TKE User's Guide

Version: 1.4

Written by: Trevor Williams

Table of Contents

Chapter 1: Introduction

TKE is a source code editing environment built using Tcl/Tk which provides a clean user interface yet rich set of editing features and tools.

Feature Set

The following is a high-level list of features built-in to the tool.

Editor features

Feature	Special Notes

Syntax highlighting	• Over 20 languages currently supported, including:
	  • AppleScript
	  • Bash
	  • C/C++
	  • CSS
	  • CoffeeScript
	  • HTML
	  • Haml
	  • Haskell
	  • HelpSystem
	  • JSON
	  • JavaScript
	  • Makefile
	  • Markdown (experimental)
	  • PHP
	  • Perl
	  • Python
	  • RSS
	  • Ruby
	  • ShellScript
	  • SystemVerilog
	  • Tcl
	  • XML
	• Syntax description files can be easily added
	• Preference control can select syntax types to highlight/not highlight
	• Built-in and customizable color themes

Multi-cursor support	• Cursor alignment
Internationalization support	• 17 languages currently supported
Customizable menu bindings	
Clipboard history	
Unlimited undo/redo	
Language-specific snippet support	• Support for tab stops, variable substitution, and special value substitutions.
Auto and smart indentation features	• Selected code can have indentation policies applied • Pasted code can have indentation policies applied
Code line marking support	
Line number display	

Feature	Special Notes
Customizable tab stops	
Many text transformation tools	
Built-in Vim support	
Expanded Vim functionality	• Vim commands for setting/clearing multiple cursors • Auto-numbering functionality
Regular expression in-file search (and optional replace)	
Regular expression multi-directory file search	
Symbol search function	• Jump to a named procedure or function call
Auto refresh	• Files modified outside of editor will be automatically updated (unless file is in a modified state)
File locking support	• File can be set to be read-only within the editor (regardless of actual file permissions)
Command launcher	
File system sidebar	• Contains functionality for creating, renaming, deleting files/directories
Multiple files can be opened at once	• Sidebar and tabs used to switch between opened files
Dual editor panel support	• Useful for viewing two files side-by-side
Split view support	• Create two independent views into the same file.
Maximum column width display	
Automatic session save	
Support for NFS mounted file systems	
Plugin support	• Full plugin development documentation available.
Support for light and dark UI/window themes	
Favorite file/directory support	

Feature	Special Notes
Automatic matching character insertion	• Curly bracket, square bracket, angled bracket, parenthesis, double and single string character matches are inserted as you type. • Preference item to enable/disable any of the above character types. • Each language syntax file specifies which characters are valid for auto-insertion.
In-app update mechanism	• Preference option to follow stable or development release track.

Chapter 2: Installation

Dependencies

The installation of TKE has a few dependencies that will need to be preinstalled before the application can begin to work. The dependencies are listed in the table below along with the URL path to find the source packages. The installation of these packages is outside the scope of this document. Please refer to each packages installation notes for this information.

Package	Download URL
Tcl (version 8.5.x — not tested with 8.6.x versions)	http://sourceforge.net/projects/tcl/files/Tcl/
Tk (version 8.5.x — not tested with 8.6.x versions)	http://sourceforge.net/projects/tcl/files/Tcl/
Extended Tcl (Library should be installed in one of the standard Tcl paths)	http://sourceforge.net/projects/tclx/files/TclX/

All other Tcl/Tk packages required by TKE have been bundled in the TKE package.

Installing for Linux

Prior to downloading/installing the TKE package, you will need to make sure that you have all of the required packages installed on your system. Because various Linux distributions have different package managers, I will leave the exact details of how to accomplish this up to you. However, if you have an Ubuntu-based distribution, you can get the needed packages by performing the following command:

```
sudo apt-get install tcl8.5 tk8.5 tclx8.4 tcllib tklib
```

The TKE installation package is downloaded in a gzipped tarball. You can get the latest version of this tarball from the following URL:

http://sourceforge.net/projects/tke/files/

Select a tarball (i.e., *.tar.gz file) to download within this page and save the resulting tarball into a temporary directory. After the download has completed, unzip and untar the file using the given command:

```
gzip -dc tarball_filename | tar xvf -
```

After the tke directory has been untarballed, you can delete the original tarball using the following command:

```
rm -rf tarball_filename
```

After all of the files have been uncompressed, change the working directory to the resulting "tke-*X.X*" directory using the following command:

```
cd tke-X.X
```

Once inside the TKE source directory, run the installation script found in that directory using the following command:

```
tclsh8.5 install.tcl
```

At the beginning of the installation process, the install script will check to make sure that you have both Tcl and Tk 8.5 installed along with a usable version of TclX. If all checks are good, the installation will continue; otherwise, it will provide an error message indicating the offending check. After the checks occur, you will be asked to provide a root directory to install both the TKE library directories/files and the TKE binary file. This can be any directory in your filesystem; however, popular directories are:

- /usr/local
- /usr

After specifying a file system directory, TKE will indicate the names of the directory and binary file that it will install. If everything looks okay, answer "Y" or "y" (or just hit the RETURN key); otherwise, hit the "N" or "n" keys to enter a different directory. Once you enter a directory, the installation script will check to see if a previous version of TKE has been installed at that directory location. If one is found, it will ask if you would like to replace the old version with the new version. Hit the "Y" or "y" key (or just hit the RETURN key) to confirm the replacement. To cancel the installation and select a new directory, hit the "N" or "n" key. If you have specified that the given directory should be replaced (or no replacement was necessary), the script will continue with the full installation. At any time you can quit the installation script by entering the CONTROL-c key combination.

Installing for Mac OSX

If you only plan on running tke from a terminal and are satisfied with running the application through the X11 server that runs on Mac, you can follow the same installation steps that is used for Linux-based systems. However, if you would like to install TKE like a native Mac OSX

application (i.e., application available in the Applications folder, TKE icon displayed in the dock, etc.), follow these installation steps.

After downloading the TKE disk image into the Downloads folder, double-click the disk image file and then drag and drop the TKE application icon in the resulting window to the Applications directory.

Important note: As of TKE version 1.1, the wish shell that is used is based on Cocoa and, as such, for Mac OS X versions 10.7 (Lion) and later have a feature that stops certain keys from being automatically repeated when its key is held down. This will make Vim-mode on these systems from working as expected. To disable this on your system, enter the following command within the Terminal application prior to starting TKE:

```
defaults write -g ApplePressAndHoldEnabled -bool false
```

Installing for Windows

The installation process for Windows operating systems is not defined at this time.

Chapter 3: Starting the Application

After TKE has been installed on your system, there are a variety of ways to start the application, depending on your usage.

The Command-Line

For Unix-based systems that support a terminal, you can invoke TKE using the command-line. To make tke easier to use, it is recommended that you add the TKE installation's bin directory into your environment path variable (see your shell's documentation for how to do this as this will be different for different OS types as well as shells).

Next, if you want TKE to always use just one window for editing all files, make sure that your xhost is setup correctly. If you get a new TKE window every time you open a file in the terminal, it is likely that you have an xhost issue.

Assuming that you have added the TKE installation bin directory to your path, invoking TKE is as simple as typing the following at the shell prompt:

```
tke
```

If this is the first time that the application has been started, this will create a single TKE window with no tabs opened and an empty sidebar. If the application is not currently running, this will start the application and load the last TKE session information into the application, including the following information:

- Window dimensions and location
- Previously opened files when the application was exited
- Sidebar entries
- Current tab of previous session will be the current tab of this session

If TKE is already running, this command will simply bring the application to the foreground of the desktop.

This, however, is not the only way of starting the application from the command-line, you can also specify any number of directories and/or files as arguments to TKE. Any directories specified will be added to the sidebar while any specified files will be opened in new tabs in the editor and their respective directories will be added to the sidebar (if they don't already exist).

In addition to files and directories, the following options are also available on the command-line invocation.

Command-Line Options

Option	Description
-h	Displays command-line usage information to standard output and exits immediately.
-v	Displays tool version information to standard output and exits immediately.
-nosb	Starts the UI without the sidebar being displayed.
-e	Exits the application when the last tab is closed (overrides preference setting)
-m	Creates a minimal editing environment (overrides preference settings)
-n	Opens a new window without attempting to merge with an existing window.

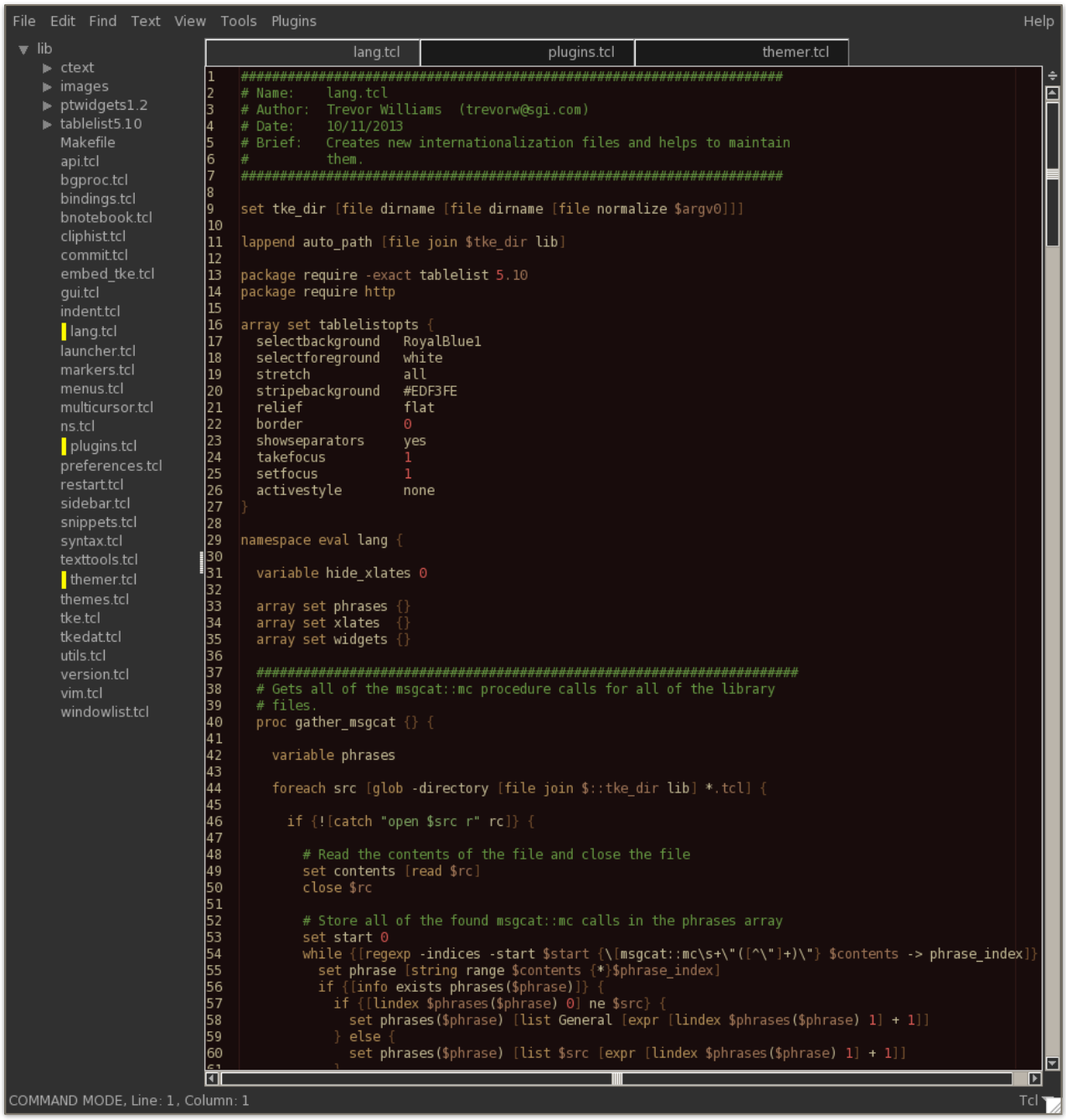
Mac OSX Desktop

On Mac OSX, the application installation will place the TKE application in the Applications directory. To launch the application, simply open the Application directory in the Finder and double-click on the application. This will launch TKE using the same session settings that TKE had when last launched.

