is being moved to the trash”
}
```

syntax

Description

The syntax action specifies the name of a .syntax file that will be added to the list of available syntax highlighters in the UI.

Tcl Registration

`{syntax filename}`

The syntax filename must be located in the same directory as the main.tcl plugin file and it must contain the .syntax extension. The base name of the syntax file will be the name displayed in the UI. For a description of the contents of this file, please refer to the syntax file chapter of this document.

vcs

Description

The vcs action allows a plugin to provide the functionality required to create a new version control system handler for the TKE difference view. When a vcs action is created, the action’s name will appear in the version control list within a difference view, and that version control system will be checked to see if it manages an opened file when a difference view is requested.

This action allows a plugin to extend the supported version control systems available.

If your plugin contains the vcs action, you will need to request permission from the user to run your plugin as vcs plugin actions will be given filenames and will be required to run shell commands to perform necessary action.

Tcl Registration

`{vcs name handles versions get_file_cmd get_diff_cmd find_version get_version_log}`

The *name* option specifies the Version Control system name that will be displayed in the difference viewer version control list. The name does not need to match the version control system; however, it is preferred that the name does match to avoid user confusion.

Tcl Procedures

The “handles” Procedure

The “handles” procedure is given the full pathname of a file and must return a boolean value of true if the version control system is managing this file; otherwise, it must return a value of false. This procedure should be written in a performance optimized manner as it will be called after the user requests to view a file’s difference view and before the file difference is viewed.

The following example is from the vcs_example plugin which represents how a Mercurial plugin would operate.

```
proc handles {fname} {
  return [expr {![catch hg status $fname]}]
}
```

The “versions” Procedure

The “versions” procedure will return a Tcl list containing the version identifiers that are associated with the filename that is passed to the procedure.

```
proc versions {fname} {
   set versions [list]
   if {![catch { exec hg log $fname } rc]} {
       foreach line [split $rc \n] {
           if {[regexp {changeset:\s+(\d+):} $line -> version]} {
               lappend versions $version
           }
       }
   }
   return $versions
}
```

The “get_file_cmd” Procedure

Given the specified filename and version identifier, returns the command to execute which will return the full contents of the given version of the given filename.

```
proc get_file_cmd {fname version} {
  return “|hg cat -r $version $fname”
}
```

The “get_diff_cmd” Procedure

Given the specified filename and two versions, return the difference command that will output a unified difference between the two versions of the given file. The value of the v2 parameter can be a value of “Current” which should be interpreted as the version of the file that is currently being edited.

```
proc get_diff_cmd {fname v1 v2} {
  if {$v2 eq “Current”} {
     return “hg diff -r $v1 $fname”
  } else {
     return “hg diff -r $v1 -r $v2 $fname”
  }
}
```

The “find_version” Procedure

The “find_version” procedure will return the file version that contained the last change to the specified line number which is no later than the given version number. Keep in mind that the value of the v2 input parameter may be a value of “Current” which should be interpreted to be the version of the file that is currently being edited. If the change could not be found, return an empty string.

```
proc find_version {fname v2 linenum} {
  if {$v2 eq “Current”} {
     if {![catch { exec hg annotate $fname } rc]} {
        if {[regexp {^\s*(\d+):} [lindex [split $rc \n] [expr $linenum-1]] -> version]} {
           return $version
        }
     }
  } else {
     if {![catch { exec hg annotate -r $v2 $fname } rc]} {
        if {[regexp {^\s*(\d+):} [lindex [split $rc \n] [expr $linenum-1]] -> version]} {
           return $version
        }
     }
  }
  return “”
}
```

The “get_version_log” Procedure

The “get_version_log” procedure returns the change descriptions for the specified version of the specified filename. If no change description could be found, return the empty string.

```
proc get_version_log {fname version} {
  if {![catch { exec hg log -r $version $fname } rc]} {
     return $rc
  }
  return “”
}
```

on_pref_load

Description

The on_pref_load action is called shortly after the plugin is loaded/reloaded. The purpose of this action is to get a name/value list of preferences that are needed by the plugin. Preference values are stored in the same manner as TKE's internal preferences, allowing the user to set options that are remembered between application invocations. These items are also changed within the TKE preference window within the Plugins category.

Tcl Registration

`{on_pref_load do_procedure}`

The *do_procedure* is a Tcl procedure that will be called when TKE needs to get the preference values from the plugin.

Tcl Procedures

The "do" Procedure

The "do" procedure must return a valid Tcl list which contains pairs of name and default value values (thus it must contain an even number of elements). The following example creates two preference values and specifies that their default values should be 0 and "red":

```
proc do_pref_load {} {
  return {
     Enable 0
     Color  "red"
  }
}
```

do_pref_ui

Description

The "do_pref_ui" action is called when the preferences window needs to build the preference panel for the plugin. The procedure called by this action is responsible for generating the plugin's preference UI which is primarily performed using the api::preferences::widget and container widgets such as ttk::frame, ttk::labelframe, ttk::panedwindow and ttk::notebook.

Tcl Registration

`{do_pref_ui do_procedure}`

Tcl Procedures

The "do" Procedure

This procedure is responsible for creating and arranging the various GUI widgets that control the layout of the plugin's preferences panel. The following is a representation of what the body of the procedure might look like:

```
proc do_pref_ui {w} {
  api::preferences::widget checkbutton $w "Enable" "Enables this plugin”
  api::preferences::widget spacer $w
  pack [ttk::labelframe $w.lf -text "Color"] -fill x
  api::preferences::widget radiobutton $w.lf "Color" "red" -value “red”
  api::preferences::widget radiobutton $w.lf "Color" "white" -value “white”
  api::preferences::widget radiobutton $w.lf "Color" "blue" -value “blue”
}
```

This plugin creates a check button which allows the “Enable” preference item to be set. It also adds three radio buttons which allow the user to select one of three colors to set the “Color” preference item. The three radio buttons are placed into a ttk::labelframe container widget. Between the check button and the radio buttons is vertical space. This code will result in a preference panel that will look something like the following:

If the do_pref_ui procedure creates any widgets outside of those provided for in the api::preferences::widget API procedure, it is the responsibility of the do_pref_ui procedure to handle packing. All widgets created with the api::preferences::widget API procedure will pack themselves in the order they are created, using either pack (default) or grid (using the -grid 1 option to api::preferences::widget). See the api::preferences::widget description for a full explanation of the available widgets that can be created with that procedure.

Plugin API

api::tke_development

Specifies if the application is being run in development mode or normal user mode. This can be useful if your plugin is meant to be used for TKE development purposes only.

Call structure

