

The Kommander Handbook

Marc Britton, Tamara King, and Eric Laffoon



The Kommander Handbook

Contents

1	Introduction	1
2	Kommander Basics	3
2.1	Concepts	3
2.2	The Editor	5
2.2.1	Main Window	6
2.2.2	The File Menu	6
2.2.3	The Edit Menu	7
2.2.4	The Tools Menu	7
2.2.5	The Layout Menu	8
2.2.6	The Run Menu	8
2.2.7	The Window Menu	9
2.2.8	The Help Menu	9
2.3	The Executor	10
2.3.1	Executor for Programmers	10
2.4	Creating a Dialog	10
3	Reference	11
3.1	Widgets	11
3.2	Specials and Built-in Global Variables	15
3.2.1	Array Function Group	16
3.2.2	File Function Group	17
3.2.3	String Function Group	17
3.2.4	Built-in Globals	17
3.3	DCOP Functions	18
3.3.1	DCOP for Global Variables	18
3.3.2	DCOP for all Widgets	19
3.3.3	DCOP for ListBox and ComboBox Widgets	19
3.3.4	DCOP for CheckBox and RadioButton Widgets	19
3.3.5	DCOP for TabWidget Widgets	20

The Kommander Handbook

4	Extending Kommander	21
4.1	Creating Kommander Widgets	21
5	Tutorials	22
5.1	Using Editor	22
5.2	Globals	22
5.3	DCOP	22
5.4	Slots	22
5.5	Settings	23
5.6	Append	23
5.7	Command Line	23
5.8	Initialize	23
6	Questions and Answers	24
7	Credits and License	25
A	Installation	26
A.1	How to obtain Kommander	26
A.2	Requirements	26
A.3	Compilation and Installation	26
A.4	Configuration	26
B	Glossary	27

Abstract

Kommander is a set of tools that allow you to create dynamic GUI dialogs that generate, based on their state, a piece of text. The piece of text can be a command line to a program, any piece of code, business documents that contain a lot of repetitive or templated text and so on. The resulting generated text can then be executed as a command line program (hence the name 'Kommander'), written to a file, passed to a script for extended processing, and literally anything else you can think of. The best part of it all? You aren't required to write a single line of code!

Chapter 1

Introduction

Introduction Eric Laffoon Kommander is a visual dialog building tool which may be expanded to create full mainwindow applications. The primary objective is to create as much functionality as possible without using any scripting language. This is provided by the following features:

- Specials are prefaced with an '@' like @widgetText. They offer special features like the value of a widget, functions, aliases, global variables and such.
- DCOP integration allows Kommander dialogs to control and be controlled in interactions with other KDE applications. It is a very powerful feature!
- Signals and Slots is a little less intuitive to a new user. It is under review for how we process things in the first major release. These offer a limited event model for when a button is pushed or a widget is changed. Combined with 'Population Text' it is rather powerful.

The central key feature of Kommander dialogs is that you can bind text (Kommander Text) to a widget. So if you have @widget1 and @widget2 and they are line edits you can set Kommander to show their contents by entering @widgetText in their Kommander Text area. Then enter hello in @widget1 and world in @widget2. A button can have the string My first @widget1 @widget2 program in Kommander. If you run this dialog from a console it will output My first hello world program in Kommander.

Hopefully you begin to see a small glimmering of the potential. Kommander enables a much faster design model for simple applications because it allows you to stop thinking so much about language and revert to the more basic and natural conceptual model. In computer language is a means to define concepts and as such it is a layer between concept and implementation that can impede progress with minutia. Kommander seeks to minimize that layer.

Kommander also seeks to build on standards. It is built on the Qt™ Designer framework and creates *.ui files which it renames to *.kmdr. It can easily import any KDE widget and this can be done without having to rebuild Kommander, by using plugins.

The Kommander Handbook

Kommander's other significant factor is how it addresses the requirements of language. Computer languages can be wonderful things but they tend to have their own dogmas and zealots often seeking to provide an advance to GUI design in an integrated development environment. Ironically the acceptance of such IDEs is limited by the number of people willing to adopt a new language to gain access to a desired feature. It is really not reasonable to expect people to need to change over to a dozen languages to access various feature sets. By being language neutral and allowing a Kommander dialog to be extended by using any scripting language Kommander positions itself in a unique position for wide spread adoption. Multiple script languages can be used in a single dialog and applications can be taken over by people using a different language than the original developer and gradually converting and extending it. New widgets and features can be instantly leveraged by all available languages.