You can also start the application using any other method available on your Mac OSX system, including Launchpad, Spotlight, third-party application launchers, etc.

Chapter 4: User Interface

By default, the editing environment consists of two panels: a file/directory sidebar and the tab-controlled editing buffer itself. As much as it possible, busy and redundant UI elements are removed from the screen when they are not in use. Most of the UI is only displayed as needed when the user calls up its functionality.



Title Bar

Within the title bar at the top of the window is the basename of the file currently being edited and the name of the current working directory. All commands that deal with the file system will be relative to this working directory.

Menu Bar

Below the title bar is the menu bar (on Windows and most Linux distributions). This contains a list of many of the available features within the tool. Any functionality that is contained within the listed menus can be assigned a keyboard shortcut, configurable via the menu binding file (see the "Menu Binding" Chapter for more details on the structure of this file). From left to right, the main menus are as follows:

- File
- Edit
- Find
- Text
- View
- Tools
- Plugins
- Help

File Menu

The File Menu contains commands that are related to either the currently selected file (i.e., the file in view within the editor which has the keyboard focus) or all files. The following table describes the listed menu items and their associated functionality

Menu Item	Description
New	Creates a new, unnamed file in a new tab.
Open File…	Displays an open file dialog, allowing the user to select one or more files to open. Each file will be opened in a separate tab in the editor. Any directories containing these files that are not in the sidebar will be added to the sidebar.
Open Directory…	Displays an open directory dialog, allowing the user to select one or more directories to add to the sidebar.
Open Recent	Displays a list of files that have been recently opened. Click on a file to open it in a separate tab in the editor.
Open Favorite	Displays the list of favorited files/directories for quick opening in either the editor (file) or sidebar (directory).
Save	Saves the contents of the current file to its original name. If an original name does not exist for the content, a "Save As" dialog will be displayed allowing the user to specify a file name.
Save As…	Displays a save file dialog window, allowing the user to save the current file contents to the given filename. The original filename of the content will be changed to this new name.
Save All	Saves all files opened in the editor to their original file names. Any files which do not have original names, will have a save file dialog window shown, allowing the user to specify the name.
Lock/Unlock	The "Lock" option will change the state of the editor to not allow text modifications to the window (content is effectively "Read Only"). A small lock icon will be displayed in the associated tab to indicate that the file content is currently "locked". The "Unlock" option will change the state of the editor back to the modifiable state.
Favorite/Unfavorite	Marks the current file as a favorite (with the "Favorite" command) or removes the file as a favorite (with the "Unfavorite" command). Favorited files can be opened quickly with the "Open Favorite" menu list or the command launcher. Additionally, favorited files/directories can be used in the "Find in File" feature.

Menu Item	Description
Close	Closes the current tab. If the text content is in the modified state (as indicated by the "*" character in the tab), a prompt will be displayed asking the user if the content should be saved prior to closing.
Close All	Closes all tabs in the editor. If text content in a tab has been modified, a prompt will be displayed asking the user if the content should be saved prior to closing.
Quit	Exits the application. Any modified files in the editor will prompt the user if the content should be saved prior to exiting the application.

Edit Menu

The Edit menu contains menu items that affect the contents within the current file. The following table describes the items available within this menu.

Menu Item	Description
Undo	Undoes the last change made to the file content. Each file can have an unlimited number of items that can be undone. Saving a file clears the undo stack for that file.
Redo	Re-applies the last undone change made to the file content. Saving a file clears the redo stack for that file.
Cut	Deletes the selected text, copying the deleted content to the clipboard.
Copy	Copies the selected text to the clipboard.
Paste	Pastes the content in the clipboard, inserting the text before the insertion cursor. The content is copied "as is".
Paste and Format	Pastes the content in the clipboard, inserting the text before the insertion cursor. The content is indented to fit into the current insertion point.
Select All	Selects all of the text in the current editor.
Enable/Disable Auto-Indent	Turns the auto-indentation mode for the current editor on or off. This setting overrides the preference file auto-indentation mode setting.
Insert / From Clipboard	Displays the command launcher in clipboard mode to allow the user to view and select one of the clipboard history elements to insert into the current editor.
Insert / Snippet	Displays the command launcher in snippet mode to allow the user to view and select one of the language-specific snippets to insert into the current editor.
Format Text	Modifies either the selected text or the entire file content (depending on whether the "Selected" or "All" menu item is selected) to match the indentation in the current context.
Preferences / View base	Adds the base preferences file to the editor in "readonly" mode.
Preferences / Edit user	Adds the user preferences file to the editor to allow the user to override global preference values.
Preferences / Set user to base	Copies the base preferences file contents to the user's preference file (note: this action destroys all user preference settings that exist before this action takes place).

Menu Item	Description
Menu Bindings / View global	Adds the global menu bindings file to the editor in "readonly" mode.
Menu Bindings / Edit user	Adds the user menu bindings file to the editor to allow the user to override global menu bindings values.
Menu Bindings / Set user to global	Copies the global menu bindings file contents to the user's menu bindings file (note: this action destroys all user menu binding settings that exist before this action takes place).
Snippets / Edit current	Adds the user's snippet file into the editor for the current language.
Snippets / Reload current	Reloads the contents of the snippets for the current language. Useful if the snippet file contents are not usable within the editor.

Find Menu

The Find menu contains items for searching and, optionally, replacing text in the current file. It also contains items that can add search text to the current selection and items for finding text in a group of files (regardless if they are currently opened in the editor or not). The following table contains the items found in this menu along with the description of its functionality.

Menu Item	Description
Find	Searches the current file for a given regular expression. The displayed search bar also contains a button for specifying whether a case sensitive search should be performed or not. All matches in the current file will be highlighted and the first match after the current cursor will be viewable and the cursor will be moved to the beginning of the match.
Find and Replace	Searches the current file for a given regular expression and replaces it with an associated string. The displayed search and replace bar also contains two buttons: one for specifying case sensitivity of the match and one for replacing the first match or all matches.
Select next occurrence	Selects the next matched occurrence..
Select previous occurrence	Selects the previous matched occurrence.
Select all occurrences	Selects all matched occurrences.
Append next occurrence	Adds the next matched occurrence to the selection.
Find marker	Jumps the cursor and file view to show the selected marker. The cursor will be placed at the beginning of the marked line.
Find matching pair	Jumps the cursor and file view to show the parenthesis or bracket that matches the parenthesis or bracket under the current cursor. The cursor will be placed on the matched pair.
Find in files	Displays a user input interface that allows the user to specify a regular expression to find within files in one or more files/directories. The directory input field can contain one or more directories (each directory is handled as a "token" in the input field). A few directories are readily available by typing a portion of their name. They are as follows: • Basename of any file/directory in the sidebar • Basename of any favorited file/directory • "Opened Files" - searches any files opened in the editor • "Opened Directories" - searches all opened directories in the sidebar • "Current Directory" - searches the current working directory The find interface also contains a button specifying the case sensitivity to use in the search.

Text Menu

The Text menu contains various text manipulation tools that can be used in the current file. The following table describes these menu items.

Menu Item	Description
Comment	Places a line comment in front of any selected text in the current file.
Uncomment	Removes any line comments found in the selected text in the current file.
Indent	Indents the selected text by one level of indentation.
Unindent	Unindents the selected text by one level of indentation.
Align cursors	When multicursors are set in the current file, this command will adjust each line such that all cursors will be aligned to the same column. The cursors will be aligned to the highest column in the multicursor set.
Insert enumeration	When one or more multicursors are set, allows the user to insert an ascending numerical values as specified in the subsequent "Starting number:" entry field. The content of the starting number determines what the base of the number will be. The following are valid starting number representations: *prefix*[0-9]+ • Inserts prefix followed by a decimal value. *prefix*d[0-9]+ • Inserts prefix followed by a decimal value preceded by "d" *prefix*b[0-1]+ • Inserts prefix followed by a binary value preceded by "b" *prefix*o[0-7]+ • Inserts prefix followed by an octal value preceded by "o" *prefix*[xh][0-9a-fA-F]+ • Inserts prefix followed by a hexadecimal value preceded by either "x" or "h". Note: If a value is not specified, a value of zero is assumed.

View Menu

The View menu allows the user to change the interface as desired. The following table lists the available menu items.

Menu Item	Description
Show/Hide Sidebar	Shows or hides the sidebar panel.
Show/Hide Console	For operating systems that allow a Tcl/Tk console to be viewed, shows/hides this console window from view. This menu item is only displayed if a console is available. The console is mostly useful for debugging purposes only.
Show/Hide Tab Bar	Shows or hides the tab bar.
Show/Hide Status Bar	Shows or hides the status bar at the bottom of the window.
Split view	When selected, creates a second view into the current file. Each view can be independently manipulated; however, any text modifications made in either window will be available in the other view. Deselecting this menu option will return the file to only showing a single view of the file in the editor.
Move to other pane	Moves the current file to the other text pane. If only one text pane is currently viewable, a second pane will be displayed to the right of the current pane and the file will be moved to that pane. If a pane only contains the file that is being moved, that pane will be removed from view. This allows two files to be viewed "side by side".
Tabs / Goto Next Tab	Changes the current file to be the file in the next tab in the current pane to the right of the current tab.
Tabs / Goto Previous Tab	Changes the current file to be the file in the next tab in the current pane to the left of the current tab.
Tabs / Goto Last Tab	Changes the current file to be the file in the last viewed tab in the current pane.
Tabs / Goto Other Pane	Changes the current keyboard focus to the current tab in the other pane. This menu item is only available if both panes in viewable.
Tabs / Sort Tabs	Alphabetically sorts the tabs in the current pane.
Set Syntax	Changes the syntax highlighting and language-specific functionality to the specified language. Be default, the language is determined by file extension. This menu allows the user to override the default behavior. To permanently add an extension to a language syntax handler, you will need to modify the associated syntax file. See the "Syntax Handling" chapter for more information about the structure of this file.

Menu Item	Description
Set Theme	Changes the current syntax coloring scheme to one of the available themes. Setting the theme to this value will only be in effect while the application is running. If the application is quit and restarted, the default theme as specified in the preferences will be used.

Tools Menu

The Tools menu contains various miscellaneous functions that are available within the editor. The following table describes the items found in this menu.