```
api::tke_development
```

Return value

Returns a value of 1 if the application is being run in development mode; otherwise, returns a value of 0.

api::get_plugin_directory

Returns the full pathname of the TKE installation directory.

Call structure

```
api::get_plugin_directory
```

Return value

Returns the full pathname of the TKE plugin installation directory for the calling plugin.

api::get_images_directory

Returns the full pathname to the directory which contains all of the plugin images used by TKE in the installation directory.

Call structure

```
api::get_images_directory
```

Return value

Returns the full pathname to the directory which contains all of the plugin images used by TKE in the installation directory.

api::get_images_directory

Returns the full pathname to the directory which contains all of the plugin images used by TKE in the installation directory.

Call structure

```
api::get_images_directory
```

Return value

Returns the full pathname to the directory which contains all of the plugin images used by TKE in the installation directory.

api::get_home_directory

This procedure returns the full pathname to the plugin-specific home directory. If the directory does not currently exist, it will be automatically created when this procedure is called. The plugin-specific home directory exists in the user's TKE home directory under ~/.tke/plugins/name_of_plugin. This directory will be unique for each plugin so you may store any plugin-specific files in this directory.

Call structure

```
api::get_home_directory
```

Return value

Returns the full pathname to the plugin-specific home directory.

api::normalize_filename

Takes a NFS-attached host where the file resides along with the pathname on that host where the file resides and returns the normalized filename to access the file on the current host.

Call structure

```
api::normalize_filename host filename
```

Return value

Returns the normalized pathname of the given file relative to the current host machine.

Parameters

Parameter	Description
host	Name of host server where the file actually resides.
filename	Name of file on the host server.

api::register_launcher

Registers a plugin command with the TKE command launcher. Once a command is registered, the user can invoke the command from the command launcher by entering a portion of the description string that is passed via this command.

Call structure

```
api::register_launcher description command
```

Return value

None

Parameters

Parameter	Description
description	String that is displayed in the command launcher matched results. This is also the string that is used to match against.
command	Command to run when this entry is executed via the command launcher.

api::unregister_launcher

Unregisters a previously registered command from the command launcher.

Call structure

```
api::unregister_launcher description
```

Return value

None

Parameters

Parameter	Description
description	The initial description string that was passed to the api::register_launcher command.

api::invoke_menu

Invokes the functionality associated with a menu item. This allows plugins to perform “workflows”. The menu hierarchy is defined by taking the names of all menus in the hierarchy (case is important) and joining them with the “/“ character. Therefore, to invoke the File -> Format Text -> All command, you would pass the following:

```
api::invoke_menu “File/Format Text/All”
```

Call structure

```
api::invoke_menu menu_hierarchy_path
```

Return value

None. Returns an error if the given menu hierarchy string cannot be found.

Parameters

Parameter	Description
menu_hierarchy_path	String representing the hierarchical menu path to the menu command to execute. Each portion of the menu path must match the menu exactly and all menu portions must be joined by the “/“ character.

api::log

Displays a logging message to standard output. This is useful for debugging plugin issues.

Call structure

```
api::log message
```

Return value

None.

Parameters

Parameter	Description
message	Message to display to standard output.

api::show_info

Takes a user message and a delay time and displays the message in the bottom status bar which will automatically clear from the status bar after the specified period of time has elapsed. This is useful for communicating status information to the user, but should not be used to indicate error information (the api::show_error procedure should be used for this purpose).

Call structure

```
api::show_info message ?clear_delay?
```

Return value

None

Parameters

Parameter	Description
message	Message to display to user. It is important that no newline characters are present in this message and that the message is no more than 100 or so characters in length.
clear_delay	Optional value. Allows for the message to be cleared in “clear_delay” milliseconds. By default, this value is set to 3000 milliseconds (i.e., 3 seconds).

api::show_error

Takes a user message and optional detail information and displays the message in a popover window. This window is the standard error window used by TKE internal code and, therefore, is the recommended way to display error information to the user.

Call Structure

```
api::show_error message ?detail?
```

Return value

None

Parameters

Parameter	Description
message	Short error message.
detail	Optional. Detailed error description.

api::get_user_input

Displays a prompt message and an entry field, placing the cursor into the entry field for immediate text entry. Once a value has been input, the value will be assigned to the variable passed to this procedure. Allows the plugin to get user input.

Call structure

```
api::get_user_input message variable ?allow_vars?
```

Return value

Returns a value of 1 if the user hit the RETURN key in the text entry field to indicate that a value was obtained by the user and stored in the provided variable. Returns a value of 0 if the user hit the ESCAPE key or clicked on the close button to indicate that the value of variable was not set and should not be used.

Parameters

Parameter	Description
message	Message to prompt user for input (should be short and not contain any newline characters).
variable	Name of variable to store the user-supplied response in. If a value of 1 is returned, the contents in the variable is valid; otherwise, a return value 0 indicates the contents in the variable is not valid.
allow_vars	Optional. If set to 1, any environment variables specified in the user string will have value substitution performed and the resulting string will be stored in the variable parameter. If set to 0, no substitutions will be performed. By default, substitution is performed.

Example

```
set filename “”
if {[api::get_user_input “Filename:” filename 1]} {
  puts “File $filename was given”
} else {
  puts “No filename specified”
}
```

api::file::current_file_index

Returns a unique index to the currently displayed file in the editor. This index can be used for getting information about the file.

Call structure

`api::file::current_file_index`

Return value

Returns a unique index to the currently displayed file in the editor panel.

api::file::get_info

Returns information about the file specified by the given index based on the attribute that this passed.

Call structure

```
api::file::get_info file_index attribute
```

Return value

Returns the attribute value for the given file. If an invalid *file_index* is specified or an invalid attribute is specified, an error will be thrown.

Parameters

Parameter	Description
file_index	Unique identifier for a file as returned by the get_current_index procedure.
attribute	Specifies the type of information to return. See the attribute table below for the list of valid values.

Attributes

Attribute	Description
fname	Normalized file name.
mtime	Last modification timestamp.
lock	Specifies the current lock status of the file.
readonly	Specifies if the file is readonly.
modified	Specifies if the file has been modified since the last save.
sb_index	Specifies the index of the file in the sidebar.
txt	Returns the pathname of the text widget associated with the file_index.
current	Returns a boolean value of true if the file is the current one being edited.
vimmode	Returns a boolean value of true if the editor is in “Vim mode” (i.e., any Vim mode that is not insert/edit mode).
lang	Returns the name of the syntax language associated with the given file.

api::file::add_buffer

Adds a new tab to the editor which is a blank editing buffer (no file is associated with the contents of the editing buffer).

Call structure