We hope that Kommander begins to get the developer support and recognition required to achieve the potential it offers. Our end goal is to make Kommander useful for novice users to extend and merge their applications. At the same time it should become a good prototyping tool. Also it opens the door to the promise of open source in a new way. We know that people can extend our GPL'd programs, but the fact remains very few have the skills. With Kommander those numbers see a huge multiplier! Some applications may be most logical as a Kommander application. We already use it in areas we want to allow extensibility in Quanta Plus.

We hope you enjoy Kommander. Please help us with bug reports and example dialogs, as well as any requests you may have. You can join our user list for help developing Kommander applications at <http://mail.kdewebdev.org/mailman/listinfo/kommander>

Best Regards from the Kommander development team!

Chapter 2

Kommander Basics

Kommander Basics Tamara King and Eric Laffoon

2.1 Concepts

Kommander was originally designed around a simple concept that has proven somewhat revolutionary among visual design tools. Typically these tools allow you to create dialogs and possibly mainwindow interfaces. Of course a mainwindow interface is the main program window which typically has menus, toolbars, statusbar and the application area. Dialogs are child windows which typically don't have menus and are so named because their purpose is to 'have a dialog' or exchange information between you and the main application. The elements on a dialog are called 'widgets' and you hook your program into these widgets. Kommander is different because it is inherently non-programmatic here. It uses the concept of associating text with the widgets on the dialog. Initially this was called 'Associated Text' but now it is called 'Kommander Text'. Widgets on Kommander dialogs can include the content of other widgets by reference and a widget can reference its own content by use of a 'Special' that looks like this, @widgetText. Specials are commands with special meaning in Kommander. So if you created a dialog with two LineEditwidgets and named the first 'FirstName' and the second 'LastName' you could create a button and set its Kommander Text to 'My name is @FirstName @LastName'. You would need to set @widgetText in the first and last name widgets. Remember? We need to tell Kommander to reference the text in them. You could run this from a Konsole and it would output the string for you. So it would reference the first name like so: @FirstName -> get the widget named FirstName(@FirstName) -> @widgetText -> get the contents of the LineEdit widget. So in this case @FirstName returns 'Eric': @FirstName -> @widgetText -> 'Eric'.

That is the simple core of Kommander. What you can do with this is where it gets interesting. First of all it is worth noting that compared to the normal approach of a language based tool Kommander does not need programming

The Kommander Handbook

statements to define these operations. This makes Kommander quick for developers. For end users it's much simpler than learning language constructs. For everyone it means you can focus on your task instead of having your reference material eternally at hand. Initially when people are exposed to a tool like Kommander the first question is 'Where could I find a use for this cool tool?' As it turns out, manipulating strings is used just about anywhere you look.

So what can Kommander do? Here is the list distilled to the base operations. Kommander can:

1. Pass strings to the calling program via stdout.
2. Call executable programs.
3. Use DCOP to interact with KDE programs

If you're not a programmer you may want that in laymans terms. In number one, if you launch Kommander from a console then the console is the calling program. There is a parent child relationship there. Sending a message to console is done with the standard output (stdout) of the child program, so named because there is also error output. This is interesting because some programs, like Quanta Plus, use stdout to receive information from programs they launch. So Kommander dialogs can output their text strings directly into Quanta Plus's editor if they are called from Quanta Plus. This means Kommander dialogs can be useful extentions to programs.

The second case is calling an executable. Any program that runs on your system is an executable. Even a script program is run by the script's interpreter so technically it's executed too. Kommander can run commands just like the console even if you run it from the menu. So for instance if you wanted it to open The GIMP you would have a button derive the string 'gimp' and put it in a special like so: @exec(gimp). Just like that you will see The GIMP open when using this. You could also exec 'ls -l' too but you would only see the output if you were running from a console.

The third case is very interesting indeed. DCOP is short for KDE's *Desktop Communication Protocol* and it is very powerful. Go ahead and run the kdcop program and have a look around. You'll quickly see that any KDE application that is built to standards has things happening in DCOP and the well designed ones have a lot going on. With DCOP you can query information of all sorts as well as set widget values and more. There is a section on using DCOP in this manual. Kommander can send DCOP to any KDE program as well as be controlled by DCOP. In fact you can send DCOP from the command line to any KDE program. So what's the big deal? The deal is, if you want to do any volume of commands you begin to realized that command line DCOP is adequate for short commands, but it can cause delays for instance being called from a loop several hundred times. This is why Kommander has a @dcop special, because this is roughly 1000 times faster. Because Kommander can send and receive DCOP, DCOP can be used to script Kommander. That is why we also have a local DCOP special, @ldcop, that allows you to type a lot less to issue a command.