Menu Item	Description
Launcher	Displays the command launcher interface, allowing the user to quickly access commands and other useful functionality as described in the "Command Launcher" chapter.
Theme Creator / Create new…	Creates a new syntax color theme using the theme editor/creation utility. More information about this window is available in the "Theme Creator" chapter.
Theme Creator / Edit…	Allows the user to modify one or more colors in the an existing theme using the theme editor/creation utility. More information about this window is available in the "Theme Creator" chapter.
Theme Creator / Import TextMate theme…	Allows the user to create a new syntax color theme by importing a TextMate theme file and displaying this new theme in the theme editor/ creation utility. More information about this window is available in the "Theme Creator" chapter.
Vim Mode	When selected, changes the editing environment to use Vim-style interaction. When deselected, changes the editing environment back to "normal" editing mode.
Development Tools	
Start Profiling	Starts the UI profiling facility. This allows procedural performance evaluation.
Stop Profiling	Stops the current profiling run. After profiling information has been gathered, a profile report can be viewed within the editor.
Show Last Profiling Report	Displays the results of the last profiling run within the editor.
Restart tke	Allows the editor to be quit, restarted and returned to the current editing state. This is useful when TKE source code is modified and needs to be reloaded.

Plugins Menu

The Plugins menu contains items that allow third-party plugins to be installed, uninstalled and reloaded. Additionally, if TKE is run in developer mode, provides a facility for creating a new plugin quickly. The following table describes the items in this menu.

Menu Item	Description
Install…	Allows new third-party plugins to be installed. See the "Plugins" chapter for more information.
Uninstall…	Allows third-party plugins to be uninstalled. See the "Plugins" chapter for more information.
Reload	Reloads all installed plugins. This is primarily useful when developing plugins. This menu option allows plugins to be quickly reloaded without requiring the application to be quit and relaunched.
Development Tools	
Create…	Creates the template for a new plugin and displays the file in the editor. See the "Plugin Development" chapter for more details about how to create third-party plugins.

Help Menu

The Help menu provides instructional facilities to help you learn how to use this application and to describe the application version information. The following table describes the available items in this menu.

Menu Item	Description
User Guide	Displays this User's Guide in the default PDF viewer application in your environment.
Check for Update	Performs an in-app update. If an update is available, a window detailing the update information will be displayed. If the application is current with the latest available release, a window will be displayed indicating that this is the case. Upon successful completion, the application will be restarted into the new version.
About TKE	Displays a window detailing the current application version and developer information.

Sidebar

The sidebar is located on the left side of the window. It contains a tree-like view of one or more root directories (any directory in a file system can be a TKE root directory), subdirectories and files. By default, whenever a file is opened within TKE, the file's directory is automatically added to the sidebar. Additionally, the user can open a directory via the "File" menu which is added to the sidebar. This sidebar view allows the user to quickly open other files that are within the same directory without having to navigate through an open dialog box.

In addition to being able to quickly open files from the sidebar, several other functions are provided for each type of directory and file. The following subsections identify the different types and their associated functionalities. To access the menu of functionality for a given type, simply right-click on an item in the sidebar. This will display a contextual menu listing the available commands.

To hide or show a level of directory hierarchy, left-click on the disclosure triangle next to the directory to show/hide.

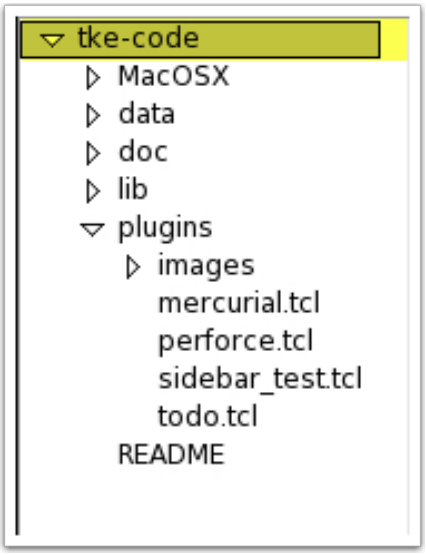
Files within the sidebar can be automatically filtered out of the sidebar via the "Sidebar/ IgnoreFilePatterns" preference item. Any files that match any patterns in this list will not be displayed in the sidebar. This is useful for de-cluttering the sidebar with files that cannot be edited within TKE (i.e. object files, image files, etc.)

Files and directories are added in alphabetical order.

Root Directory

A root directory in the sidebar is any directory that doesn't have a parent directory immediately shown in the sidebar. You may have more than root directory listed in the sidebar. To view the full pathname of a root directory, hover the cursor over the directory name until the tooltip appears.

The following image is a depiction of the sidebar with the root directory highlighted.



The following table lists the available contextual menu functions available for root directories.

Menu Item	Description
New File	Adds a new file to the root directory. If this menu item is selected, an entry field at the bottom of the window displayed, allowing the user to specify a filename for the new file. Entering a name and hitting the RETURN key will create the new file in the directory and open the file in the editor.
New Directory	Adds a new directory to the root directory. If this menu item is selected, an entry field at the bottom of the window is displayed, allowing the user to specify a name for the directory. Entering a name and hitting the RETURN key will create the new directory.
Open Directory Files	Opens all shown files that are within the directory.
Close Directory Files	All open files in the editor that exist within the root directory and below it will be closed. Any files which require a save will prompt the user to save or discard the file modifications.
Rename	Renames the root directory in the file system. If this item is selected, the name will be editable within the sidebar. Changing the name and hitting the RETURN key will rename the directory. Hitting the ESCAPE key will cancel the rename operation.
Delete	Deletes the root directory from the filesystem and removes the directory from the sidebar. If this item is selected, an affirmation prompt will be displayed to confirm or cancel the deletion.
Favorite/Unfavorite	Marks the selected directory to be a favorite (if the Favorite command is selected) or removes it from the favorites list (if the Unfavorite command is selected). Favorited directories can be quickly added to the sidebar via the File / Open Favorite menu or the command launcher.
Remove from Sidebar	Removes the root directory from the sidebar (no modification to the file system will take place). If this item is selected, the entire root directory is removed from the sidebar.
Add Parent Directory	Adds the parent directory in the filesystem of the root directory. The current root directory will no longer be a root directory (replaced by the parent directory) but will become a standard directory underneath the parent.

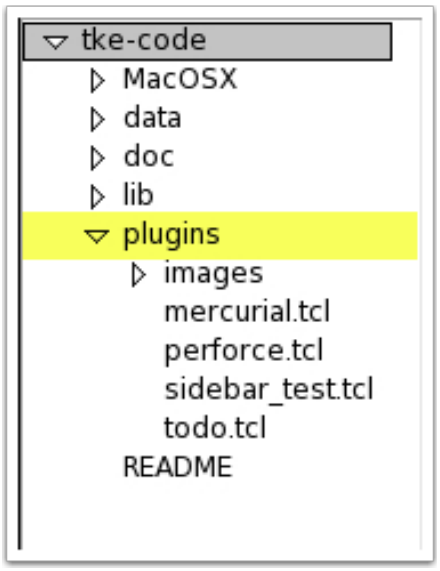
Menu Item	Description
Make Current Working Directory	Changes the current working directory to the root directory. Selecting this item will make all file operations within the editor relative to the selected directory. Additionally, the working directory information in the title bar will be updated to match this directory.
Refresh Directory Files	Updates the sidebar contents for the root directory.

In addition to these functions, plugins can also add functionality beneath these items in the menu. See the Plugins and Plugin Development chapters for more information.

Directory

A non-root directory is any directory in the sidebar which has a parent directory associated with it (i.e., any directory in the sidebar that is not a root directory).

The following image depicts the sidebar with a non-root directory highlighted.



The following table lists the available contextual menu functions available for non-root directories.

Menu Item	Description
New File	Adds a new file to the directory in both the sidebar and the file system. If this menu item is selected, an entry field at the bottom of the window displayed, allowing the user to specify a filename for the new file. Entering a name and hitting the RETURN key will create the new file in the directory and open the file in the editor.
New Directory	Adds a new directory under the selected directory in both the sidebar and the file system. If this menu item is selected, an entry field at the bottom of the window is displayed, allowing the user to specify a name for the directory. Entering a name and hitting the RETURN key will create the new directory.
Open Directory Files	Opens all shown files that are within the directory.
Close Directory Files	All open files in the editor that exist within the directory and below it will be closed. Any files which require a save will prompt the user to save or discard the file modifications.
Rename	Renames the directory in the file system. If this item is selected, the name will be editable within the sidebar. Changing the name and hitting the RETURN key will rename the directory. Hitting the ESCAPE key will cancel the rename operation.
Delete	Deletes the directory from the filesystem and removes the directory from the sidebar. If this item is selected, an affirmation prompt will be displayed to confirm or cancel the deletion.
Favorite/Unfavorite	Marks the selected directory to be a favorite (if the Favorite command is selected) or removes it from the favorites list (if the Unfavorite command is selected). Favorited directories can be quickly added to the sidebar via the File / Open Favorite menu or the command launcher.
Remove from Sidebar	Removes the directory from the sidebar (no modification to the file system will take place). If this item is selected, the entire directory is removed from the sidebar.
Remove Parent from Sidebar	Removes all parent directories of the selected directory from the the sidebar and makes the selected directory a root directory in the sidebar.

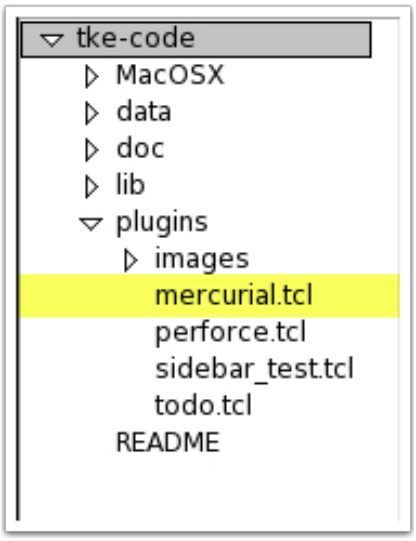
Menu Item	Description
Make Current Working Directory	Changes the current working directory to the selected directory. Selecting this item will make all file operations within the editor relative to the selected directory. Additionally, the working directory information in the title bar will be updated to match this directory.
Refresh Directory Files	Updates the sidebar contents for the selected directory.

After these functions will be listed any directory popup menu items that are added via plugins. See the Plugins and Plugin Development chapters for how to create these plugin types.

Files

Files have different functions available to them than directories. Double-clicking any file will open the file in the editor.

The following image depicts the sidebar with a file highlighted.



