Aida-Commander: Command Reference

History

Dare	Author	Rev.	Type	Description
2012-02-13	Thomas Grewe	1.00.06	cont.	config: getCurrentUIClassName, setIcon, addIcon added log: systemEnvironment added
2010-04-27	Thomas Grewe	1.00.05	cont.	Translation into english
2010-03-29	Thomas Grewe	1.00.04	inh.	core: listen, listenfor hinzu
2007-11-19	Thomas Grewe	1.00.03	inh.	config: setConsoleAutoFocus, settooltipdismissdelay, settooltipinitialdelay und settooltipsenabled hinzu
2004-03-30	Thomas Grewe	1.00.02	inh.	config: getUIName, setStartBatch hinzu, Erläuterungen zu hide erweitert log: buffer, setWrapColumn hinzu
2003-06-06	Thomas Grewe	1.00.01	inh.	config: iconify/deiconify, hide/show, setStatusText hinzu core: kill, killOnLostConnection, gc, heapInfo,pause, close(force-Option) hinzu debug: itemWindow, set/getRequestInterval, set/getIdleUpdateInterval hinzu log: Neue Kommandoklasse tools: getEvaluatedString hinzu Änderungsindex hinzu
2001-05-28	Hans-Joachim Pflug	1.00.00		Erstfassung

Contents

Format of commands

Commands consists of a class name, a point and a method name, e.g. *core.about*. Class name and method name can alternatively also be separated by a comma. The command name is followed by the parameters. As separators "," ":" and "=" are allowed. Parameters containing one of the separators, double or single quotes or an escape sequence, have to be put into quotes (see also Escape sequences). Commands with parameters of type *File* might contain System environment variables. Commands, which are triggered directly by the Commander (e.g. by a Menu Item or a Button in a Toolbar) might also use Aida-Commander-Tags. Correct are for example:

```
config.changeSize:20,50
tools.echo=Hello
tools.echo,"Hello, world"

tools.echo,"\"Hello, world\""
```

Parameter types

Parameters can have the following types:

byte	8 bit, signed
short	16 bit, signed
int	32 bit, signed
long	64 bit, signed
float	32 bit floating point
double	64 bit floating point
Array	Byte-Array
String	String
File	File names. *File* is different from *String* by the fact, that in a *File* parameter ":" is not interpreted as a separator. Also in a *File* parameter system environment variables are interpreted.
Object	This parameter type can be used to reference objects which have been defined.
RestOfLine	The rest of the command line will be taken without special treatment of further separators.

Most parameters can be required or omitted. In the following reference required parameters are written in squared brackets, parameters which can be omitted are written in curly braces. The command

config.load,[File f],{String Options}

might, e.g. by omitting the last parameter, written like this:

```
config.load,D:\util\menu.cmdr-cfg
```

Escape sequences for String parameters

String or File parameter might contain escape sequences, if the parameter is enclosed in single or double quotes. The following escape sequences are supported:

Sequenz	Erläuterung
\b	backspace
\t	horizontal tab
\n	line feed
\f	form feed
\r	carriage return
\"	double quote
\'	single quote
\\	backslash
\000 bis \377	ASCII-characters according to their octal value

System environment variables

In parameters of type *File* system environment variables are evaluated. The syntax $ (*Environment variable*) has to be used.

Aida-Commander-Tags

The following tags are recognized in all internal and Text commands:

Tag	Description
<bs>	backspace
<ht>	horizontal tab
<lf>	line feed
<cr>	carriage return
<fo>	form feed
<gt>	greater-than sign (>)
<lt>	less-than sign (<)
<qu>	quotes (")
<fi>	file: shows a Fileselect-Box , and uses the selected file name at this place.
<di>	directory: shows a Fileselect-Box, and uses the selected directory at this place.
<ip>	input: shows a Dialog box and uses the entered text as parameter value.
<wo>	word: Takes over the last word before the cursor from the I/O-Window. The word has to be in the same line as the cursor.
	line: Takes over the contents of the line with the cursor from the start till the position of the cursor.
<lc>	local: inserts the IP address of the commander host.
<00>	The corresponding ASCII character.

- <ff>	

The following tags are recognized in specific commands only:

Tag	Kommando	Erläuterung
<po>	Many config commands	Point: Replaces the point in the reference of Buttons, Menu Items and windows.