The Kommander Handbook

Is that all the core concepts in Kommander? No, but it should help you to make sense of how it works so that what is covered does not look like a foreign language to you. There are a few more. Signals and slots are how Kommander handles events. An event in a program basically means ‘something happened’ like a widget was created or had its text changed. These changes ‘emit signals’ and you can connect those signals to a receiving slot which will then do something when the event happens. One use of this in Kommander is the sibling of Kommander Text, ‘Population Text’. Population Text will populate a widget when called. Just like Kommander Text, Population Text can contain text strings or scripts.

That should give you the base concepts to begin using Kommander. We try to keep the number of Specials low and we use DCOP a lot. The idea is that we want to keep the power of Kommander as consistent and streamlined as possible. You will find that you can incorporate any scripting language into Kommander where ever you need to and even multiple scripting languages in a dialog. The rest of the information in this document assumes you are familiar with the concepts and terms presented here. The examples and tutorials are also very useful to understanding what can be done with Kommander.

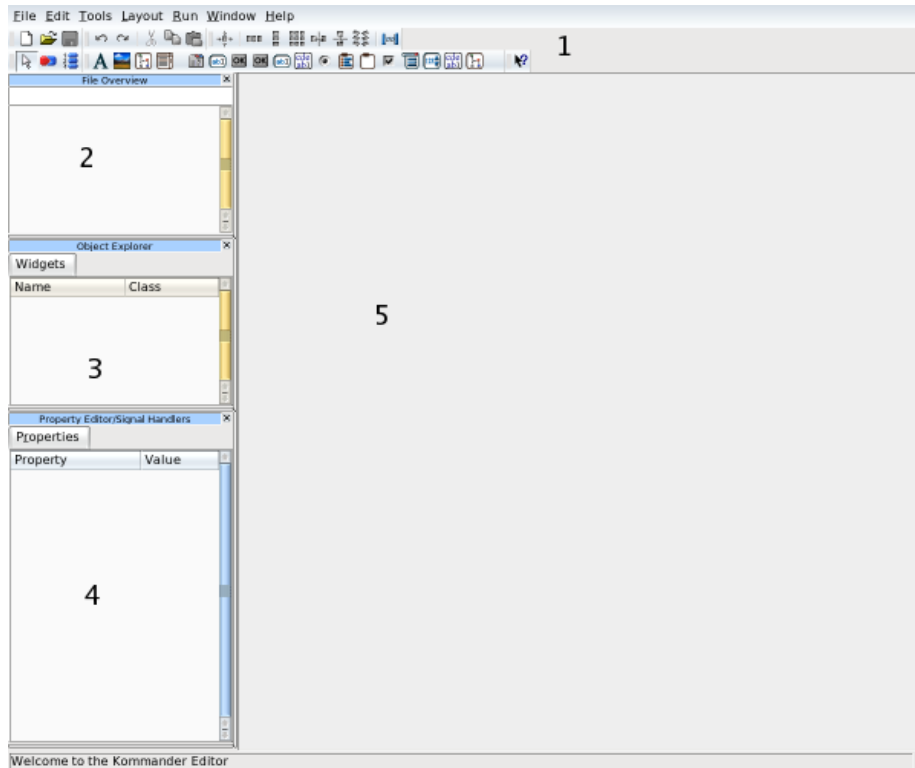
2.2 The Editor

The editor is based on Qt™ Designer, a tool for designing and implementing user interfaces created by Trolltech. We have modified Qt™ Designer in the following ways:

- Its interface is much simpler
- Built in our own widgets
- Added the ability to setup Kommander Text
- Various other superficial changes

For those of you already familiar with using Qt™ Designer, using the Kommander Editor will be trivial.

2.2.1 Main Window



1. Toolbars contain a number of buttons to provide quick access to number of functions.
2. The File Overview displays all of the files. Use the search field to rapidly switch between files.
3. The Object Explorer provides an overview of the relationships between the widgets in a form. It is useful for selecting widgets in a form with a complex layout.
4. The Property Editor is where the behavior and appearance of a widget is changed.
5. The Dialog Editor is where dialogs are created and edited.

2.2.2 The File Menu

File → **New (Ctrl+N)** Creates a new dialog

File → **Open (Ctrl+O)** Search the file system to open an existing dialog

The Kommander Handbook

File → Close Closes the active dialog

File → Save (Ctrl+S) Saves the active dialog

File → Save As Saves the active dialog with another name

File → Save All Saves all open dialogs

File → Recently Opened Files Quick list of the last several files you've opened. This list will change each time you open a file that is not on it with the oldest being bumped off first.