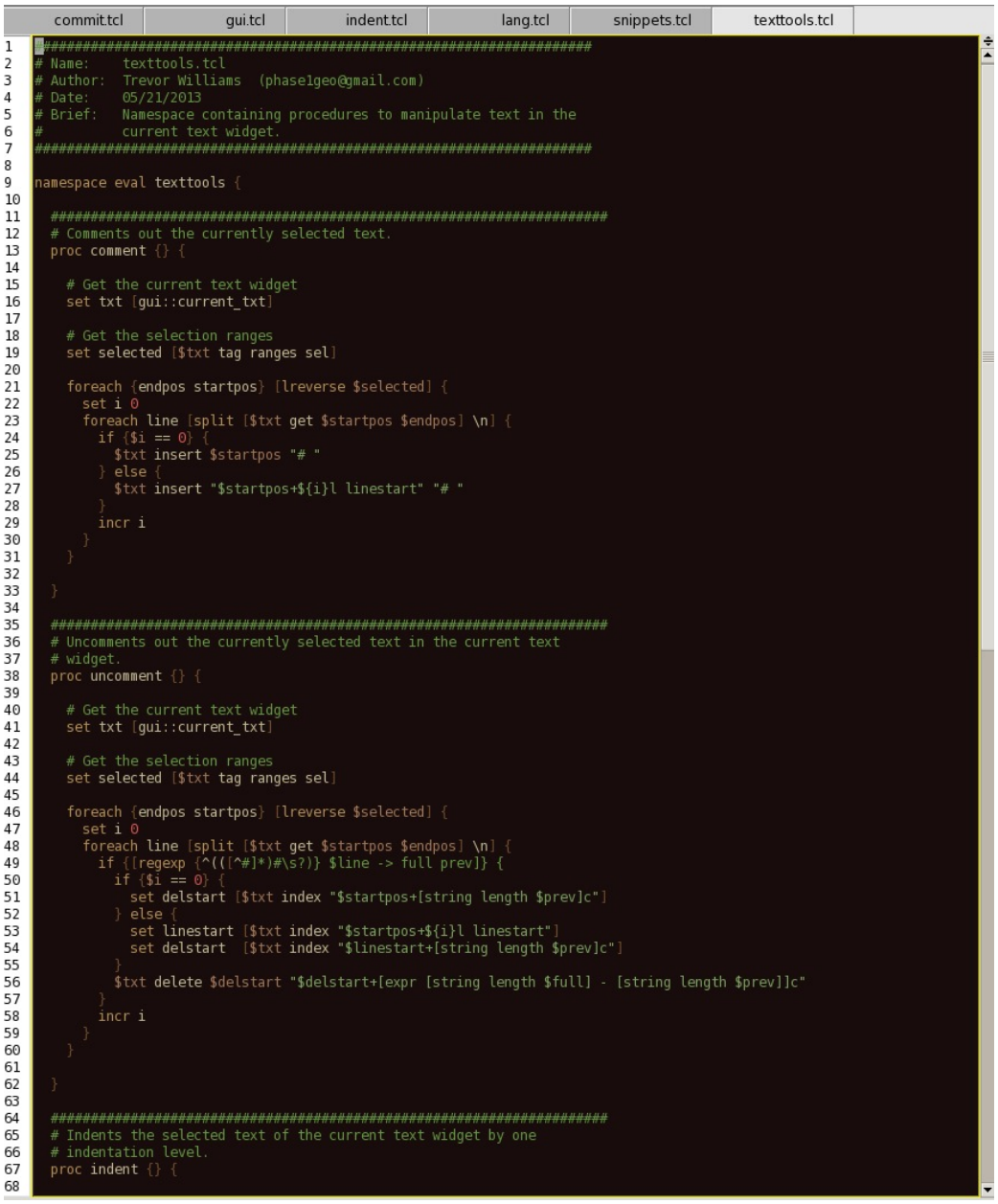
The following table lists the available functions for files.

Menu Item	Description
Open	Opens the selected file in the editor. An opened file will have a color applied to its background to make it easy to identify opened files in the sidebar.
Close	Closes the selected file in the editor. If the file has been modified, a prompt will be displayed asking if the changes should be saved or not.
Rename	Renames the file in the file system. If this item is selected, the name will be editable within the sidebar. Changing the name and hitting the RETURN key will rename the file. Hitting the ESCAPE key will cancel the rename operation.
Duplicate	Will create a duplicate file of the selected file in the same directory. The file will be named with a unique name and will be editable. Enter a new name for the file and hit the RETURN key to change the name. Hit the ESCAPE key to undo the file rename operation.
Delete	Deletes the file from the filesystem and removes the file from the sidebar. If this item is selected, an affirmation prompt will be displayed to confirm or cancel the deletion.
Favorite/Unfavorite	Marks the selected file to be a favorite (if the Favorite command is selected) or removes it from the favorites list (if the Unfavorite command is selected). Favorited files can be quickly opened in the editor via the File / Open Favorite menu or the command launcher.

After these functions will be listed any file popup menu items that are added via plugins. See the Plugins and Plugin Development chapters for how to create these plugin types.

Editing Pane

The editing pane comprises the majority of the application window, displaying all files that are open for reading/writing. The following image shows a representation of this pane.




```
1  ##################################################################
2  # Name:    texttools.tcl
3  # Author:  Trevor Williams  (phase1geo@gmail.com)
4  # Date:    05/21/2013
5  # Brief:   Namespace containing procedures to manipulate text in the
6  #          current text widget.
7  ##################################################################
8
9  namespace eval texttools {
10
11   ##################################################################
12   # Comments out the currently selected text.
13   proc comment {} {
14
15     # Get the current text widget
16     set txt [gui::current_txt]
17
18     # Get the selection ranges
19     set selected [$txt tag ranges sel]
20
21     foreach {endpos startpos} [lreverse $selected] {
22       set i 0
23       foreach line [split [$txt get $startpos $endpos] \n] {
24         if {$i == 0} {
25           $txt insert $startpos "# "
26         } else {
27           $txt insert "$startpos+${i}l linestart" "# "
28         }
29         incr i
30       }
31     }
32
33   }
34
35   ##################################################################
36   # Uncomments out the currently selected text in the current text
37   # widget.
38   proc uncomment {} {
39
40     # Get the current text widget
41     set txt [gui::current_txt]
42
43     # Get the selection ranges
44     set selected [$txt tag ranges sel]
45
46     foreach {endpos startpos} [lreverse $selected] {
47       set i 0
48       foreach line [split [$txt get $startpos $endpos] \n] {
49         if {[regexp {^(([^#]*)#\s?)} $line -> full prev]} {
50           if {$i == 0} {
51             set delstart [$txt index "$startpos+[string length $prev]c"]
52           } else {
53             set linestart [$txt index "$startpos+${i}l linestart"]
54             set delstart  [$txt index "$linestart+[string length $prev]c"]
55           }
56           $txt delete $delstart "$delstart+[expr [string length $full] - [string length $prev]]c"
57         }
58         incr i
59       }
60     }
61
62   }
63
64   ##################################################################
65   # Indents the selected text of the current text widget by one
66   # indentation level.
67   proc indent {} {
68
```

The pane is made up of three main areas:

1. Tab bar
2. Line number bar
3. Text window

Tab Bar

The Tab Bar sits at the top of the editing pane, allowing for more than one file to be edited at a time and quick switching between text windows. To switch to a different tab, simply left-click on any tab in the tab bar.

By default, tabs are added to the tab bar in the order they were opened; however, the user can change this order by clicking on a given tab and dragging the tab to a new position in the tab bar. Additionally, you can sort the tabs in alphabetical order by selecting the "View / Tabs / Sort Tabs" menu item. If you would always like tabs to be sorted in alphabetical order, you can change this behavior in the user preference file (see the Preferences chapter for information on how to edit this file).

When more files are opened than the tab bar has space to hold, the tab bar will adjust itself to show as many tabs as is comfortable to display and then display some tab shift buttons on each side of the tab bar (indicated with left and right arrows) in addition to a quick switch drop down menu button (indicated by a down arrow). To view tabs off-screen to the right, click on the right button. To view tabs off-screen to the left, click on the left button. To see a list of all opened tabs, click on the down button. The resulting list will show an in-order list of tabs in the resulting menu with separators strategically placed, to quickly find which tabs are off-screen on each side of the currently displayed tab bar.

In addition to switching between different opened text windows, the tab bar also provides a number of other functions and visual indicators. When a file is locked, a locked icon will be displayed on the left side of the associated tab. When a file has been modified, an asterisk will be displayed to the left of the file name, indicating that a save is needed to commit the changes to the file. The base name of the file is displayed in the tab along with file extension; however, if you would like to see the full pathname of the file, simply hover the mouse over any portion of the tab and a tooltip is displayed with this information. To close the tab, move the mouse cursor over the tab and a close icon is displayed on the right side of the tab, clicking on this button will close the associated tab (if the file is closed and the text has been modified, a prompt will be displayed indicating that the file has not been saved, allowing the user to save the file).

In addition to these functions, you can also access a drop down menu of functionality by right-clicking on a tab. The following table summarizes this functionality.

Menu Item	Description
Close Tab	Closes the current tab. This is the exact same behavior as clicking on the close button within the tab.
Close Other Tabs	Closes all of the other tabs in the tab bar, leaving this tab as the only opened tab in the editing pane.
Close All Tabs	Closes all tabs in the tab bar.
Locked	Indicator of the locked status of the file and allows the user to toggle the locked status of the file. If a checkmark is displayed next to this item, the file is locked and cannot be modified within TKE. If no checkmark exists, the file is modifiable.
Favorited	Indicator of the favorite status of the file and allows the user to toggle the favorite status of the file. Favorited files can be quickly opened via the File / Open Favorite menu or the command launcher.
Show in Sidebar	Causes the current file to be displayed in the sidebar (even if the directory is not disclosed).
Move to Other Pane	Moves the tab and associated text window to the other editing pane, allowing for side-by-side text editing. If the other editing pane does not exist, it will be created. If the current tab is the only tab in the current pane and it is moved, the current pane will disappear, leaving only a single pane displayed in the editing panel. This option will not be available if there is only one tab opened in the editing panel.

Line Number Bar

The line number bar is displayed on the left side of the editing panel. Each number is associated with a corresponding line in the text window.

In addition to showing line numbers, the line number bar also provides some useful selection functionality. If the left-button is clicked on a line number, the entire corresponding line is selected in the text window. If the left-button is held down while the mouse cursor is moved, the clicked line an all lines between that line and the current mouse cursor will be selected. If the SHIFT button is held when the left mouse button is clicked, all lines between the last left-clicked line and the current line are selected.

Finally, the line number bar can be used to create markers. Markers are basically jump points in the text window. Any line can be marked by right-clicking on a line number. When this occurs, an entry field at the bottom of the window is displayed, allowing the user to provide a name for the marker. Giving a name to a marker is optional, but useful. To give the marker a name, enter

a string and hit the RETURN key. To create an unnamed marker, simply hit the RETURN key in the name entry field. To cancel the creation of the marker, hit the ESCAPE key in the name entry field. After a marker has been created, the corresponding line number will be given an orange highlight in the line number bar.

To clear an existing marker, simply right-click on a line that has already been given a marker. The line highlight will be cleared to indicate that the marker no longer exists.

Text Window

The text window provides the main source of editing functionality. TKE supports two modes of editing functionality: Vim mode and standard mode. See the Vim chapter to see what functions are available when editing in this mode, the rest of this section will only mention functions available when the editor is not in Vim command mode (i.e., either Vim insert mode or standard mode).

Visually the text window contains three basic UI elements: the text editor pane, the scrollbars and the split pane button (located just above the vertical scrollbar).

The text editor pane allows text to be read and modified. The following table specifies the different key and mouse bindings on the editor that the user can take advantage of.

Key/Mouse Binding	Function
Left-mouse click	Sets insertion cursor just before the character underneath the mouse cursor. Clears any selections. If left-button is held while mouse is moved, selection is created between insertion cursor and under mouse cursor.
Left-mouse double click	Selects the word under the mouse and positions insertion cursor at the start of the word. Holding mouse button while dragging will select all words between insertion cursor and mouse cursor.
Left-mouse triple click	Selects the entire line under the mouse. Holding mouse button while dragging will select all lines between insertion line and mouse cursor line.
Shift + Left-mouse click + drag	Adjusts the end of the selection nearest the mouse cursor when the left button is pressed.
Shift + Left-mouse double click + drag	Adjusts the end of the selection nearest the mouse cursor in whole word units.
Shift + Left-mouse triple click + drag	Adjusts the end of the selection nearest the mouse cursor in line units.
Control + left-mouse click	Repositions the cursor without affecting the selection.
Middle-mouse click	Selection is copied into the text at the position of the mouse cursor.
Middle-mouse click + drag	Moves the current view of the text window.
Insert key	Inserts the current selection at the position of the insertion cursor.
Left/Right key	Moves the insertion cursor one position to the left/ right and clears the selection
Shift + left/right key	Moves the insertion cursor one position to the left/ right and adds the character to the selection.
Control + left/right key	Moves the insertion cursor to the left/right by one word.
Shift + Control + left/right key	Moves the insertion cursor to the left/right by one word and adds the word to the selection.
Up/Down key	Moves the insertion cursor one line up/down and clears the selection.
Shift + up/down key	Moves the insertion cursor one line up/down, extending the selection.
Control + up/down key	Moves the insertion cursor by paragraphs (groups of lines separated by blank lines).