Parameter transfer from the POOL runtime

The POOL command *IDECommand* can be used to call internal commands from the POOL runtime. Type conversion will be done, if possible, during the transfer. It will be checked if a type conversion is possible without loss of data and an error message will be returned if necessary.
Numeric values can be converted without problems into Strings, also Strings into numeric types, if the string contains an allowed numeric representation. Strings can be converted into parameters of type *File*.

Return values

Some internal commands have return values. These return values can not be interpreted at the moment internally by the Commander. However this is possible if the inernal command is called from the POOL runtime by means of the IDECommand function, which returns the answer as output of type *String*.

List of standard internal commands of the Commander

Commands of the class *config*

This class contains commands for the configuration of the Commander window (e.g. size, look, menus, toolbars..

---

config.activate, [String name]

Activates a Button, Button Row, Toolbar, Menu Item, Menu or whole Menubar. Also the corresponding graphics objects will be updated. A deactivated Menu Item can be activated again by *config.activate*.

name: The name of the graphics object. For naming conventions see config.removeItem.

---

config.addButton, [String parent], [String name], [int active], [int mode], {File filename}, {Array icon}, {String text}, [int width], [String command], {String tooltip}

Adds a Button into a Button Row.

parent: The name of the Button Row into which the Button should be added. For naming conventions see config.removeItem.

name: The name of the new Button. Under this name the Button will be addressed later.

active: A flag which determines, if the Button should be inactive (greyed out) in the beginning. With *active* = 0 the Button is inactive at the beginning, all other values are interpreted as active. This state can be changed afterwards by config.activate.

mode: The mode of the Button affects, if an icon, text or both is displayed. Two bits are used:

Bit 0: If Bit 0 is set, an icon is displayed.
Bit 1: If Bit 1 is set, text will be displayed, if necessary at the right side of the icon.

filename: If an icon should be displayed, the name of a file can be specified, which should be read by the Commander. Allowed file formats are XBM, GIF, and JPEG. If data is provided with the *icon* parameter, the file contents is not read, only the filename will be transferred into the related variable.

icon: The icon file data can also be transferred from the POOL runtime in a byte array. The byte array has to contain the data in XBM, GIF, or JPEG format.

text: The text which can be displayed instead of or additionally to the icon.

width: The width of the icon. A width of 1 leads to a quadratic icon area, higher numbers lead to accordingly wider icon areas.

command: The command string, which will be executed by pressing the Button. The command string may consist of a sequence of internal and text commands.

tooltip: The tool tip, which will be displayed, if the cursor is moved into the area of the Button.

---

config.addButtonRow, [String parent], [String name], [int active]
Inserts a Button Row into a Toolbar.

parent: The name of the Toolbar, into which the Button Row should be inserted. For naming conventions see config.removeItem.

name: The name of the new Button Bar. Under this name the Button Bar will be addressed later.

active: A flag which determines, if the Button should visible in the beginning. With *active* = 0 the Button is inactive at the beginning, all other values are interpreted as active.

---

config.addIcon, [String name]
Adds an Icon to the list of icons for use in the main Window. This can be used to provide alternative icons in different resolutions to be used by the operating system in different scenarios, e.g. window title, task bar, task switching,

name: The file name of the icon file. Image formats known to work are gif, png. The ico format is not supported.

---

config.addMenu, [String parent], [String text], {String mnemonic}

Adds a Menu or Submenu.

parent: The object into which the Menu should be inserted. If *parent* has the name *menu,* the new Menu will be inserted as new menu column into the main menu bar. If *parent* is a Menu or Submenu the new Menu will be inserted as Submenu into *parent*. It is possible to create nested Submenus. Under all tested user interfaces Submenus open up to the right side. For naming conventions of *parent* see config.removeItem. After adding or removing of menu elements the menu structure has to be updated by config.activate.menu.

text: The text of the Menu or Submenu.

mnemonic: The shortcut key to reach the Menu via the keyboard. Only the first character of the string is evaluated. Mnemonics are underlined in Menus. The Menu can be opened by pressing *ALT+Mnemonic.*

---

config.addMenuItem, [String parent], [String text], {String mnemonic}, [int shortcutActive], {String keycode}, {String modifiers}, [int active], [String command]