File → Exit Quits Kommander

2.2.3 The Edit Menu

Edit → Undo (Ctrl+Z) Undo the last action performed.

Edit → Redo (Ctrl+Y) Redo the last action undone.

Edit → Cut (Ctrl+X) Cut the current item and place its content on the clipboard.

Edit → Copy (Ctrl+C) Copy the current item to the clipboard.

Edit → Paste (Ctrl+V) Paste the contents of the clipboard at the current cursor position.

Edit → Delete (Ctrl+Z) Delete the current item.

Edit → Select All (Del) Select all of the items in the current dialog.

Edit → Check Accelerators (Alt+R) Verifies that all the accelerators are used only once.

Edit → Script Objects (Alt+S)

Edit → Slots Displays window to edit and create slots and functions.

Edit → Connectors Displays the view and edit connections dialog.

Edit → Form Setting Displays the form setting dialog.

Edit → Preferences Displays the preferences dialog.

2.2.4 The Tools Menu

Tools → Pointer (F2)

Tools → Connect Signal/Slots (F3)

Tools → Tab Order (F3)

Tools → Kommander

Tools → Kommander → TextLabel
Tools → Kommander → PixelmapLabel
Tools → Kommander → ListView
Tools → Kommander → ListBox
Tools → Kommander → SubDialog
Tools → Kommander → TabWidget
Tools → Kommander → LineEdit
Tools → Kommander → ExecButton
Tools → Kommander → CloseButton
Tools → Kommander → FileSelector
Tools → Kommander → TextEdit
Tools → Kommander → RadioButton
Tools → Kommander → ButtonGroup
Tools → Kommander → GroupBox
Tools → Kommander → CheckBox
Tools → Kommander → ComboBox
Tools → Kommander → SpinBoxInt
Tools → Kommander → RichTextEditor
Tools → Kommander → TreeWidget
Tools → Kommander → Unknown
Tools → Kommander → Wizard

Tools → Custom

Tools → Custom → Edit Custom Widgets

2.2.5 The Layout Menu

Layout → Adjust Size (Ctrl+J)

Layout → Lay Out Horizontally (Ctrl+H)

Layout → Lay Out Vertically (Ctrl+L)

Layout → Lay Out in a Grid (Ctrl+G)

Layout → Lay Out Horizontally (in Splitter)

Layout → Lay Out Vertically (in Splitter)

Layout → Break Layout (Ctrl+B)

Layout → Add Spacer

2.2.6 The Run Menu

Run → Run Dialog (Ctrl+R) Runs the current dialog.

2.2.7 The Window Menu

Window → Close (Ctrl+F4) Closes current dialog.

Window → Close All Closes all dialogs.

Window → Next (Ctrl+F6)

Window → Previous (Ctrl+Shift+F6)

Window → Tile

Window → Cascade

Window → Views

Window → Views → File Overview

Window → Views → Object Explorer

Window → Views → Property Editor/Signal Handlers

Window → Views → Line Up

Window → Toolbars

Window → Toolbars → File

Window → Toolbars → Edit

Window → Layout → File

Window → Toolbars → Tools

Window → Toolbars → Kommander

Window → Toolbars → Custom

Window → Toolbars → Help

Window → Toolbars → Line Up

2.2.8 The Help Menu

Help → Kommander Handbook (F1) Invokes the KDE Help system starting at the Kommander help pages. (this document).

Help → What's This? (Shift+F1) Changes the mouse cursor to a combination arrow and question mark. Clicking on items within Kommander will open a help window (if one exists for the particular item) explaining the item's function.

Help → Report Bug... Opens the Bug report dialog where you can report a bug or request a 'wishlist' feature.

Help → About Kommander This will display version and author information.

Help → About KDE This displays the KDE version and other basic information.

2.3 The Executor

The executor, called `kmdr-executor`, runs Kommander scripts. It loads `.kmdr` files and dynamically produces a fully functional dialog.

2.3.1 Executor for Programmers

C++ developers can easily use the `KmdrDialogInstance` class in their C++ programs so that the execution functionality is embedded in the their application obsoleting the need for running the external executor program. For standard dialog the dialog creation overhead is minimal but the creation of the KDE application may delay the dialog for around a second.

2.4 Creating a Dialog

Chapter 3

Reference

Command Reference

3.1 Widgets

The building blocks of a Kommander dialog are the widgets. They are like any other widget in the Qt™ and KDE libraries except they have some extra functionality that allows them to have a ‘text association’. Text is associated with a state of the widget or its populate slot. The number of states depends on the widget. If a widget only has one state, that state is called default.