Key/Mouse Binding	Function
Shift + Control + up/down key	Moves the insertion cursor by paragraphs, extending the selection.
Next/Prior key	Moves insertion cursor forward/backward by one screenful of text and clears the selection.
Shift + next/prior key	Moves insertion cursor forward/backward by one screenful of text, extending the selection.
Control + next/prior key	Moves screen forward/backward by one screenful of text without affecting insertion cursor or selection.
Home key OR Control + 'a' key	Moves the insertion cursor to the beginning of its current line and clears any selection.
Shift + Home key	Moves the insertion cursor to the beginning of the line, extending the selection to that point.
Control + Home key	Moves the insertion cursor to the beginning of the text and clears the selection
Shift + Control + Home key	Moves the insertion cursor to the beginning of the text, extending the selection.
End key OR Control + 'e' key	Moves the insertion cursor to the end of its current line and clears the selection.
Shift + End key	Moves the insertion cursor to the end of its current line, extending the selection to that point.
Control + End key	Moves the insertion cursor to the end of the text and clears the selection.
Shift + Control + End key	Moves the insertion cursor to the end of the text, extending the selection.
Control + '/' key	Selects all of the text.
Control + '\' key	Clears the selection.
Delete key	Deletes the selection (if one exists) or deletes the character to the right of the insertion cursor.
Backspace key	Deletes the selection (if one exists) or deletes the character to the left of the insertion cursor.
Control + 'k' key	Deletes from the insertion cursor to the end of the line. If the insertion cursor is already at the end of the line, the newline character is deleted.
Control + 'o' key	Opens a new line by inserting a newline character in front of the insertion cursor without moving the insertion cursor.
Multicursor Bindings	

Key/Mouse Binding	Function
Alt + Left mouse click	Adds a cursor to the multicursor list at the character under the mouse cursor. Also makes the current cursor the anchor cursor.
Alt + Right mouse click	Adds one or more cursors between the anchor cursor and the current cursor such that one cursor will be placed on each line at the same column location as the anchor cursor.
Block Selection Binding	
Shift + Alt + Left mouse click + drag	Selects a column of text with the upper left corner of the selection starting at the button press position and the lower right corner ending at the button release position.

Find Highlighting

TKE supports searching within a text window via the "Find" menu. When a string is searched within the text window, all matching text will be highlighted and the insertion cursor will be placed at the beginning of the first matched text. This allows you to quickly see all matches within the text window.

Find in Files

When the user performs a "Find in Files", a special tab will be added to the editing pane. This file will contain snippets of lines of text from all files that have matches to the search text. Within the window, all matching text will be highlighted the same yellow that is used for normal searching. If the user left clicks or hits the space bar when the insertion cursor is within on any matching text within this tab, the corresponding file will automatically be added to the editing pane as a new tab and the insertion cursor will be placed at the beginning of the matching text in the file. This allows you to quickly find and get to the matches within the editor.

Status Bar

The status bar is the area located at the bottom of the application window. It's function is to display the following information to the user.

- Vim mode (if the editor is currently in Vim mode)
- Current row and column position of cursor within the current editor
- Informational, temporal messages provided by the application
- Current mode of auto-insert for the current editor
- Display current syntax applied to current editor (and ability to change that language).

The following image is a representation of the status bar.

Vim Mode

If the current editor is in Vim mode, the right-most indicator will specify the current mode that Vim is in.

Position Status

The position status information is located on the left side of the status bar. If a file is currently being edited, the current row and column position of the cursor is displayed. Additionally, if the editor is running in Vim mode, the current Vim mode is displayed just to the right of the position information.

Message Display

To the right of the position status information is a typically blank area which can be used to display temporary informational messages to the user from the application. If a message is displayed in this area, it will appear and then disappear after several seconds (keeping the message area blank again).

Auto-Indent Display

Near the right side of the status bar is an indicator of the auto-indent mode. A value of "IND" indicates that the auto-indent mode is on. A lack of a value indicates that the auto-indent mode is off. To change the value of the auto-indent mode, select the "Edit -> Enable/Disable Auto-Indent" menu option.

Syntax Display

On the right side of the status bar is the syntax display area. If a file is currently being edited, the current syntax highlighting language is displayed. To change the current language, simply left-click on the language name and select a different language from the list. If the current file cannot be automatically discerned by TKE, a value of "<None>" will be displayed in the status bar.

Chapter 5: Command Launcher

The command launcher provides access to all of the available functionality from anywhere within the application. To call up the launcher, simply hit the key combination (by default, the key combination is Control-Space but this can be changed within the menu bindings file). The resulting widget is a simple entry field displayed in the upper center portion of the window. The cursor will be placed within the entry field for immediate command entry.

To perform a command, simply begin typing the name of the command that you wish to perform. As you enter characters, the command list will be immediately updated with the best matches. The command launcher uses a fuzzy search algorithm for matching that remembers the most used commands based on the input string, allowing you to quickly perform most commands with only a few typed characters.

If one or more matches are found, the top-most entry will be the best match. The best match will also be selected. To execute the best match, simply enter the RETURN key. To change the selection to another displayed match in the list, simply use the up/down arrow keys until the desired command is selected and hit the RETURN key.

The following table describes the types of commands that can be executed within the command launcher along with any special characters that call up specific functionality.

Command Type	Description	Character Sequence
Menu commands	Any menu item commands can be executed from within the launcher	(Enter any portion of the menu command string)
Clipboard History	Inserts any of the items stored in the clipboard history into the current editor and/or copies the text into the clipboard.	#…
Snippet insertion	Inserts any of the language-specific snippets available for the current editor.	:…
Symbol Jumping	Jump to any supported language symbol (i.e., procedure, function, etc.) in the current editor	@…
Marker Jumping	Jump to any marker in the current editor	,…
Calculator	Perform numerical calculator expressions (any valid numerical Tcl expression is allowed). Selected result is copied to the clipboard.	(Enter any valid Tcl calculation)
URL launcher	Open a specified URL in the local web browser or recall a previously used URL from history and open that location.	(Enter any valid URL)
Plugin installation	Displays all available plugins that can be installed. Selecting a plugin in the resulting list installs the plugin.	install
Plugin uninstallation	Displays all available plugins that can be uninstalled. Selecting a plugin in the resulting list uninstalls a plugin.	uninstall
Syntax modification	Changes the syntax highlighting rules for the current editor	• Enter a name of any supported language OR • Enter "Syntax:" for a full list of all available languages
Theme modification	Changes the syntax highlighting color scheme for all editors	• Enter a name of any installed theme OR • Enter "Theme:" for a full list of all available themes

In addition to the normal command launcher UI (entry field with a list of matching commands listed below), the command launcher also has a preview window that is available for a subset of

functionality. The preview window will be displayed below the entry field and to the right of the command list. Highlighting a command in the command list will update the preview window. The preview window is available for the following command launcher functions.

Function	Displayed in Preview
Snippets	Raw snippet content from the snippet file
Clipboard history	Full content for a paste item
Plugin installations	Revision and description of the selected plugin

Chapter 6: Vim Commands

The editor supports a subset of the classic Vim functionality as well as some useful extensions while operating in Vim mode. To edit documentation in Vim mode, go to the "Tools" menu and click on the "Vim mode" option. When the mode is selected in the menu, the editor will respond to Vim input. To place the editor back into standard editing mode, click on the "Vim mode" menu option again. To set the default editing mode, go to the preferences file and set the "Editing/VimMode" value to a value of 1 (for Vim mode) or 0 (for standard mode).

Standard Vim Commands

The following table describes the available standard Vim functionality.

Any characters in bold (ex. **b**) represent the actual character. Any characters in all caps (ex. ESC or CONTROL-) represent its associated key on the keyboard (these are unprintable characters). Any characters in grey italics (ex. *num*) font represent variables whose value is described in the description field.

Command or KEY	Description
Line numbers	
.	Specifies current line.
^	Specifies the first line in the file.
$	Specifies last line in the file.
number	Specifies the line at line number *number*.
marker_name	Specifies the line marked by *marker_name*.
Undoing/Cancelling commands	
ESC	Cancels unexecuted command or if in editing mode, ends editing mode to return to command mode. If the current mode is the command mode, any selections or search highlighting is cleared from the current editor.
u	Counteracts last command that changed the buffer.
Repeating a command	
.	Repeats the last command that changed the buffer.
Reading in a file	
:n	Edits the next file in the editor tab order.
:e *filename*	Edits the specified file (if the file is not already opened, opens the file in a new tab).
:e#	Edits the previously edited file.
:r *filename*	Places a copy of the specified file below the current line.
CONTROL-**g**	Displays number of lines and characters in the current file in the status bar.
Saving/Closing a file	
:w	Writes file under the original name. If an original name has not been specified, a "Save As" window will be displayed.
ZZ or **:wq**	Writes the file under the original name and closes the current tab. If an original name has not been specified, a "Save As" window will be displayed.

Command or KEY	Description
:q	Closes the current tab. If the text has been modified since the last save, a prompt will be displayed asking if you would like to save before closing.
:q!	Closes the current tab regardless of the modification status. Changes will not be saved and a prompt will not be displayed.
:w *filename*	Writes the current file under the specified filename.
:*x*,*y***w** *filename*	Writes the specified range of lines to the given *filename*.
:*x*,*y***w!** *filename*	Writes the specified range of lines to the given *filename* overwriting the contents of the file.
Searching/Replacing	
/*string*	Finds all occurrences of the given string, jumping the cursor to the first occurrence below the current line.
?*string*	Finds all occurrences of the given string, jumping the cursor to the first occurrence above the current line.
n	Jumps to the next occurrence of the previous search.
?	Jumps to the previous occurrence of the previous search.
:*x*,*y***s/***oldstring***/***newstring***/***flags*	Finds and replaces one or more occurrences of "oldstring" with "newstring" where "oldstring" can be any Tcl regular expression. The "flags" value if empty, causes only the first match to be replaced in the given range. If "flags" is set to g, all matches are replaced in the given range.
*****	Searches the text for the next occurrence of the current word.
Inserting/Replacing text	
i	Inserts before the current character.
a	Inserts after the current character.
A	Inserts at the end of the current line.
I	Inserts at the beginning of the current line.
o	Inserts below the current line (opens new line).