```
api::file::add_buffer name save_command ?options?
```

Return value

Returns the pathname of the created ctext widget.

Parameters

Parameter	Description
name	Title of editor tab.
save_command	Command to run when the user attempts to save the contents of the buffer. If the save command returns a value of 1, TKE will prompt the user for a filename and the contents will be saved to the specified file. From that point on, the editing buffer will transition to a normal file and the save command will no longer be invoked on future saves. If the save command returns a value of 1, the save_command will continue to be used for future saves.
options	Optional arguments passed to the newly added tab. See the list of valid options in the table below.

Options

Option	Description
-lock *boolean*	Initial value of the lock setting of the buffer (the user does have permission to unlock the file).
-readonly *boolean*	Specifies if the buffer will be editable by the user.
-gutters *gutter_list*	Creates one or more gutters in the editor (one character wide vertical strip to the left of the line number gutter which allows additional information/functionality to be provided for each line in the editor). See the gutter_list description below for additional details about the structure of this option.
-other *boolean*	Specifies if the buffer should be added to pane that does not currently have the focus.
-tags *tags*	Specifies a list of text bindings that can only be associated with this tab.
-lang *language*	Specifies the initial syntax highlighting language to use for highlighting the buffer.
-background *boolean*	If true, causes the added buffer tab to be created but not made the current editing buffer; otherwise, if false, the tab will be made the current tab.

api::file::add_file

Adds a new tab to the editor. If a filename is specified, the contents of the file are added to the editor. If no filename is specified, the new tab file will be blanked and named “Untitled”.

Call structure

```
api::file::add_file ?filename? ?options?
```

Return value

Returns the pathname of the created ctext widget.

Parameters

Parameter	Description
filename	Optional. If specified, opens the given file in the added tab.
options	Optional arguments passed to the newly created tab. See the table below for a list of valid values.

Options

Option	Description
-savecommand *command*	Specifies the name of a command to execute after the file is saved.
-lock *boolean*	If set to 0, the file will begin in the unlocked state (i.e., file is editable); otherwise, the file will begin in the locked state.
-readonly *boolean*	If set to 1, the file will be considered readonly (file will be indefinitely locked); otherwise, the file will be editable.
-sidebar *boolean*	If set to 1 (default), the file's directory contents will be included in the sidebar; otherwise, the file's directory components will not be added to the sidebar.
-diff *boolean*	If set to 0 (default), the file will be added as an editable file; however, if set to 1, the file will be inserted as a difference viewer, allowing the user to view file differences visually within the editor.
-gutters gutter_list	Creates one or more gutters in the editor (one character wide vertical strip to the left of the line number gutter which allows additional information/functionality to be provided for each line in the editor). See the gutter_list description below for additional details about the structure of this option.
-other *boolean*	If set to 0 (default), the file will be added to the current pane; however, if set to 1, the file will be added to the other pane (the other pane will be created if it currently does not exist).
-tags *tags*	Specifies a list of text bindings that can only be associated with this tab.
-name *filename*	If this option is specified when the filename is not specified, it will add a new tab to the editor whose name matches the given name. If the user saves the file, the contents will be saved to disk with the given file name. The given filename does not need to exist prior to calling this procedure.

api::sidebar::get_selected_indices

Returns the index of the currently selected files/directories in the sidebar. This value is useful for future calls to the api::sidebar::get_info procedure.

Call structure

```
api::sidebar::get_selected_indices
```

Return value

A list of integer values specifying the indices of the currently selected files/directories in the sidebar. An empty list will be returned if no file/directory is currently selected.

Parameters

None.

api::sidebar::get_info

Returns information for the sidebar file/directory at the given index.

Call structure

```
api::sidebar::get_info sb_index attribute
```

Return value

Returns the value associated with the given index/attribute. If the specified index is not a valid index value for the sidebar, an empty string will be returned.

Parameters

Parameter	Description
sb_index	Index of file/directory in the sidebar. Calling api::sidebar::get_current_index will provide the currently selected element in the sidebar.
attribute	Specifies the type of information to obtain for the given index. See the list of valid values in the table below.

Attributes

Attribute	Description
fname	Normalized filename.
file_index	The file index of the file. This value can be used in the api::file::get_info API call to get other information about the file.

api::plugin::save_variable

Saves the value of the given variable name to non-corruptible memory so that it can be later retrieved when the plugin is reloaded.

Call structure

```
api::plugin::save_variable index name value
```

Return value

None.

Parameters

Parameter	Description
index	Unique index provided by the plugin framework (passed to the writeplugin action command).
name	Name of a variable to save.
value	Value of a variable to save.

api::plugin::load_variable

Retrieves the value of the named variable from non-corruptible memory (from a previous save_variable call).

Call structure

```
api::plugin::load_variable index name
```

Return value

Returns the saved value of the given variable. If the given variable name does not exist, an empty string will be returned.

Parameters

Parameter	Description
index	Unique index provided by the plugin framework (passed to the readplugin action command).
name	Name of a variable to retrieve the value for.

api::utils::open_file

Opens a file in the default external web browser.

Call structure

```
api::utils::open_file filename ?in_background?
```

Return value

Returns a boolean value of true if the file was successfully opened; otherwise, returns false.

Parameters

Parameter	Description
filename	Name of file to display in an external web browser.
in_background	Optional. If set to a boolean value of true, keeps the focus within TKE (i.e., opening web browser in the background). If set to a bool value of false, changes the focus to the web browser window.

api::get_default_foreground

Returns the color associated with the foreground color of a standard UI element (i.e., a button).

Call structure

```
api::get_default_foreground
```

Return value

Returns an RGB color value.

Parameters

None.

api::get_default_background

Returns the color associated with the background color of a standard UI element (i.e., a button).

Call structure

```
api::get_default_background
```

Return value

Returns an RGB color value.

Parameters

None.

api::color_to_rgb

Returns a list containing three values, the R, G and B integer values of the specified color.

Call structure

`api::color_to_rgb color`

Return value

Returns the R, G and B color values of the input color.

Parameters

Either an RGB color value (i.e., #RRGGBB or #RGB) or a valid color name as specified in the TK colors list.

api::get_complementary_mono_color

Assuming that the specified RGB color is a background color, determines whether a black or white foreground color would show up best.

Call structure

`api::get_complementary_mono_color color`

Return value

Returns a color value of either “white” or “black”.

Parameters

The value of color should be a valid RGB color.

api::rgb_to_hsv

Converts the given RGB color into an HSV value.

Call structure

`api::rgb_to_hsv R G B`

Return value

Returns a list containing the H, S and V values of the given RGB color.

Parameters

The R, G and B values must be a value between 0 and 255, inclusive.

api::hsv_to_rgb

Converts the given HSV color into an RGB value.

Call structure