Widget text blah blah

The dialog has two special states for Kommander text. They are Initiate and Destroy. These are run when the dialog is initialized and when it is destroyed. These protect against what is know as ‘race’ problems on open and mean that you do not require any special procedures on close to manage housekeeping.



ButtonGroup A container to organize buttons into a group. An optional title can be set using the title property. The frame can be adjusted with the lineWidth property. The button group can be set to exclusive by setting the exclusive property to true. This means when one toggle button is clicked all other toggle buttons will be set to off with the exception of radio buttons that are always mutual exclusive even if the group is non-exclusive. Radio buttons can be set to non-exclusive using the radioButtonExclusive property. (I am not so sure that this property actually works.)

ButtonGroup has one state, default.

The widget text for a ButtonGroup is the text associations for each of the buttons in the order they appear in the ButtonGroup.



CheckBox A button that can be checked on and off. It can also be semi-checked if the tristate property is set to true. The label associated with the CheckBox is set in the text property. Setting the checked property will have the CheckBox initially checked.

A CheckBox has 3 states, checked, semichecked, and unchecked.

The widget text for a CheckBox is the value from the text property.



CloseButton A button that when clicked, executes its text association and then closes the dialog. The label on the button is set with the text property. Output from the text association (how to say that) will be echoed to stdout if the writeStdout property is set to true. The button can be the default action for the dialog if the default property is set to true.

CloseButton has one state, default.

There isn't any widget text associated with a CloseButton.



ComboBox ComboBox is a selection widget that combines a button and a pop-up menu. It shows the user's current choice from a list of options in minimal space. Items are added to the list using the edit window. If the editable property is set to true the user can enter arbitrary strings.

ComboBox has one state, default.

The widget text for a ComboBox is the text of the selected item.



ExecButton A button that when clicked executes its text association. The label on the button is set with the text property. Output from the text association (how to say that) will be echoed to stdout if the writeStdout property is set to true. The button can be the default action for the dialog if the default property is set to true.

ExecButton has one state, default.

There isn't widget text associated with ExecButton.



FileChooser The FileChooser widget combines a LineEdit with a button when clicked will present the user with dialog for the selection of files/folders. The file/folder selected is stored in the LineEdit. The type of the FileChooser is set with the selectionType property. Available types are Open, Save, and Directory. Multiple files/folders can be selected if the selectionOpenMultiple property is set to true. A caption for the FileChooser can be set with the selectionCaption property. This is display as the window title of the dialog. If a caption isn't specified, the type of selection will be display in the title. The files displayed in the dialog can be limited using the selectionFilter property.

FileChooser has one state, default.

The widget text for a FileChooser is the text contained in the LineEdit (the file chosen by the user).



GroupBox A container widget that holds other widgets. The frame is adjusted with the `lineWidth` property. A title can be added by setting the `title` property.

GroupBox has one state, default.

The widget text for GroupBox is the text associations of each of the widgets it contains combined. They will be in the order they appear inside of the GroupBox.



LineEdit A LineEdit widget is a one line text editor. It allows the user to enter and modify a single line of text. Initial text for the editor can be set in the `text` property. The widget can be set to read-only with the `readOnly` property. There are 3 modes for the widget, Normal, NoEcho, and Password. The mode is set with the `echoMode` property.

LineEdit has one state, default.

The widget text for LineEdit is the text contained in the editor.



ListBox A ListBox widget provides a list of selectable items. Normally one or no items are selected. This behavior can be changed with the `selectionMode` property. Items are added to the ListBox using the edit window.

A ListBox has only 1 state, default.

The widget text for a ListBox is the items contained in the ListBox. `@selectedWidgetText` will return only the items that are currently selected.



ListView This widget is now Kommander enabled. It is functionally the same as the tree widget so please reference that.



PixmapLabel A simple widget that contains an image or text label. The pixmap to display is set in the `pixmap` property. The text is set in the `text` property. Only one of these properties can be set at the same time (I think, I can't get the editor to set both at the same time). If `scaledContents` is set to true the image will be scaled to fit the size of the widget. The format of the text can be set with the `textFormat` property.

This widget isn't Kommander enabled and doesn't have any states or widget text.



RadioButton A button that can be checked or unchecked, usually used in the ButtonGroup to make an exclusive choice. A label associated with the button can be set in the `text` property. A button can be initialized to checked by setting the `checked` property to true. If all RadioButtons in

a `ButtonGroup` have the `checked` property set to true, then the last button will be the one that is checked.