Command or KEY	Description
O	Inserts above the current line (opens new line).
r	Replaces the current character (no ESC necessary).
R	Replaces from current cursor position to end of line; does not change characters not typed over.
cw	Replaces the current word
ci*char*	Replaces all text contained within the pair of *char* characters before and after the current insertion cursor.
cc	Replaces the current line
C	Replaces all text from the current insertion cursor to the end of the current line.
Joining text	
J	Joins the current line and the line below it.
*#***J**	Joins # lines, starting with the current line
Changing case	
~	Changes case of the current character.
Moving around in a file	
h	Moves left one character
j	Moves down one line
k	Moves up one line
l	Moves right one character
w	Moves the insertion cursor to the beginning of the next word.
b	Moves the insertion cursor to the beginning of the previous word.
0 or **^**	Moves to the beginning of current line
$	Moves to the end of the current line
:*#*	Moves to line # (note: # value of 0 or 1 takes you to the first line)
G	Moves to the end of the file
*#***G**	Moves to the line #.

Command or KEY	Description
RETURN	Moves the insertion cursor to the first non-whitespace character in the line after the current line.
-	Moves the insertion cursor to the first non-whitespace character in the line before the current line.
H	Moves the insertion cursor to the first line on the screen.
M	Moves the insertion cursor to the middle line on the screen.
L	Moves the insertion cursor to the last line on the screen.
*#***l**	Moves the insertion cursor to the specified column in the current line.
CONTROL-**f**	Scrolls forward one screen
CONTROL-**b**	Scrolls backward one screen
Deleting text	
x	Deletes the current character
*#***x**	Deletes # characters, starting with current character
dw	Deletes current word
*#***dw**	Deletes # words, starting with the current word
dd	Deletes current line (deleted contents are placed in clipboard)
*#***dd**	Deletes # lines, starting with the current line (deleted contents are placed in clipboard).
D	Deletes from current cursor position to the end of the line.
:*x*,*y***d**	Deletes lines x through y (deleted contents are placed in clipboard).
Copying text	
y	Yanks the current character (yanked contents are placed in clipboard).
*#***y**	Yanks # characters, starting with the current character (yanked contents are placed in clipboard).

Command or KEY	Description
yw	Yanks the current word.
*#*__yw__	Yanks # words, starting with the current word.
yy	Yanks the current line (yanked contents are placed in clipboard).
*#*__yy__	Yanks # lines, starting with the current line (yanked contents are placed in clipboard).
:*x*,*y*__y__	Yanks lines x through y (yanked contents are placed in clipboard).
Pasting text	
p	Places contents in the clipboard below the current line.
P	Places contents in the clipboard above the current line.
Visual (Selection) mode	
v	Changes the mode to visual mode. Using the navigation commands during visual mode will change the current selection.
Miscellaneous	
%	Moves to matching (,), [,], { or }

Extended Vim Commands

To provide additional functionality to the user, Vim command extensions have been added to the standard list. The following table specifies these Vim command extensions.

Command or KEY	Description
Tab or text pane traversal	
:N	Changes to the previous tab.
:p	Changes focus to the tab in the other opened text pane (only available when the other pane exists).
Marker (bookmark) creation	
:m	Creates a marker (bookmark) for the current line. This marker can be named in the subsequent entry field that is displayed. Hitting return in the marker entry field will create a named marker (or if no text was typed, an unnamed marker). Hitting the ESC key in the entry field will cancel the marker creation process.
:m *marker*	Creates a marker (bookmark) for the current line using the provided name.
:cd *directory*	Changes the current working directory (as displayed in the title bar) to the specified directory.
Multicursor Functionality	
s	Sets a multicursor cursor on the current character. Also makes this character the anchor for any multiline cursor sets.
S	Sets multicursors for every line between the current line and the last multicursor anchor, inclusive. Each multicursor will match the column of the anchor multicursor.
J	When one or more multicursors are set, moves all of the cursors down one line
K	When one or more multicursors are set, moves all of the cursors up one line.
H	When one or more multicursors are set, moves all of the cursors to the left by one character.
L	When one or more multicursors are set, moves all of the cursors to the right by one character.

Command or KEY	Description
#	When one or more multicursors are set, allows the user to insert an ascending numerical values as specified in the subsequent "Starting number:" entry field. The content of the starting number determines what the base of the number will be. The following are valid starting number representations: *prefix*[0-9]+ • Inserts prefix followed by a decimal value. *prefix*d[0-9]+ • Inserts prefix followed by a decimal value preceded by "d" *prefix*b[0-1]+ • Inserts prefix followed by a binary value preceded by "b" *prefix*o[0-7]+ • Inserts prefix followed by an octal value preceded by "o" *prefix*[xh][0-9a-fA-F]+ • Inserts prefix followed by a hexadecimal value preceded by either "x" or "h". Note: If a value is not specified, a value of zero is assumed.
String/Bracket Insertion	
‘	If a selection exists, all selected code will be encapsulated in single quotes. If no selection exists and current insertion cursor is within a single quote quotation, the right single quote is moved one word to the right. If none of the above is true, the current word is encapsulated in single quotes.
“	If a selection exists, all selected code will be encapsulated in double quotes. If no selection exists and current insertion cursor is within a double quote quotation, the right double quote is moved one word to the right. If none of the above is true, the current word is encapsulated in double quotes.

Command or KEY	Description
{	If a selection exists, all selected code will be encapsulated in curly brackets. If no selection exists and current insertion cursor is within a curly bracketed code block, the right curly bracket is moved one word to the right. If none of the above is true, the current word is encapsulated in curly brackets.
[	If a selection exists, all selected code will be encapsulated in square brackets. If no selection exists and current insertion cursor is within a square bracketed code block, the right square bracket is moved one word to the right. If none of the above is true, the current word is encapsulated in square brackets.
(	If a selection exists, all selected code will be encapsulated in parenthesis. If no selection exists and current insertion cursor is within a parenthetical code block, the right parenthesis is moved one word to the right. If none of the above is true, the current word is encapsulated in parenthesis.
<	If a selection exists, all selected code will be encapsulated in angled brackets. If no selection exists and current insertion cursor is within a angle bracketed code block, the right angle bracket is moved one word to the right. If none of the above is true, the current word is encapsulated in angled brackets.
Line Bubbling	
CONTROL-j	Moves the current line down one line, moving the line below the current line above it. If lines are selected, this command moves all of the selected lines down by one line.
CONTROL-k	Moves the current line up one line, moving the line above the current line below it. If lines are selected, this command moves all of the selected lines up by one line.

Chapter 7: Snippets

Snippets allow the user to enter a short bit of text (herein called the *abbreviation*) which will be replaced by a larger piece of text (called the *snippet*) when a whitespace character is entered. For example, suppose we have defined an abbreviation called "hw" which is assigned the snippet text "Hello, world!". If we enter the following string in an editor:

```
cout << "hw
```

and follow it with hitting either the SPACE, RETURN or TAB key, the editor will replace the abbreviation to look like the following:

```
cout << "Hello, world!
```

In addition to simple ascii text, the snippet text can contain various styles of variables. For example, suppose we are editing a file called "foobar.cc" and have defined an abbreviation called "cf" which is assigned the snippet text "$FILENAME". If we enter the following string in an editor:

```
File:  cf
```

and follow it with hitting either the SPACE, RETURN or TAB key, the editor will replace the abbreviation to look like the following:

```
File: foobar.cc
```

Snippet Variables

The following table represents the various variables that can be used within snippet text. Note that all variables are expanded at the time the snippet replacement occurs. Additionally, the DOLLARSIGN ($) and BACKTICK (`) characters are special characters. If you require these characters to be treated as literal characters in your snippet, you will need to escape these characters by placing a BACKSLASH (\) character just before it.

Variable	Description
$CLIPBOARD	Places the contents that are currently in the clipboard at this variable's location.
$CURRENT_LINE	Places the current line contents (minus the abbreviation) at this variable's location.
$CURRENT_WORD	Places the current word at this variable's location.
$DIRECTORY	Places the current directory at this variable's location.
$FILEPATH	Places the current file pathname at this variable's location.
$FILENAME	Places the root file name at this variable's location.
$FILENAME_UPPER	Places the root file name entirely capitalized at this variable's location.
$LINE_INDEX	Places the position of the current insertion cursor (specified as *line.column*) at this variable's location.
$LINE_NUMBER	Places the line position of the current insertion cursor at this variable's location.
$CURRENT_DATE	Places the current date at this variable's location. The date is specified as MM/DD/YYYY.
$0	Places the cursor at this variable's location after the entire snippet has been expanded.
$*number*	Places the cursor at this variable's location in the order of *number*. Hitting the TAB key will jump the cursor to the next cursor stop. For example, if a snippet uses the variables "$1 … $2", the cursor will first be placed at location "$1" and when the TAB key is pressed, the cursor will jump to the location of "$2". If more than one "$1" is use within the same snippet, the text that is entered in the first occurrence of this variable will also be entered in all other places within the snippet that share the same number.

Variable	Description
${*number*:*value*}	Places the cursor at this variable's location in the order of *number*, placing the string *value* at the cursor's location. The string *value* can be used as a placeholder to remind the user what information to insert at that location. The *value* string will be automatically selected when the snippet is inserted so that immediately typing text will delete the *value* string with the user's entered string.
`*shell_command*`	Executes the specified command between the back tick characters.

Creating Snippets

Snippets are maintained in individual files according to the syntax language of the buffer that uses them. Therefore, all Tcl snippets will be placed in a Tcl.snippets file within your ~/.tke/snippets directory while C++ snippets will be placed in a C++.snippets file in the same directory.

To create or edit a snippet for a specific language, make sure that the current editor language is set to the language of the snippet being created or modified and select the "Edit / Snippets / Edit" menu command. This will add the language-specific snippets file from your ~/.tke/snippets directory into the editor in a new tab. If the file doesn't yet exist, TKE will automatically create it for you.

To add a new snippet, you will use the following syntax rules:

1. A new snippet is designated by the name "snippet" on a new line followed by whitespace and the abbreviation to use for the snippet.
2. Abbreviations must not contain any whitespace characters.
3. The snippet text must follow the "snippet" line on the next line.
4. Each snippet line must be preceded by a TAB character.
5. All DOLLARSIGN ($) and BACKTICK (`) characters within the snippet text will be treated as the start of variables or shell commands unless they are escaped by the BACKSLASH (\) character.
6. Any text that is specified between snippets is ignored by the snippet parser (i.e., you can use this space to document your snippets if desired).

In the following examples, the PERIOD (.) character is only there to show you the end of a blank line for demonstration purposes. It should not be regarded as the literal PERIOD character.

The following is legal snippet syntax:

```
snippet hw
	Hello, world!
```

```
.
snippet 2day
	$CURRENT_DATE
.
snippet stuff
	set stuff \$foobar
```

The following snippet is invalid (i.e., "hw" would not be treated as an expandable abbreviation) because it breaks rule #1:

```
snip hw
	Hello, world!
```

The following snippet is invalid (i.e., "h w" would not be treated as an expandable abbreviation) because it breaks rule #2:

```
snippet h w
	Hello, world!
```

The following snippet is invalid (i.e., it would replace the string "hw" with the empty string) because it breaks rule #3:

```
snippet hw
.
	Hello, world!
```

The following snippet is invalid (i.e., it would replace the string "hw" with the empty string) because it breaks rule #4:

```
snippet hw
Hello, world!
```

The following snippet is invalid (i.e., an error message would occur because $foobar is not a valid variable name) because it breaks rule #5:

```
snippet stuff
	set stuff $foobar
```

Save as many snippets in the file as you need. Once you have saved the snippets file, the snippet will be immediately available for you to use within an editor, editing the same syntax as the snippet file.