Adds a Menu Item into a Menu or Submenu.

parent: The name of the Menu or Submenu to which the Menu Item should be added. For naming conventions see config.removeItem.

text: The text of the Menu Item.

mnemonic: The shortcut key to reach the Menu Item via the keyboard. Only the first character of the string is evaluated. Mnemonics are underlined in Menu Items. The Menu can be reached by pressing *ALT+Mnemonic* if the containing Menu or Submenu is already opened.

shortcutActive: A flag which specifies, if a shortcut should be active. A shortcut is a key combination to trigger a Menu Item directly. In contrast to a Mnemonic the containing Menu or Submenu needs not to be opened.

keyCode: The keycode of the shortcut. **Attention:** The keycode of a key normally is corresponding to the US layout of the keyboard. Below two methods are descibed to get the keycodes of a key.

1. In the Button editor of the Commander click into the field *shortcut* and press a single key (without Shift, Ctrl etc.). The keycode of the key is displayed in the entry window.

2. A list of all available key codes can be found in the Java documentation under *..\docs\api\java\awt\event\KeyEvent.html* . All statical variables with a *VK_* prefix mentioned there can be used. The prefix *VK_* has to be omitted when entering the key code. Only parts of the key codes described in the Java documentation are available with a standard keyboard.

modifiers: Modifying keys for the shortcut, like *Shift*, *Ctrl,* or *Alt*. Again two methods can be used to get the description of the modifiers.

1. In the Button editor of the Commander click into the field *shortcut* and press the complete key combination. The modifiers are the parts in front of the minus sign displayed in the entry window.

2. A list of possible modifiers can be found in the Java documentation under *..\docs\api\java\awt\event\InputEvent.html*. All static variables with the suffix *_MASK* may be used as modifiers. The the suffix *_MASK* has to be omitted when specifying the modifier. Multiple modifiers can be combined by '+' , e.g. *Shift + Alt*. Only some of the modifiers described in the Java documentation can be generated by a standard keyboard.
   In the Java 1.2 documentation the following modifiers are listed:
   Modifiers reachable via a standard keyboard: *Shift, Ctrl, Alt.*
   Modifiers not reachable via a standard keyboard: *Alt_Graph* (doesn't work with 'Alt Gr' key, instead 'Ctrl + Alt' has to be entered), *Button1, Button2, Button3, Meta*.

active: A flag which determines, if the Menu Item should be inactive (greyed out) in the beginning. With *active* = 0 the Menu Item is inactive at the beginning, all other values are interpreted as active. This state can be changed afterwards by config.activate.

command: The command string, which will be executed by selecting the Menu Item. The command string may consist of a sequence of internal and text commands.

---

config.addSeparator, [String parent]
Adds a Separator to the Menu Items of the specified Menu.

parent: Name of the Menu or Submenu to which the Separator should be added. For naming conventions see config.removeItem.

---

config.addToolbar, [String title]
Creates a new Toolbar in the Commander Window. At first the Toolbar is invisible and has to be made visible by config.activate.

title: The title of the new toolbar.

---

config.changeSize, [int width], [int height]
Changes the size of the console to the POOL runtime in the I/O-Window. During the execution of this command, there must be no connection to the POOL runtime established.

width: The new number of columns of the console in the I/O-Window.

height: The new number of lines of the console in the I/O-Window.

---

config.changeSizeBox
Changes the size of the console to the POOL runtime in the I/O-Windows. In a dialog window the new column and row numbers have to be specified. During the execution of this command, no connection to a POOL runtime might be established.

---

config.configuration
Shows the dialog window for entering of start batches. This dialog window is described in more detail in the chapter *Commander-Start-Batches* of *Commander Anleitung.*

---

config.deactivate, [String name]

Deactivates a Button, Button Row, Toolbar, Menu Item, Menu or whole Menubar. Also the corresponding graphics objects will be updated. Buttons, Menus, and Menu Items are not selectable any more and will be greyed out. Toolbars, Button Rows, and Menubars will become invisible by deactivation. A deactivated graphics object can be activated again by *config.activate*.

name: The name of the graphics object. Names are structured as follows:
Buttons: [Name of the Toolbar]\[Name of the Button Row]\[Name of the Button]
Menu Items: [Name of the Menu]\{Name of the Submenu}\...\[Name of the Menu Item

---

config.deiconify

Switches a Commander window, which had been previously iconified by config.iconifiy back to the normal state.