`RadioButton` has 2 states checked and unchecked.

There is no widget text associated with a `RadioButton`.



RichTextEditor This widget provides a text editor that allows for simple text formatting.

`RichTextEditor` has one state, default.

The widget text for `RichTextEditor` is the text contained in the editor in rich text format. Selected text can be returned with `@selectedWidgetText`.



SpinBoxInt A widget that allows the user to change an integer value by either pressing up and down arrows or entering a value into the box. Minimum and maximum values for the widget can be set with the `minValue` and `maxValue` properties. The `specialValueText` property is used to set a text value that will be displayed instead of the minimum value.

This widget has only one state, default.

The widget text for a `SpinBoxInt` is the currently displayed integer.



SubDialog A button that runs another Kommander dialog when pressed. The dialog to run is set in the `kmdrFile` property. If the default property is set to true, the dialog will be run if enter is pressed while the dialog has focus. I think that you can also use it as a container. I need to play with this some more.

`SubDialog` has one state, default.

The widget text for `SubDialog` is the text association of the dialog it executes.

NOTE

This was meant to contain the dialog, which is deprecated with the new project concept. Should we leave it with `@dialog()` in its Kommander text or get rid of it? As it is it is not right.



TabWidget A widget that provides multiple tabs each may contain other widgets.



TextEdit A simple multi-line text editor.



TextLabel A simple widget that contains a piece of text. This widget lets you set a pixmap too. OK, the editor says that they are both

QLabels. Why do we have 2 widgets that look to be the same thing, but different names? - Scheduled to be fixed in A7.

As of Alpha 6 this widget is partially enabled and can be set using external DCOP calls.



TreeWidget A widget that provides a list in the form of a tree structure. This widget is now fully enabled to add or remove items as of Alpha 6. You can add child items and multi-column data. The current limitation is that you cannot modify columns. To add a child node use '/' as a separator. To add column data use the escaped tab '\t' character between columns.

3.2 Specials and Built-in Global Variables

Specials are functions that are processed by Kommander. You should be aware that until Kommander has a full parser all Kommander specials will be executed first and then the script will be executed. In most cases this is not a problem, but in a few it is.

@dcop (*(appId, , object, , function, , arguments)*) Make a DCOP call. `@dcop('kmail', 'KMailface', 'checkMail()', '')`

@dcopid The DCOP id of the process. (`kmdr-executor-@pid`)

@dialog (*(dialog, [,parameters])*) Launches the specified Kommander dialog. Dialog is sought in dialog directory and in current directory - in that order. This prepends the call to the executor and sets the default directory to the one the Kommander application is in. Parameters can be passed in the usual Unix way or you can pass named parameters like 'variable=value'. You can then find passed parameters in the global pool. `@global(variable)` would return 'value'.

@env (*(environmentVariable)*) Expands to the specified environment variable. `@env(PWD)` expands to `$PWD`. Remember that '\$' is part of the shell and shouldn't be used.

@exec (*(command)*) returns the output of executing the specified command. `@exec(ls -l)`.

@execBegin ... **@execEnd** same as `@exec`, but supports shebang and multiline scripts. This serves for various scripting languages either by decalring them or using a shebang.

- `@execBegin (php)`
- `@execBegin#!/usr/bin/php`

The Kommander Handbook

The first one uses the name of the PHP executable. Kommander searches PATH for php and if it is not found looks to see if it is registered with Kommander in a location outside of your path. If not it tells the user it cannot be found. The second examples uses the classic 'shebang' which can have some benefits and also problems. If you have a beta copy of PHP5, for instance, in /usr/local/bin which would not be found because it would find on in /usr/bin this is useful. If, however, you distribute the dialog to someone who has PHP in /usr/local/bin only it will not be found with the shebang used. So using shebangs is cautioned and using the executable is recommended if you are sharing files.

- @global ((variable))** expands to the value of the specified global variable.
- @null** Returns null. Now that Kommander checks for empty widgetText on execution this will prevent erroneous errors in the case of an unset state on a widget.
- @parentPid** The PID of the parent process.
- @pid** The PID of the process.
- @readSetting ((key,, defaultValue))** reads a value from kommanderrc
- @selectedWidgetText** the selected content in a widget that can show more than one value, like list widgets
- @setGlobal ((variable,, value))** Sets the global variable to the specified value.
- @widgetText** the content of a widget
- @writeSetting ((key,, value))** write value to kommanderrc

3.2.1 Array Function Group

- @Array.values ((array))** Returns an EOL-separated list of all values in the array. Can be used to walk through an array.
- @Array.keys ((array))** Returns an EOL-separated list of all keys in the array.
- @Array.setValue ((array,, key,, value))** Sets a key and value for an element of an array. If no array exists it is created.
- @Array.clear ((array))** Remove all elements from the array.
- @Array.count ((array))** Return number of elements in the array.
- @Array.value ((array,,key))** Return the value associated with the given key.
- @Array.remove ((array,,key))** Remove element with the given key from the array.
- @Array.fromString ((array,,string))** Add all elements in the string to the array. String should have *key\tvalue\n* format."
- @Array.toString ((array,,string))** "Return all elements in the array in *key\tvalue\n* format."