Chapter 8: Preferences

Preferences allow you to customize your experience with TKE by modifying various behaviors and/or appearances within the tool. TKE preferences are handled by two files: a base preference file located in the TKE installation directory (in data/preferences.tkedat) and a user-specific preference file located at ~/.tke/preferences.tkedat.

The preferences files are read and handled at two event times: when TKE is started and when the user preference file is written/saved. If the user preference file does not exist, the base preference file is copied and to the user's ~/.tke directory and the resulting file is read and its values used. If the user preference file already exists, its modification timestamp is compared to the timestamp of the base preference file — if the base file is newer than the user preference file, the base preference file is read in, the user preference file values are used in place of the base preference file and the resulting content is written back out to the user preference file. This makes sure that the user's preference file is always up-to-date with the preferences currently available in the tool. If the user preference file's modification time is the same as or newer than the base preference file, the user preference contents are used to configure the tool.

The content found in the preference file is a fairly straightforward ASCII name-value pair. Strictly speaking, the file uses a Tcl list syntax; however, the file is specially handled to verify that it does not contain any syntax that can be executed for security purposes. All names and values must be hard-coded values. Additionally, Tcl-style comments are allowed and provided. When you view the preference file, you will notice that every preference option contains documentation (in the form of comments) directly above the value that it describes along with a list of valid values that the preference can be set to.

Within TKE, you will only be able to view the base preference file (since these values will effect more than just the individual user). To do so, go to the "Edit / Preferences / View base" menu command. This will add the global preferences file in a separate editor tab in readonly mode. This allows you to see what each default global value is set to and allows you to read the preference documentation for setting your own value.

To modify a preference value, it is preferred that you do so within the tool using the "Edit / Preferences / Edit user" menu command. This will add the user preferences file in a separate editor tab and any saves performed on this window will automatically apply the preference setting changes to the application without requiring a restart.

To get a user copy of the base preferences file or to "reset" the user preferences to match the base preferences (including documentation comments), use the "Edit / Preferences / Reset user to base" command. This will create a copy of the global preferences in the current user's preference file. Note that this operation is destructive and irreversible — the old preference file will be destroyed and replaced with the new file.

Since all preference values are adequately documented in the preference file itself, this document will not duplicate this information.

Chapter 9: Menu Binding

The menu binding capability within TKE simply allows any user to customize the keyboard shortcuts to launch any menu command. By default, TKE contains a set of menu bindings; however, any of the menu items can be overridden.

The default (global) menu binding file is located in the TKE installation directory (in data/menu_bindings.tkedat). In addition to the global file, each user will have their own menu binding file which overrides the global file settings. This file is located at ~/.tke/menu_bindings.tkedat.

To view the global menu bindings file from within TKE, go to the "File / Menu Bindings / View global" menu command. This will create a new tab in the editor with the global file loaded in readonly mode. It is recommended that the global file content not be changed as any changes to this file to menu items not overridden in another user's menu binding file will change their environment.

To edit the user menu bindings file from within TKE, go to the "File / Menu Bindings / Edit user" menu command. This will create a new tab in the editor with the user's own menu binding file loaded. Any saves of this file with immediately cause TKE to re-evaluate the menu bindings file, updating the menu shortcuts.

File Format

The format of the menu binding file is essentially a Tcl key-value list where each key is the hierarchical path to a menu command and the value is the keyboard shortcut that will invoke the command. However, before the file is read, it is parsed to verify that the file doesn't contain any Tcl commands.

The following is a breakdown of the rules governing the parsing of the menu values within this file.

1. All menu hierarchies MUST start with .menubar.*lowercase_menu_name*
2. The menu specified must match the file command exactly (case-sensitive, space sensitive).
3. Each submenu must be preceded by the SLASH (/) character (ex. .menubar.file/Open File).
4. The entire menu hierarchy must be surrounded by curly brackets (ex. {.menubar.file/Open File}).

The following rules describe the parsing rules for the keyboard shortcut value.

1. All keys that are part of the combination must be separated by a single DASH (-) character (ex. a combination of the CONTROL, ALT and 'a' keys should be described as Cntl-Alt-A).
2. All alphabet keys must be capitalized.
3. No whitespace is allowed in the shortcut value.
4. The following special key values are specified as follows:

- CONTROL: "Cntl"
- ALT: "Alt"
- SHIFT: "Shift"
- COMMAND: "Cmd"
- SUPER (Windows key): "Super"
- SPACE: "Space"
- TAB: "Tab"

The following is an example of a user menu binding file (this file was specified on a Mac hence the usage of the command key):

```
# File menu bindings
{.menubar.file/New}                          Cmd-N
{.menubar.file/Open File...}                 Cmd-O
{.menubar.file/Open Directory...}            Cmd-Shift-O
{.menubar.file/Save}                         Cmd-S
{.menubar.file/Save As...}                   Cmd-Shift-S
{.menubar.file/Lock}                         Cmd-L
{.menubar.file/Close}                        Cmd-W
{.menubar.file/Quit}                         Cmd-Q

# Edit menu bindings
{.menubar.edit/Undo}                         Cmd-Z
{.menubar.edit/Redo}                         Cmd-R
{.menubar.edit/Cut}                          Cmd-X
{.menubar.edit/Copy}                         Cmd-C
{.menubar.edit/Paste}                        Cmd-Shift-V
{.menubar.edit/Paste and Format}             Cmd-V

# Find menu bindings
{.menubar.find/Find}                         Cmd-F
{.menubar.find/Find and Replace}             Cmd-Shift-F
{.menubar.find/Select next occurrence}       Alt-N
{.menubar.find/Select previous occurrence}   Alt-P
{.menubar.find/Append next occurrence}       Alt-D
{.menubar.find/Select all occurrences}       Alt-A

# Text menu bindings
{.menubar.text/Comment}                      Cmd-I
{.menubar.text/Uncomment}                    Cmd-Shift-I

# Tools menu bindings
{.menubar.tools/Launcher}                    Ctrl-Space
{.menubar.tools/View Sidebar}                Alt-B
{.menubar.tools/Vim Mode}                    Cmd-M
```

Chapter 10: 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 basename 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.

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}
```

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} *}
```

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}} *
}
```

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, 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 *tag_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.
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} {

  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 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/C++.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}

tabsallowed
{0}

casesensitive
{1}

indent
{\{}

unindent
{\}}

lcomments {{#}}

bcomments {}

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
}

numbers
{
  HighlightClassForRegexp {\m[0-9]+}
}

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

precompile {}

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
  }
}
```

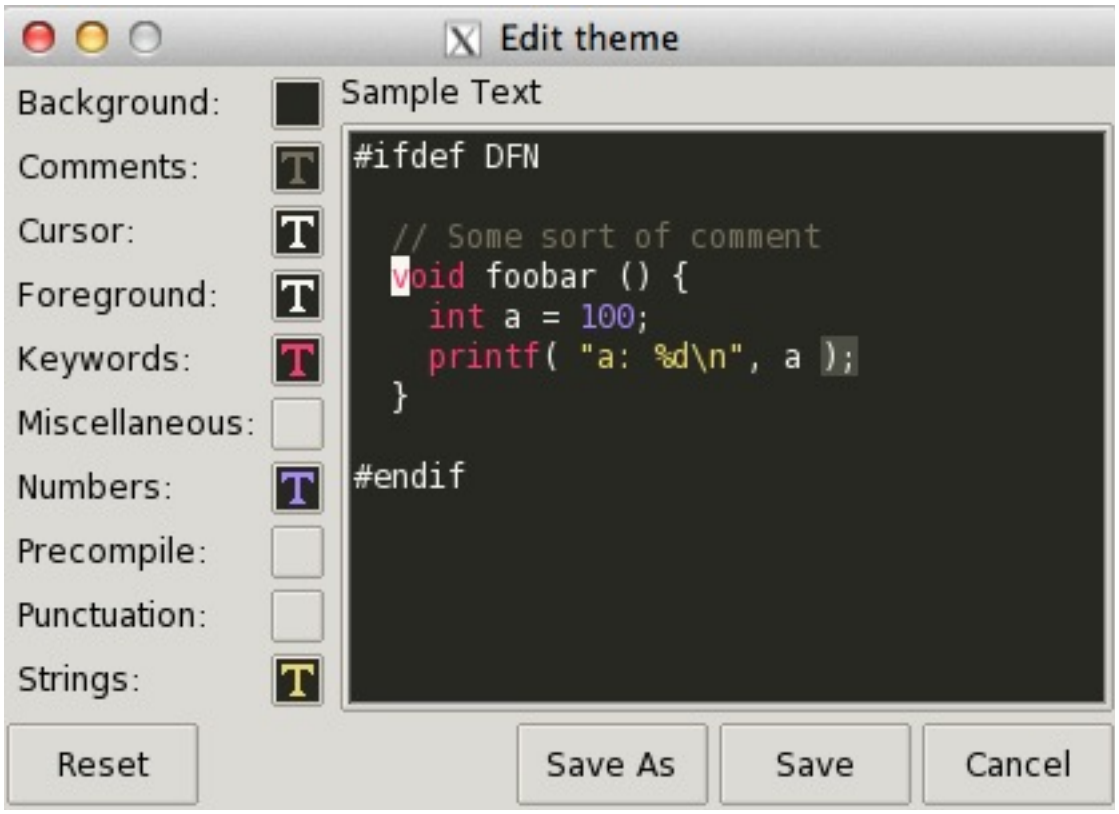
```
miscellaneous3 {
  HighlightClassForRegexp {\.[a-zA-Z0-9\_\-]+}
  HighlightClassWithOnlyCharStart \$
}
```

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. Tab characters should not be used for indentation.
3. Use case sensitive matching for parsing purposes.
4. Whenever an open curly bracket is found, increase the indentation level, and whenever a closing curly bracket is found, decrease the indentation level.
5. All comments start with the HASH (#) character.
6. All strings start and end with the QUOTE (") character.
7. Apply keyword coloring to the list of keywords (ex., "after", "bindtags", "uplevel", etc.)
8. Whenever a "proc" keyword is found, use the name of the proc as a searchable symbol in the file.
9. Highlight any integer values as numbers.
10. Highlight the], [, {, } characters as punctuation
11. There are no precompiler syntax to be colored.
12. Highlight Tk keywords in a different color than Tcl keywords.
13. Highlight Tcl/Tk option values in a different color than normal Tcl keywords.
14. Highlight Tk window pathnames in the miscellaneous3 color.
15. Highlight variables in the miscellaneous3 color.

Chapter 11: Theme Creator

The Theme Creator is a GUI interface for creating or editing the color schemes for a given Theme. The window displays a preview of the syntax highlighting scheme in a sample window. The following image is a representation of this window.



On the left side of the window is a list of syntax coloring categories. On the left is the name of the category while on the right is a button which displays a sample of the current color (on the current background). Clicking on the button will display a color picker window that will allow you to select any color to use. Clicking the "OK" button in the color picker will cause the sample in the button to get updated as well as the color in the preview window (on the right side of the window). Some buttons have no color associated with them (as represented by a blank sample on the button). Clicking one of these buttons, selecting a color and clicking the "OK" button will associate a color with the button.