```
api::hsv_to_rgb H S V
```

Return value

Returns a list containing the R, G and B values of the given HSV color.

Parameters

The H, S and V values must be an integer value.

api::rgb_to_hsl

Converts the given RGB color to an equivalent HSL color value.

Call structure

```
api::rgb_to_hsl R G B
```

Return value

Returns a list containing the H, S and L values of the given RGB color value.

Parameters

The R, G and B colors must be an integer value between 0 and 255, inclusive.

api::hsl_to_rgb

Converts the given HSL color to an equivalent RGB color value.

Call structure

```
api::hsl_to_rgb H S L
```

Return value

Returns a list containing the R, G and B values of the given HSL color value.

Parameters

The H, S and L color values must be integer values.

api::get_color_values

Returns a number of different versions of the given RGB color value.

Call structure

```
api::get_color_values color
```

Return value

Returns a list of five elements described as follows:

1. The V value from the HSV value.
2. The R value from the RGB value.
3. The G value from the RGB value.
4. The B value from the RGB value.
5. The RGB value expressed as #RRGGBB form.

Parameters

The color can be any legal RGB value accepted by TK.

api::auto_adjust_color

Automatically adjusts the given RGB color by a value equal to diff such that if the color is a darker color, the value will be lightened or if a color is a lighter color, the value will be darkened.

Call structure

```
api::auto_adjust_color color diff ?mode?
```

Return value

Returns an RGB color value in the #RRGGBB format.

Parameters

The color value is any legal RGB color value. The diff value is an integer value that describes the value difference to create from the color value. The mode value can be either “auto” (default) or “manual”. The auto mode will automatically discern the darkness of the color value and lighten or darken the color by the given value of diff. The manual mode will change the color value by the amount specified by diff (a negative value will darken the value while a positive value will lighten the value).

api::auto_mix_colors

Adjusts the hue of the given RGB color by the value of the specified difference.

Call structure

```
api::auto_mix_colors color type diff
```

Return value

Returns an RGB color value in the #RRGGBB format.

Parameters

The color value is any legal TK RGB color. The value of type can be either “r”, “g” or “b” which will adjust the RGB color value by the given diff amount.

api::color_difference

Returns the RGB color value which is exactly between two specified RGB colors.

Call structure

```
api::color_difference color1 color2
```

Return value

Returns an RGB color value in the #RRGGBB format.

Parameters

Both color1 and color2 must be valid RGB TK values.

api::preferences::widget

Creates a preferences widget which controls the display and manipulation of one plugin preference value. The created widget is automatically added for searching within the preferences window and it is packed into the plugin’s preference panel using either pack (default) or grid (using the -grid 1 option). Several types of widgets are supported:

- checkbutton
- radiobutton
- menubutton
- spinbox
- tokenentry
- entry
- text

There is also a spacer widget for creating additional vertical division between components for improving readability within the panel.

Call Structure

```
api::preferences::widget type parentwin prefname message options
```

Return Value

Returns the pathname of the main widget which allows you to customize the widget details, if desired.

Parameters

Parameter	Description
type	Specifies the type of widget to create and pack. See the table below for the list of legal values.
parentwin	Pathname of parent window to pack the widget into.
prefname	Name of preference value that is controlled by this widget. Note: This field is not valid for the spacer type.
message	Message to display in the widget.

Widget Types

Widget	Usage
checkbutton	Useful for preference items that have a boolean (on or off) value.
radiobutton	Useful for preference items that have a relatively small number of enumerated values.
menubutton	Useful for preference items that have a relatively large number of enumerated values.
spinbox	Useful for preference items that have an integer value within a specified range.
entry	Useful for preference items that have a single string value that can be input in a single line.
token	Useful for preference items that have one or more string values that are not enumerated.
text	Useful for preference items that have a single string value that may require newlines.
spacer	Only used for inserting vertical whitespace in the preference panel. This widget does not represent a preference value.

Options

Option	Default Value	Description
-value *value*	none	Only valid for the radiobutton type. This value will be assigned to the preference item if the radiobutton is selected.
-values *value_list*	none	Only valid for the menubutton type. The list of values will be displayed in the dropdown menu when the menubutton is clicked. The selected value will be assigned to the preference item.
-watermark *string*	none	Only valid for the entry and token types. This string will be displayed in the entry field when no other text is entered.
-from *number*	none	Only valid for the spinbox type. Specifies the lowest legal value that the spinbox can be set to.
-to *number*	none	Only valid for the spinbox type. Specifies the highest legal value that the spinbox can be set to.
-increment *number*	1	Only valid for the spinbox type. Specifies the amount that will be added/subtracted from the current value when the up/down arrow is clicked in the spinbox.
-grid *number*	0	If set to 0, the widget will be packed using the Tk pack manager. If set to 1, the widget will be packed using the Tk grid manager.

Message Display

The following table describes how the message parameter will be displayed in relation to its widget.

Widget	Message Layout
checkbutton	Message is displayed to the left of the check box.
radiobutton	Message is displayed to the left of the radio button.
menubutton	Message is displayed to the left of the menu button.
spinbox	Message is displayed to the left of the spin box.
entry	Message is displayed just above the entry field.
token	Message is displayed just above the entry field.
text	Message is displayed just above the text field.

api::preferences::get_value

Returns the current value of the input plugin preference.

Call structure