3.2.2 File Function Group

`@File.read((file))` Return content of the given file.

`@File.write((file, string))` Write given string to a file.

`@File.append((file, string))` Append given string to the end of a file.

3.2.3 String Function Group

`@String.length((string))` Return number of chars in the string.

`@String.contains((string, substring))` Check if the string contains given substring.

`@String.find((string))` Return position of a substring in the string, or -1 if it isn't found."

NOTE

This will have an optional integer start position for find next uses in Alpha 6.

`@String.left((string, int))` Return first n chars of the string.

`@String.right((string, int))` Return last n chars of the string.

`@String.mid((string, int start, int end))` Return substring of the string, starting from given position.

`@String.remove((string, substring))` Remove all occurrences of a given substring.

`@String.replace((string, substring find, substring replace))` Replace all occurrences of a given substring with a given replacement.

`@String.upper((string))` Convert the string to uppercase.

`@String.lower((string))` Convert the string to lowercase.

`@String.compare((string, string))` Compare two strings. Return 0 if they are equal, -1 if the first one is lower, 1 if the first one is higher

`@String.isEmpty((string))` Check if string is empty.

`@String.isNumber((string))` Check if string is a valid number.

3.2.4 Built-in Globals

Built-in globals are accessed just like regular global variables with `@global`.

`@global(_KDDIR)` The directory the current dialog is in.

`@global(_NAME)` The name of the dialog

3.3 DCOP Functions

DCOP can be called in several ways in Kommander. First is the console method `dcop kmdr-executor-@pid KommanderIf changeWidgetText myWidget 'new text'`

This assumes you are inside a Kommander file and have access to the special `@pid` which contains the process ID. In fact it is simpler to replace `'kmdr-executor-@pid'` with `@dcopid`. However, you can use this syntax (obviously without the specials) from the command line or any external script to alter the Kommander window.

Because Kommander does not have a full parser in it's Alpha stage, if you want to use the much faster internal DCOP from another application window (console DCOP is very slow) it is more complicated because you must give lots of information, including a prototype of the call. The above call would become:

```
@dcop(@dcopid, KommanderIf, 'enableWidget(QString, bool)', Widget, true)
```

At the time of this writing you should be aware that nesting DCOP calls inside script language structures (like bash) means that you must use console method calls. *If you use internal DCOP all Kommander specials will be executed first and then the script will be executed.*

There is a new simplified way to use DCOP inside Kommander using an object syntax. Let's say you want to change the text in a widget name `@LineEdit1`. It would look like this.

```
@LineEdit1.changeWidgetText(New text)
```

As you can see the new syntax is very easy, as well as consistent visually with function groups. All the DCOP reference here will use the new object syntax listed above. *Please note that if you are referencing a widget using DCOP from another window or another application the first parameter will always be the widget name. All functions are listed here starting with the second parameter.*

3.3.1 DCOP for Global Variables

global(QString variableName) Returns the value of the specified global variable. When a script is run from within a Kommander window any (non-global) variables set in that script will cease to exist after the script completes and therefore will not be available to other script processes or in a new instance of the calling process. The global 'scope' means the variable will exist for any process of the window until that window is closed. You may change these variables at any time with a new call to `@setGlobal`.

setGlobal(QString variableName, QString value) Creates a variable that is global to the window process and assigns the value to it. This value can be retrieved with `global(QString variableName)` or set again.

3.3.2 DCOP for all Widgets

changeWidgetText(QString text) This should be renamed setWidgetText and this name will probably be deprecated. This removes the text displayed in the widget and replaces it with the text supplied.

enableWidget(bool enable) Enables or disables a widget.

associatedText Returns the text associated with the specified widget. This is not the same as the displayed text. It would be '@widgetText' or the text and/or scripting used to arrive at the displayed value.

setAssociatedText(QString text) This sets the Kommander Text default string. This is typically set to '@widgetText' to display what is entered into the widget. It is unlikely you will have much need for this, but if you do it is there. Applies to all widgets that can contain data.