The following table describes the various button categories:

Category	Description
Background	The background color of the editor window.
Comments	The color of any line or block comments in the code (as represented by the "lcomments" and "bcomments" values in the .syntax file)
Cursor	The color of the cursor in both block and line format.
Foreground	The default color of the text if no other syntax highlighting colors are used on the text.
Keywords	The color of any keywords or symbols as specified in the .syntax file.
Miscellaneous1	The color of any miscellaneous1 items as specified in the .syntax file.
Miscellaneous2	The color of any miscellaneous2 items as specified in the .syntax file.
Miscellaneous3	The color of any miscellaneous3 items as specified in the .syntax file.
Numbers	The color of any numbers as specified in the .syntax file.
Punctuation	The color of any punctuation as specified in the .syntax file.
Strings	The color of any text which resides in between string identifiers (as specified in the .syntax file)

On the right side of the theme creator window is the preview area. You may not directly alter any text or colors in this display; however, the current colors chosen on the left side of the window will be displayed to give you an idea of how the window will look with the current color scheme.

On the bottom of the window is the button bar.

The "Reset" button will undo any changes that you have made to the current theme since opening it for editing.

The "Create" button (not shown in the above window) will display a window allowing a name to be associated with the theme. Enter a name in the entry field (the name of the theme will have a ".snippet" extension automatically added to the file) and click the "OK" button to save the theme and exit the theme creator utility. Click on the "Cancel" button to exit the window and return to the theme creator window.

The "Save As" button will allow you to save the current theme under a different name. This button will only be displayed when editing an existing theme. Clicking the "Save As" button will display a window allowing a new name to be associated with the current theme settings. Enter

a name in the entry field (the name of the new theme will have a ".tketheme" extension automatically added to the file) and click the "OK" button to save the theme and exit the theme creator utility. Click on the "Cancel" button to exit the window and return to the theme creator window.

The "Save" button will save the current theme settings under the current theme name. This button is only displayed when you are editing an existing theme or importing a TextMate theme (the name of the TKE theme will match the base name of the TextMate theme with the exception that the extension of the resulting file will be ".tketheme" and the file will be automatically put into the TKE installation directory under data/themes).

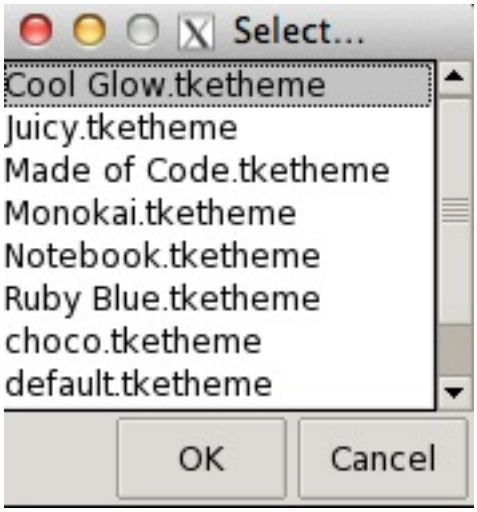
The "Cancel" button will exit the theme create window, discarding any changes.

Creating a New Theme

To create a new theme, select the "Tools / Theme Creator / Create new…" menu command. This will display the theme creator window. Click on the various coloring categories to create a color scheme to your liking. When the settings are to your satisfaction, click on the "Create" button, enter a name for the theme, and click the "OK" button to save the changes.

Editing an Existing Theme

The edit an existing theme, select the "Tools / Theme Creator / Edit…" menu command. This will display a theme selection window, displaying all of the currently available themes. The following image is a representation of this window.



Select a theme name and click on the "OK" button to bring up the theme creator window. The colors represented in the theme will be automatically preset in the window. Make any changes

to the color scheme and either click the "Save" button to change the theme or click the "Save As" button to save the settings under a new name.

Importing a TextMate Theme

In addition to being able to create a theme from scratch, TKE also has the ability to import an existing TextMate formatted theme. The import process will parse the contents of the TextMate theme and pull out various tag values that make a TKE theme look as close as possible to the original TextMate theme. This process is not perfect, however, so after a theme has been imported, the theme creator window is displayed with the parsed interpretation of the original theme. If there are any colors will are not quite right, you can simply select the offending color categories and set the associated color to match the original theme.

To import a TextMate theme, download the TextMate theme file to your local file system (the location at this point is not important), select the "Tools / Theme Creator / Import TextMate theme…", and find and open the file in the open file dialog window. This will import the file content and display the theme creation window.

After the theme settings are set to your satisfaction, click on the "Save" button to save the theme to the themes directory (the base name of the TextMate theme will be the base name of the TKE theme but the extension will be ".tketheme").

Chapter 12: Plugins

In addition to all of the built-in functionality that comes standard, TKE also provides a plugin API which can allow development of new functionality and tools without needing to modify the source code. TKE ships with a small set of these plugins which are located in the TKE installation directory under the "plugins" directory.

Out of the box, plugins are not installed and available from within the tool; however, any plugin can be installed, uninstalled or reloaded within TKE (no restart is required). This will save those plugin settings to the user's "plugin.dat" file in their ~/.tke directory. When TKE is exited and restarted, any previously installed plugins will be installed on application start.

Plugins can interface to TKE in a variety of ways. Which interfaces are used is entirely up to the developer of the plugin. Each plugin can create multiple interfaces into the tool to accomplish its purposes. The following table lists the various ways that plugins can interface into TKE.

Interface Type	Description
menu	Plugins can create an entire subdirectory structure under the "Plugins" menu.
tab popups	Create menu items within a tab's popup menu.
sidebar popups	Create menu items within any of the sidebar's popup menus.
application events	Plugins can be run at certain application events (i.e., on start, opening a file, closing a file, saving a file, receiving editor focus, on exit, etc.)

Installing a Plugin

Installing a new plugin is accomplished from the "Plugins / Install…" menu command. Doing so will display the command launcher in plugin installation mode. Any uninstalled plugins will be displayed in the launcher. To install a plugin from the list, simply select a plugin name from the list or begin typing a portion of the plugin name in the entry field until it is selected, and hit the RETURN key.

This will cause the plugin to be immediately installed. No application restart is required.

Uninstalling a Plugin

Uninstalling a plugin is similar to the process of installing a plugin. Select the "Plugins / Uninstall…" menu command. This will display the command launcher in plugin uninstallation

mode. Any installed plugins will be displayed in the launcher. To uninstall a plugin from the list, simply select a plugin name from the list or begin typing a portion of the plugin name in the entry field until it is selected, and hit the RETURN key.

This will cause the plugin to be immediately uninstalled. No application restart is required.

Reloading a Plugins

Reloading plugins is only necessary in two cases. In the first case, the user installs a new plugin in the TKE installation's "plugins" directory. In this case, the plugins will need to be reloaded so that the new plugin name will be viewable in the plugin install list without requiring an application restart. The second case where plugin reloading is needed is in plugin development (more on this process is described in the "Plugin Development" chapter within this document).

To reload plugins, simply select the "Plugins / Reload" menu command. This will cause all plugins to be immediately reloaded. No application restart is necessary.

Chapter 13: Plugin Development

This chapter is written for anyone who is interested in writing plugins for TKE. The material found in the rest of the document is not necessary to know to use TKE for all other coding purposes and may be ignored.

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 must contain at least the following two files:

File	Description
header.tkedat	Contains plugin information that describes the plugin and is used by the plugin installer.
main.tcl	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.

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.

Tcl Command Differences in Untrusted Mode

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.

Command	Difference Description
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.
load	The requested file, a shared object file, is dynamically loaded into the safe 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 set the state of the newly added menu command to either normal or disabled. The determination of this state is up to the plugin to decide. By default, menu items are set to the "normal" (enabled) state.

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.

Plugin 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.

Command	Difference Description
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.

Command	Difference Description
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.

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 argument *plugin_name* is the name of the plugin that is associated with this launcher 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
```

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). 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 {e-mail address}
```

The value of *email* 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 and how to operate it.

The following is an example of what a plugin header looks 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"
    plugins::word_count::menu_do
    plugins::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 plugins::word_count::menu_do will be run when the command is selected. The plugins::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 {

}
```

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, README files, data files, and images.

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

Appendix A. 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 setting the state of the menu item to normal or disabled as deemed appropriate by the plugin creator. It contains one parameter, the lowest level menu containing the menu item.

Example:

```
proc foobar_menubar_handle_state {mnu} {
  if {$some_test_condition} {
    $mnu entryconfigure "foobar" -state normal
  } else {
    $mnu entryconfigure "foobar" -state disabled
  }
}
```

---

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 setting the state of the menu item to normal or disabled as deemed appropriate by the plugin creator. It contains one parameter, the lowest level menu containing the menu item.

Example:

```
proc foobar_menubar_handle_state {mnu} {
  if {$some_test_condition} {
    $mnu entryconfigure "foobar" -state normal
  } else {
    $mnu entryconfigure "foobar" -state disabled
  }
}
```

---

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 setting the state of the menu item to normal or disabled as deemed appropriate by the plugin creator. It contains one parameter, the lowest level menu containing the menu item.

Example:

```
proc foobar_root_popup_handle_state {mnu} {
  if {$some_test_condition} {
    $mnu entryconfigure "foobar" -state normal
  } else {
    $mnu entryconfigure "foobar" -state disabled
  }
}
```

---

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 setting the state of the menu item to normal or disabled as deemed appropriate by the plugin creator. It contains one parameter, the lowest level menu containing the menu item.

Example:

```
proc foobar_dir_popup_handle_state {mnu} {
  if {$some_test_condition} {
    $mnu entryconfigure "foobar" -state normal
  } else {
    $mnu entryconfigure "foobar" -state disabled
  }
}
```

---

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 setting the state of the menu item to normal or disabled as deemed appropriate by the plugin creator. It contains one parameter, the lowest level menu containing the menu item.

Example:

```
proc foobar_file_popup_handle_state {mnu} {
  if {$some_test_condition} {
    $mnu entryconfigure "foobar" -state normal
  } else {
    $mnu entryconfigure "foobar" -state disabled
  }
}
```

---

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 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 *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 plugins::current_line {

  proc do_cline {btag} {

    bind $btag <<Modified>>    "after idle [list
plugins::current_line::update_line %W]"
    bind $btag <ButtonPress-1> "after idle [list
plugins::current_line::update_line %W]"
    bind $btag <B1-Motion>     "after idle [list
plugins::current_line::update_line %W]"
    bind $btag <KeyPress>      "after idle [list
plugins::current_line::update_line %W]"

  }

  proc update_line {txt} {
```

```
    …

  }

}

api::register current_line {
  {text_binding pretext cline plugins::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]

  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_close action type is called when a file is closed in the editor pane. More specifically, the associated procedure is called prior to the file actually closed; therefore, you can get information about the file being closed in this procedure.

Tcl Registration

```
{on_close 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 the on_close action occurs. 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 being closed.

```
proc foobar_do {file_index} {

  set fname [api::file::get_info $file_index fname]

  puts "File $fname is being closed"

}
```

---

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

}
```

---

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 Procedures

```
     {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.

Appendix B. 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_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::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 (a Tk dialog or message box should be used for this case).

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::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	One of the following values: • 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

api::file::add

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 ?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. The following options are available: • -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. • -saveas *boolean* If set to 0 (default), the file will be saved to the current file; otherwise, the file will always force a save as dialog to be displayed when saving. • -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.

api::sidebar::get_selected_index

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

Call structure

```
api::sidebar::get_selected_index
```

Return value

Integer value specifying the index of the currently selected file/directory in the sidebar. A value of -1 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. The valid values are as follows: • 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

Parameters	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

Parameters	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

None.

Parameters

Parameters	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.

Appendix C. 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

---

ctext

Documentation for the ctext package can be found at http://tcllib.sourceforge.net/doc/ctext.html

---

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