```
api::preferences::get_value prefname
```

Return value

Returns the current value of the associated preference item.

Parameters

The *prefname* specifies the preference item name to lookup. This must be one of the names returned from the on_pref_load plugin action. If the value of *prefname* was not found, returns an error.

Ctext Widget

The Ctext widget that is supplied with TKE is a custom version that is originally based on the original 4.0 version of ctext. Due to the significant number of changes to the usage API, it makes sense to document the widget in this document for purposes of plugin development.

The Ctext widget is essentially a wrapper around the Tk text widget. All text widget commands and options are valid for the Ctext widget. The documentation for these options and commands are not provided in this document (read the Tk text documentation for more details). Instead, all Ctext-specific options and commands will be documented in this appendix. Any deviations of Tk text options, commands and bindings will also be documented in this appendix.

Differences from the original Ctext widget

Like the original Ctext widget, the main purpose of the new Ctext widget is to allow text to be edited such that syntax highlighting is actively performed while the user enters information in the editor. It also provides line numbering support.

In addition to these functions, the new Ctext widget provides several new functions:

- Gutter support
  - In addition to line numbers, the Ctext widget provides a set of APIs to add additional programmable gutter information such that each line can be tagged with a symbol and/or colors in the gutter area to convey additional information about the line. Each symbol displayed can receive on_leave, on_enter and on_click events and execute a command when the user causes any of the events to occur.
  - Each gutter operates independently of other gutters in the same Ctext widget.
  - Gutter tagging stays on the assigned line even if the line changes to a new location due to text being inserted or deleted from the Ctext widget.
- Difference mode
  - Displays a new version of the gutter which shows two sets of line numbers per line.
  - Enables the ‘diff’ command API (more on this command in the command section of this appendix) to allow the developer to mark lines as changed which are displayed visually to the user.
  - Change lines can respond to developer supplied on_enter, on_leave, and on_click events.
- Enhanced syntax highlighting capabilities
  - Though the original Ctext widget provides several syntax highlighting API procedures, languages like HTML, XML and Markdown require more complex syntax highlighting requirements which require more than a single regular expression to properly support.
  - The new Ctext widget provides the ability to handle more complex syntax by specifying a callback procedure that is called when a certain kind of syntax is detected via a regular expression.
  - The return value from the callback procedure can cause the syntax parser current index to be set to a value less than or greater than the index returned from the original regular expression check.
  - In addition to highlighting text, the new Ctext widget can allow text to be clickable, underlined, italicized, emboldened, overstriken, superscripted, subscripted, and sized to six different sizes. If the gutter is displayed, any line height changes are automatically reflected in the gutter such that all line numbers and symbols in the gutter correlate to the lines in the editing area.
- Customized undo/redo support

    - Due to limitations in the Tk text widget support for the undo/redo buffer, the Ctext widget provides its own undo/redo function that provides an enhanced API for querying information about the undo/redo stack, provides support for cursor memory and re-positioning, and provides all of this while remaining extremely efficient in terms of memory and performance.
- Enhanced <> event support
    - The %d variable contains a list of information about what caused the widget to go modified, including:
        - operation: 'insert' or 'delete' (replace operations cause two modify events, one for the delete and one for the insert)
        - starting cursor position
        - number of characters inserted or deleted
        - number of lines inserted or deleted
        - an optional, user-specified list of information such that any insert, delete and replace commands can pass specialized information to the callback procedure handling the modified event.
    - Note: Because TKE specifies the modify callback procedure, this is not a feature that is useful for TKE development but rather for any other project that wishes to use the version of Ctext provided with TKE.
- Optimized
    - The new Ctext widget syntax highlighter and linemap code has been modified to optimize performance while editing to make the widget feel snappier and more usable.
    - Many improvements have been also made to improve the correctness of highlighting and bracket matching.
    - Includes support for the replace command such that undo/redo will behave as expected as well as provides the proper syntax highlighting support.
- New version number
    - The new Ctext widget provided with TKE is set to version 5.0. The original version set its version number to 4.0. This allows development environments to use both widgets, if necessary (although 5.0 is a superset of 4.0 such that 4.0 should no longer be necessary to use).

Options

The following options are available in the Ctext widget.

Option	Default Value	Description
-highlightcolor *color*	yellow	Specifies the color that will be drawn around the outside of the editor window when the window has keyboard input focus.
-unhighlightcolor *color*	None	Specifies the color that will be drawn around the outside of the editor window when the window does not have keyboard input focus.
-linemap *boolean*	TRUE	If true displays the line number information in the gutter; otherwise, hides the line number information.
-linemapfg *color*	Same as the value of the -fg	Specifies the color of the foreground in the line information in the gutter.

	the -fg option.	
-linemapbg *color*	Same as the value of the -bg option.	Specifies the color of the background in the line information in the gutter.
-linemap_mark_command *command*	None	Specifies a command to execute when the user creates a marker in the gutter area. The command has the following information appended to this command: pathname of the ctext widget, a value of “marked” (if a marker was added to the gutter) or “unmarked” (if a marker was deleted from the gutter), and the tag name of the marker created.
-linemap_markable *boolean*	TRUE	If true, specifies that the linemap can be clicked on to create and remove markers. If -linemap is false when this option is true, the line numbers will not be displayed, but a one character area will be visible allowing the user to click on a line to create a visible marker.
-linemap_select_fg *color*	black	Specifies the foreground color of line numbers when they are marked with a marker.
-linemap_select_bg *color*	yellow	Specifies the background color of line numbers when they are marked with a marker.
-linemap_cursor *cursor*	left_ptr	Specifies the name of a Tk cursor to display when the mouse cursor is within the linemap gutter area.
-linemap_relief *relief*	Same value as the -relief text option	Specifies the relief to to use when displaying the linemap area.
-linemap_minwidth *number*	1	Specifies the minimum number of characters to show for line numbers. If the number of characters required to display the last line number of the current file exceeds this value, that value is used instead.
-linemap_type (**absolute** or **relative**)	absolute	Specifies if line numbering should use absolute line numbering (i.e., the first line of the file is numbered 1 and subsequent line numbers increment from there) or relative line numbering (i.e., the line containing the insertion cursor is numbered 0 with lines above incrementing by 1 from 0 and lines below the current line incrementing by 1).
-warnwidth *number*	None	If set to a positive number, sets the width warning line just after the specified text column. If set to the empty string, the warning line will be removed from the display.

-warnwidth_bg *color*	red	Specifies the color of the width warning line (if it is displayed).
-casesensitive *boolean*	1	Specifies the case-sensitivity of the syntax highlight parser.
-maxundo *number*	0	Specifies the maximum number of undo operations stored in memory. If this value is set to 0, unlimited undo is supported (if the Tk text -undo option is set to a value of true).
-diff_mode *boolean*	0	If set to true, runs the Ctext editor in diff mode. In this mode, the gutter displays two sets of line numbers (one for each file in the diff). This option also enables the diff commands (documented in the Commands section of this appendix). If set to false, runs the editor in normal editing mode. Note that diff mode still allows syntax highlighting to occur. Additionally, if we are running in diff mode, the line number gutter will always be displayed regardless of the value of -linemap.
-diffsubbg *color*	pink	Specifies the background color of difference lines from the first file (i.e., lines that are not a part of the second file).
-diffaddbg *color*	{light green}	Specifies the background color of difference lines from the second file (i.e., lines that are not a part of the first file).

Command API

Like the widget options, only those commands which differ from the standard Tk text widget will be included in this chapter.

delete

The delete command works almost exactly the same as the standard text delete command with one exception, it can accept an optional “-moddata” option. The moddata option allows the user to pass user-specific information to the callback procedure that handles the <> virtual event.

In TKE, there is only one value that is used for -moddata, the value of “ignore”. Adding the “-moddata ignore” option will cause the deletion to not change the modified state of the text widget. This allows plugin code to delete data from the buffer without making it look like the user modified the contents of the buffer.

Call structure