3.3.3 DCOP for ListBox and ComboBox Widgets

addListItem(QString item, int index) Adds an item to a ListBox widget at the specified index. List index starts at zero. To add to the end of the list use -1.

addListItems(QStringList items, int index) This adds a list of strings all at once. The list should be delimited by EOL (\n - newlines). This is handy as you can use bash to derive the list rather easily. For instance, using @exec(ls -l /projects | grep kmdr) for items will give you a directory listing of Kommander files in your projects folder. List index starts at zero. Use -1 to add to the end of the list.

addUniqueItem(QString item) addUniqueItem will add an item to the end of the list only if it is unique.

clearList Removes all items.

removeListItem(int index) Removes the item at the specified index.

item(int index) Returns the text of the item at the specified index.

setCurrentListItem(int index) Set the current (or selected) item to the index specified. Applies to ListBox and ComboBox widgets.

3.3.4 DCOP for CheckBox and RadioButton Widgets

setChecked(QString widgetName, bool checked) Checks/unchecks CheckBox or RadioButton widgets.

3.3.5 DCOP for TabWidget Widgets

setCurrentTab(QString widgetName, int index) Selected the tab by index for TabWidgets. Index starts at 0.

Chapter 4

Extending Kommander

Extending Kommander

4.1 Creating Kommander Widgets

With Kommander you can create new widgets based on non-Kommander widgets fairly easily. The approach is to derive your new Kommander widget class from the Qt™/KDE widget which you wish to integrate with Kommander, and then also from the KommanderWidget class. Overriding methods from this class gives the Kommander widget its functionality.

Most of the code of a Kommander widget is just template code. Therefore, you can use the widgetgenerator.kmdr Kommander dialog to generate most the Kommander widget code for you. All you have to do is fill in the important parts relating to your widget like any state information, widget text etc.

Let's say we want to create a new line edit widget for Kommander, based on the KDE widget KLineEdit. Using the Kommander widget generator dialog, we get something like this in the outputted header file:

Chapter 5

Tutorials

Tutorials

5.1 Using Editor

This might go in the editor section.

5.2 Globals

Shows using global and setGlobal DCOP calls to provide global variables for script

5.3 DCOP

Shows how to use both local and external DCOP calls to communicate with external application.

5.4 Slots

Shows how to use connections/slot to handle events. Both population and standard slots are used.

5.5 Settings

Shows how to use `@readSetting` and `@writeSetting` functions to write/restore widget content. Also, show how to use `populate()` slot to initialize widget content.

5.6 Append

Shows how you can append text to `TextEdit` and how you can use it to display formatted text.

5.7 Command Line

Shows how you can pass parameters to Kommander dialog via command line. Also, shows how to change list content and button text.

5.8 Initialize

Shows how you use 'initialization' to 'destroy' scripts of main dialog to initialize and store some settings.

Chapter 6

Questions and Answers

Questions and Answers This document may have been updated since your installation. You can find the latest version at <http://docs.kde.org/development/en/kdewebdev/> .

Chapter 7

Credits and License

Credits and License Tamara King THE KOMMANDER DEVELOPMENT TEAM

Britton, Marc consume@optusnet.com.au Developer and documentation

King, Tamara tik@acm.org Documentation

Laffoon, Eric sequitur@kde.org Project manager and documentation

Mantia, András amantia@kde.org Developer

Rudolf, Michal mrudolf@kdewebdev.org Developer

Kommander © 2004 Kommander Development Team.

Kommander User Manual © 2004 Kommander Development Team.

This documentation is licensed under the terms of the [GNU Free Documentation License](#).

This program is licensed under the terms of the [GNU General Public License](#).

Appendix A

Installation

A.1 How to obtain Kommander

Kommander is part of the KDE project <http://www.kde.org/> .

Kommander can be found in the kdewebdev package on <ftp://ftp.kde.org/pub/kde/> , the main FTP site of the KDE project.

A.2 Requirements

A.3 Compilation and Installation

In order to compile and install Kommander on your system, type the following in the base directory of the Kommander distribution:

```
% ./configure
% make
% make install
```

Since Kommander uses **autoconf** and **automake** you should have no trouble compiling it. Should you run into problems please report them to the KDE mailing lists.

A.4 Configuration

Appendix B

Glossary

Keywords

Text Association A piece of text that is associated or bound to a widget's particular state.

Widget Text Text associated to a widget. This is represented in Kommander with the special `@widgetText`. The widget text varies depending on the widget.