---

config.editMenu

Shows the Menu editor window. This dialog window is described in more detail in the chapter *Menü editieren* of *Commander Anleitung.*

---

config.font, [String name], [int size]

Specifies a new font for the I/O-Windows. Before setting a new font, a connection to the POOL runtime must be closed.

name: The name of the font.This might be a font specific for the operating system like e.g. the Windows font "Courier". Under other operating systems another font will than be used by the Commander. On the other hand, Java offers some reserved font names which map to different fonts depending on the particular operating system, like e.g. "Monospaced". *Important: Not all fonts are useful for the I/O-Window. At first the font should be not proportional, also all needed characters have to be available. Under the Windows OS from all tested fonts only two remained:* ***Courier New*** *und* ***Lucida Console****. The font names "monospaced.bold" und "dialoginput.bold" reserved by Java on Windows OS map to "Courier New" and are useable as well.*

size: The size of the font.

---

config.fontBox

Specifies a new font for the I/O-Windows. The font name and font size can be chosen in a dialog box. The Commander selects only non proportional fonts to choose from, so not all fonts available in the system are displayed. For more info about useful fonts see under config.font.

---

config.getCurrentUIName

Read out the UI (User Interface) currently used by the Commander(e.g. Metal, Motif, Windows).

---

config.getCurrentUIClassName

Read out the UI class name(User Interface) currently used by the Commander(e.g. Metal, Motif, Windows, WindowsClassic).

---

config.hide

Sets the whole Commander window to be invisible. Under Windows OS this leads to the fact, that also no more icon is displayed in the task bar and no more entry can be seen in the task survey of the task manager. The Commander window can be made visible again by config.show .
When loosing a connection to the POOL runtime while the Commander window is invisible, the Java Virtual Machine will be terminated immediately similar to the command core_killOnLostConnection

---

config.iconify

Iconifies the Commander window. By config.deiconifiy the Commander window can be brought back to normal mode.

---

config.load, {File name}, {String options}

Load a new Commander configuration. The configuration consists of:

the contents of the Menu Bar,
a variable number of Toolbars,
Start-Batches, which will be executed when starting the Commander (and e.g. determine size and position of the displayed windows),
size of the I/O-Window,
font of the I/O-Window,
the "Look&Feel",
the password.

name: The file name of the configuration file. If no filename is specified, a Fileselect-Box will be shown.

options: options. Possible are:

m - Load Menus
b - Load Buttons (Toolbar)

Start-Batches, screen configurations, and password will always be loaded and replace the old settings.

Important: If under options neither b nor m are specified, the Menus and the Buttons will be loaded. Without any option all configuration settings will be loaded and replace the former settings.

---

config.lockKey[String key]

Locks or unlocks the keyboard for entering into the I/O-Window. Keyboard actions into other windows, shortcuts and mnemonics are not concerned by this.

key: Two keywords are possible for this parameter:

all: Locks the whole keyboard.
none: Unlocks the whole keyboard.

---

config.newToolbar, {String name}

Creates a new Toolbar. At first the Toolbar contains a single empty Button.

name: The name of the new Toolbar. If the parameter is omitted, the Toolbar has no name at first. Buttons contained in nameless Toolbars can not be addressed directly. In the edit mode (see *Commander Anleitung*) can be renamed interactively.

---

config.presetBounds, [int x], [int y], [int width], [int height]

Specifies the size and position of the next window which will be initialized. The command was mainly used for debugging windows used together with the BSK diagnosis and is still there for compatibility. It works without contraints for:

symbol selection window (used together with BSK diagnosis),
macro selection window (used together with BSK diagnosis),
breakpoint window (used together with BSK diagnosis),
and macro listing window (used together with BSK diagnosis).

For symbol watch windows (used together with BSK diagnosis), the command *config.presetBounds* works if debug.watchSymbols was called with *mode=file* or

mode=names, but not with *mode=box*.
The size of Toolbar windows is determined automatically, there position is specified by config.presetNamedLocation. Size and position of the I/O-Window can be set by config.setConsoleBounds. Finally there is the command config.setMainBounds, which sets size and position of the main Commander window on the screen.