```
pathname delete ?–moddata value? index1 ?index2?
```

Return value

None.

diff reset

This command must be called prior to calling any of the other diff-related commands. If the difference information needs to be changed (i.e., one of the files in the difference is changed), this command must be called to remove all embedded difference information and reset the widget.

Call structure

`pathname diff reset`

Return value

None.

Parameters

None.

Example

```
proc apply_diff {txt} {
  # Reset the widget for difference display
  $txt diff reset
  # Show that the second file had two lines added, starting
  # at line 5
  $txt diff add 5 2
}
```

diff ranges

Returns a list of text indices that specify the start and end ranges of text marked as different in the Ctext widget. The differences from the first file (sub), second file (add) or both can be returned. All indices are returned in index order.

Call structure

`pathname diff ranges type`

Return value

A Tcl list containing an even number of text indices specifying the start and end of each difference range.

Parameters

Parameter	Description
type	One of three values: • sub = returns the ranges of all differences in the first file • add = returns the ranges of all differences in the second file • both = returns the ranges of all differences in both files

diff sub

Adds a difference change for the first file (as it would be displayed in unified diff format). Before any calls can be made to this command, the diff reset command must be called.

When the editor operates in diff mode, it is important the the second file in the diff is displayed in the Ctext widget in its entirety. The ‘diff sub’ command contains an extra parameter when called which contains the text that exists in the first file but not in the second file. This text is inserted into the widget at the given line number and line numbers in the gutter are adjusted accordingly. All inserted text will be highlighted in the background color specified by the -diffsubbg option.

Call Structure

```
pathname diff sub startline linecount text
```

Return value

None.

Parameters

Parameter	Description
startline	Starting line containing file difference information.
linecount	Specifies the number of lines that will be inserted into the widget.
text	String containing text that exists in the first file but not in the second file. If a unified diff file is parsed, this string would be all contiguous lines prefixed by a single ‘-‘ character in the output.

diff add

Like its other diff command ‘diff sub’, the ‘diff add’ command marks lines in the text widget as being different from the first file. All lines marked with this command are highlighted using the background color specified by the -diffaddbg option. After this command has been called, the line numbers in the linemap area will be automatically changed to match the difference information.

Call structure

```
pathname diff add startline linecount
```

Return value

None.

Parameters

Parameter	Description
startline	Starting line containing file difference information.
linecount	Specifies the number of lines that will be inserted into the widget.

fastdelete

This command performs a standard deletion without repairing any syntax highlighting due to removing the text.

Call structure

```
pathname fastdelete ?options? startpos endpos
```

Return value

None.

Parameters

Parameter	Description
options	• -moddata *data* = Value that will be passed to any <<Modified>> callback procedures via the %d bind variable. • -update *bool* = If set to 1 will cause the <<Modified>> and <<CursorChanged>> events to be triggered; otherwise, these events will not be generated.
startpos	Starting text position of character range to delete.
endpos	Ending test position of character range to delete.

fastinsert

This command performs a standard insertion without performing any syntax highlighting on the inserted text.

Call structure

```
pathname fastinsert ?options? startpos text
```

Return value

None.

Parameters

Parameter	Description
options	• -moddata *data* = Value that will be passed to any <<Modified>> callback procedures via the %d bind variable. • -update *bool* = If set to 1 will cause the <<Modified>> and <<CursorChanged>> events to be triggered; otherwise, these events will not be generated.
startpos	Starting text position to begin inserting the given text.
text	Text to insert.

fastreplace

This command performs a standard text replacement without performing any syntax highlighting on the inserted text.

Call structure

```
pathname fastreplace ?options? startpos endpos text
```

Return value

None.

Parameters

Parameter	Description
options	• -moddata *data* = Value that will be passed to any <<Modified>> callback procedures via the %d bind variable. • -update *bool* = If set to 1 will cause the <<Modified>> and <<CursorChanged>> events to be triggered; otherwise, these events will not be generated.
startpos	Starting position of text range to delete and starting position to insert text.
endpos	Ending position of text range to delete.
text	Text to insert in replacement of the deleted text.

highlight

Causes all text between the specified line positions to be re-evaluated for syntax highlighting. This is most often used after a number of fastdelete or fastinsert commands are called. This eliminates a lot of syntax highlighting calls which can improve the performance of the widget in certain situations.

Call structure

```
pathname highlight ?options? startpos endpos
```

Return value

None.

Parameters

Parameter	Description
options	• -moddata *data* = Value that will be passed to any <<Modified>> callback procedures via the %d bind variable. • -insert *bool* = Set to a value of 1 if the highlight is being performed after inserting text; otherwise, set it to a value of 0. • -dotags *list* = Set to any string value to force comment/string parsing to be performed during the highlight process.
startpos	Character position index to begin highlighting. The starting character position will be the beginning of the line of this character.
endpos	Character position index to end highlighting. The ending character position will be the end of the line containing this character.

insert

The insert command works almost exactly the same as the standard text insert command with one exception, it can accept an optional “-moddata” option. The moddata option allows the user to pass user-specific information to the callback procedure that handles the <<Modified>> virtual event.

In TKE, there is only one value that is used for -moddata, the value of “ignore”. Adding the “-moddata ignore” option will cause the insertion to not change the modified state of the text widget. This allows plugin code to initially insert data into the buffer without making it look like the user modified the contents of the buffer.

Call structure

```
pathname insert ?–moddata value? index chars ?taglist chars taglist …?
```

Return value

None.

inBlockComment

The inBlockComment command returns a value of true if the specified index exists within a block comment. It can also return the range of the comment in the given text widget.

Call structure

```
ctext::inBlockComment pathname index ?rangeref?
```

Return value

Returns a value of true if the specified index exists within a block comment; otherwise, returns a value of false.

Parameters

The *pathname* references the ctext widget to check. The *index* value refers to the ctext index to check. The *rangeref* parameter, if specified, will be populated with a list containing the starting and ending position of the block comment that surrounds the given index. This value will only be valid when the procedure returns a value of true.

inComment

The inComment command returns a value of true if the specified index exists within a line or block comment. It can also return the range of the comment in the given text widget.

Call structure

```
ctext::inComment pathname index ?rangeref?
```

Return value

Returns a value of true if the specified index exists within a block or line comment; otherwise, returns a value of false.

Parameters

The *pathname* references the ctext widget to check. The *index* value refers to the ctext index to check. The *rangeref* parameter, if specified, will be populated with a list containing the starting and ending position of the comment that surrounds the given index. This value will only be valid when the procedure returns a value of true.

inCommentString

The inCommentString command returns a value of true if the specified index exists within a comment or a string. It can also return the range of the comment/string in the given text widget.

Call structure

```
ctext::inCommentString pathname index ?rangeref?
```

Return value

Returns a value of true if the specified index exists within a comment or string; otherwise, returns a value of false.

Parameters

The *pathname* references the ctext widget to check. The *index* value refers to the ctext index to check. The *rangeref* parameter, if specified, will be populated with a list containing the starting and ending position of the comment/string that surrounds the given index. This value will only be valid when the procedure returns a value of true.

inDoubleQuote

The inDoubleQuote command returns a value of true if the specified index exists within a double-quoted string. It can also return the range of the string in the given text widget.

Call structure

```
ctext::inDoubleQuote pathname index ?rangeref?
```

Return value

Returns a value of true if the specified index exists within a double-quoted string; otherwise, returns a value of false.

Parameters

The *pathname* references the ctext widget to check. The *index* value refers to the ctext index to check. The *rangeref* parameter, if specified, will be populated with a list containing the starting and ending position of the string that surrounds the given index. This value will only be valid when the procedure returns a value of true.

inLineComment

The inLineComment command returns a value of true if the specified index exists within a line comment. It can also return the range of the comment in the given text widget.

Call structure

```
ctext::inLineComment pathname index ?rangeref?
```

Return value

Returns a value of true if the specified index exists within a line comment; otherwise, returns a value of false.

Parameters

The *pathname* references the ctext widget to check. The *index* value refers to the ctext index to check. The *rangeref* parameter, if specified, will be populated with a list containing the starting and ending position of the comment that surrounds the given index. This value will only be valid when the procedure returns a value of true.

inSingleQuote

The inSingleQuote command returns a value of true if the specified index exists within a line comment. It can also return the range of the comment in the given text widget.

Call structure

```
ctext::inSingleQuote pathname index ?rangeref?
```

Return value

Returns a value of true if the specified index exists within a single-quoted string; otherwise, returns a value of false.

Parameters

The *pathname* references the ctext widget to check. The *index* value refers to the ctext index to check. The *rangeref* parameter, if specified, will be populated with a list containing the starting and ending position of the quote that surrounds the given index. This value will only be valid when the procedure returns a value of true.

inString

The inString command returns a value of true if the specified index exists within a single-, double- or triple-quoted string. It can also return the range of the string in the given text widget.

Call structure

```
ctext::inString pathname index ?rangeref?
```

Return value

Returns a value of true if the specified index exists within a string; otherwise, returns a value of false.

Parameters

The *pathname* references the ctext widget to check. The *index* value refers to the ctext index to check. The *rangeref* parameter, if specified, will be populated with a list containing the starting and ending position of the string that surrounds the given index. This value will only be valid when the procedure returns a value of true.

inTripleQuote

The inTripleQuote command returns a value of true if the specified index exists within a triple-quoted string. It can also return the range of the string in the given text widget.

Call structure

```
ctext::inTripleQuote pathname index ?rangeref?
```

Return value

Returns a value of true if the specified index exists within a triple-quoted string; otherwise, returns a value of false.

Parameters

The *pathname* references the ctext widget to check. The *index* value refers to the ctext index to check. The *rangeref* parameter, if specified, will be populated with a list containing the starting and ending position of the string that surrounds the given index. This value will only be valid when the procedure returns a value of true.

isEscaped

The isEscaped command returns a value of true if the specified character has an odd number of subsequent escape (\) characters preceding it on the same line. This procedure has been optimized for speed so it is recommended to use this if this information needs to be retrieved.

Call structure

```
ctext::isEscaped pathname index
```

Return value

Returns a value of true if the character at the given index is immediately preceded by an odd number of escape (\) characters; otherwise, returns a value of false.

Parameters

The *pathname* references the ctext widget to check. The *index* value refers to the ctext index to check.

edit undoable

Returns a boolean value of true if the undo stack contains at least one set of changes. Returns a boolean value of false if the undo stack is empty.

Call structure

```
pathname edit undoable
```

Return value

Returns false if the undo stack is empty; otherwise, returns true.

Parameters

None.

edit redoable

Returns a boolean value of true if the redo stack contains at least one set of changes. Returns a boolean value of false if the redo stack is empty.

Call structure

```
pathname edit redoable
```

Return value

Returns false if the redo stack is empty; otherwise, returns true.

Parameters

None.

edit cursorhist

Returns a list of cursor indices from the undo stack such that the first index corresponds to the oldest cursor in the stack while the newest cursor position is at the end of the list.

Call structure

```
pathname edit cursorhist
```

Return value

Returns a list of indices containing the history of stored cursor positions from oldest to newest. If the undo stack is empty, an empty list will be returned.

Parameters

None.

gutter create

A gutter is a single character column drawn in the linemap area of the widget. All gutters are added to the right of the line number column (if displayed). Each row in the gutter will correspond to the associated line in the editing area. Each gutter row can display a background color and/or a symbol (i.e., unicode character). If the user moves the mouse over a gutter row an on_enter or on_leave event can be bound to handle the event. Additionally, if the user left clicks on a gutter row, an on_click event can be bound to handle the event.

The ‘gutter create’ command must be called prior to calling any other gutter commands. Each gutter is given a developer-provided name. This name is used in all of the other gutter commands to refer to which gutter to operate on.

This command adds the gutter to the gutter area and, optionally, configures its setup/behavior.

Call structure

```
pathname gutter create name ?symbol_name symbol_opt_list ...?
```

Return value

None.

Parameters

Parameter	Description
name	Developer-supplied name of the gutter to create. All other gutter commands require this name.
symbol_name	Name of a symbol that will exist in the gutter.
symbol_opt_list	List containing an even number of option/value pairs that will be associated with the symbol called symbol_name. The possible values contained in this list are represented in the Gutter Symbol Options table.

Gutter Symbol Options

The following options are used in the ‘gutter create’, ‘gutter cget’ and ‘gutter configure’ commands.

Option	Value(s)	Description
-symbol	Unicode character	Specifies the character to draw in all gutter rows associated with the given symbol name.
-bg	color	Specifies background color to use for all rows tagged with the associated symbol name.
-fg	color	Specifies foreground color to use for all rows tagged with the associated symbol name.
-onenter	command	Command to execute whenever the user mouse cursor enters a row represented by the associated symbol name. The pathname of the widget is appended to the command when executed.
-onleave	command	Command to execute whenever the user mouse cursor leaves a row represented by the associated symbol name. The pathname of the widget is appended to the command when executed.
-onclick	command	Command to execute whenever the user left clicks on a row represented by the associated symbol name. The pathname of the widget and the line number of the clicked symbol is appended to the command when executed.
-onshiftclick	command	Command to execute whenever the user holds the Shift key while left clicking on a row represented by the associated symbol name. The pathname of the widget and the line number of the clicked symbol is appended to the command when executed.
-oncontrolclick	command	Command to execute whenever the user holds the Control key while left clicking on a row represented by the associated symbol name. The pathname of the widget and the line number of the clicked symbol is appended to the command when executed.

gutter destroy

Removes a previously created gutter from the gutter and destroys all memory associated with the gutter. If the specified gutter name does not refer to a created gutter, nothing is done.

Call structure

```
pathname gutter destroy name
```

Return value

None.

Parameters

Parameter	Description
name	Name of gutter to destroy.

gutter set

Tags one or more rows in the gutter with one or more gutter symbols. If a gutter row in one of the lists already is tagged with a different symbol, that symbol is replaced with the new symbol.

Call structure

```
pathname gutter set name ?symbol_name rows ...?
```

Return value

None.

Parameters

Parameter	Description
name	Name of gutter to modify.
symbol_name	Name of symbol to associate the given list of rows with. The value of symbol_name must be created prior to the ‘gutter set’ call in either the ‘gutter create’ or ‘gutter configure’ commands.
rows	A list containing one or more integer values ranging from 1 to the maximum number of lines in the widget.

gutter delete

Removes one or more symbol names from the gutter.

Call structure

```
pathname gutter delete name symbol_name_list
```

Return value

None.

Parameters

Parameter	Description
name	Name of gutter to modify.
symbol_name_list	List of symbol names to delete from the specified gutter.

gutter get

Returns a list of information about the specified gutter, depending on the call structure. In the first call structure where only the name of the gutter is specified, the command returns a list of all symbol names stored in the gutter. If both the name of the gutter and the name of a symbol is specified, returns a list of rows that are tagged with the given symbol.

Call structure

```
pathname gutter get name ?symbol_name?
```

Return value

If *symbol_name* is not specified, returns a list of all symbol names in the first case; otherwise, returns a list of rows tagged with the given symbol in the second case.

Parameters

Parameter	Description
name	Name of gutter to query.
symbol_name	Name of gutter symbol to query.

gutter clear

Clears all rows in the specified row range of all tagged symbols.

Call structure

```
pathname gutter clear name first ?last?
```

Return value

None.

Parameters

Parameter	Description
name	Name of gutter to modify.
first	First row in range to clear.
last	Last row in range to clear (inclusive). If last is not specified, only the row specified with first is affected.

gutter cget

Retrieves the gutter option value associated with the given gutter value.

Call structure

```
pathname gutter cget name symbol_name option
```

Return value

Returns the value associated with the given gutter name / symbol name / gutter option.

Parameters

Parameter	Description
name	Name of gutter to query.
symbol_name	Name of gutter symbol name to query.
option	Name of gutter option to query the value of.

gutter configure

Either allows gutter options to be set to different values or retrieves a list of all option/value pairs associated with the given gutter/symbol name.

Call structure

```
pathname gutter configure name symbol_name ?option? ?value option value ...?
```

Return value

If the first call is made, returns a list of all symbol names and associated gutter symbol options assigned to the given gutter.

Parameters

Parameter	Description
name	Name of gutter to modify/query.
symbol_name	Name of gutter symbol to modify/query.
option	Name of gutter symbol option to modify.
value	Value to assign to the associated symbol option.

gutter names

Returns a list of names for all of the stored gutters.

Call structure

```
pathname gutter names
```

Return value

Returns a list of stored gutter names.

Parameters

None.

replace

The replace command works almost exactly the same as the standard text replace command with one exception, it can accept an optional “-moddata” option. The moddata option allows the user to pass user-specific information to the callback procedure that handles the <<Modified>> virtual event.

In TKE, there is only one value that is used for -moddata, the value of “ignore”. Adding the “-moddata ignore” option will cause the replacement to not change the modified state of the text widget. This allows plugin code to modify data in the buffer without making it look like the user modified the contents of the buffer.

Call structure

```
pathname replace ?–moddata value? index1 index2 chars ?taglist chars taglist …?
```

Return value

None.

Packages

The following is a list of of available packages to the plugin code that is already included in TKE.

Tclx

Documentation for the tclx package can be found at http://www.tcl.tk/man/tclx8.2/TclX.n.html

Tablelist

Documentation for the tablelist package can be found at http://www.nemethi.de/tablelist/index.html. TKE currently uses version 5.14.

tooltip

Documentation for the tooltip package can be found at http://docs.activestate.com/activetcl/8.5/tklib/tooltip/tooltip.html

msgcat

Documentation for the msgcat package can be found at http://www.tcl.tk/man/tcl8.5/TclCmd/msgcat.htm

tokenentry

Documentation for the tokenentry package can be found at http://ptwidgets.sourceforge.net/page3/files/tokenentry.html

tabbar

Documentation for the tabbar package can be found at http://ptwidgets.sourceforge.net/page3/files/tabbar.html

fontchooser

The fontchooser packet provides a UI for selecting a font. If the window is canceled, an empty string is returned; otherwise, the font value of the selected font is returned. The returned font will be in the option/value Tcl font form. This package provides a single user command:

```
fontchooser ?option value …?
```

The available options are specified as follows:

Option	Description
-parent *window*	Makes window the logical parent of the fontchooser dialog. The file dialog is displayed on top of its parent window.
-initialfont *font*	Specifies the initial font to display in the window. The value of font can be any legal Tcl font description.
-title *titleString*	Specifies a string to display as the title of the dialog box. If this option is not specified, no title is displayed.