Example: A new macro listing should have the size 400x200 Pixels and should appear at the position (500,0) on the screen The needed commands are

```
config.presetBounds, 500, 0, 400, 200
debug.showMacro, Test-Makro
```

The set and preset commands are mainly designated to create a certain work environment on the Commander desktop after start. It isn't necessary to enter these commands manually, because it is possible to import these settings automatically into the Start Batch which contains all necessary set and preset commands in the Start-Batch-Dialog window (see config.configuration). Later window position and size can be changed by config.setPos, config.setSize, and config.setBounds.

x: The x-Position of the upper left corner in the main Commander window.

y: The y-Position of the upper left corner in the main Commander window.

width: The width of the new window in pixels..

height: The height of the new window in pixels..

---

config.presetNamedLocation, [String name], [int x], [int y]
This command should be used mainly in Start Batches. The position of a window with the name *name*, which will be created later can be specified in advance, so it appears at the desired position right away.

This command currently works with Toolbars only. See also config.presetBounds.

name: The name of the window, meaning the window title.

x: The x-Position in the main Commander window.

y: The x-Position in the main Commander window.

---

config.removeItem, [String name]
Deletes a graphical control object. This can be a whole Menubar structure, a Menu, Submenu, Menu Item, Toolbar, Button Row, Button, or Window.

name: The fully qualified name of the graphical control objects, which consists of the name of the elements:

Element	Fully qualified name
Complete Menubar structure	menu
Menu	menu.[Menu]
Submenu	menu.[Menu].[Submenu]

	or for nested Submenus menu.[Menu].{Submenu1}. ... .[SubmenuN]
Menu Item	menu.[Menu].{Submenu1}. ... .[Menuitem]
Toolbar	desktop.[Toolbar]
Button Row	desktop.[Toolbar].[Buttonrow]
Button	desktop.[Toolbar].[Buttonrow].[Button]
Window	desktop.[Window]

The names of the elements are defined by:

Element	Definition of Name
Menu	Title of the menu column
Submenu	Name of the submenu
Menu Item	Name ofthe Menu Item
Toolbar	Title of the Toolbar
Button Row	Name of the Button Row (left entry field in the Toolbar-Editor)
Button	Name of the Button (upper entry field in the Button-Editor, not identical with the text displayed on the Button)
Window	Title of the window

If some of the names contains themselves already contains a point, this has to be described by the Tag <PO> .

---

config.save, {File name}, {String options}

Saves the actual configuration. See also *config.load*

name: The file name under which the actual configuration should be saved. If this parameter is omitted, a Fileselect-Box will be displayed. An asterisk (*) as parameter means that the configuration will be saved under the name of the last loaded configuration.

options: Save options. As options allowed is a combination of the following characters:

b - Saves all Buttons (Toolbars)

m - Saves all Menus

s - Saves all other screen settings (Size of I/O-Window, font, Look&Feel, Start-Batches)

Passwords are always saved.

Important: If no option is specified, the Commander automatically chooses the option "bms", which means all is saved.

---

config.setBounds, [String name], [int x], [int y], [int width], [int height]

Sets size and position of a Window.

name: The name of the window. The fully qualified name is: *desktop.[Title of the Window]*.

x: The new x-Position in the main Commander window.

y: The new y-Position in the main Commander window.

width: The width of the window in pixels.

height. The height of the window in pixels.

---

config.setConsoleAutoFocus, [int modus]

Sets the behaviour of the focus system when requesting user input in the Commander I/O-Window .

modus: With a value of **0,** upon activation of an input request the focus is moved to the I/O-Window from inside the Commander window. With a value of **1,** upon activation of an input request the focus is moved to the I/O-Window from the whole Java Virtual Machine (JVM), which means also from Visual Objects which run from the same JVM. With introduction of this command in Commander Version 1.01.015 (AIDA Version 1.02.022) the default is the behaviour according to mode 0, in prior versions only the behaviour according to mode 1 was available.

---

config.setConsoleBounds, [int x], [int y], [int width], [int height]

Sets the position and size of the I/O-Window. This command sets the size of the area used for the display of the commander I/O-window in pixels, not the size of the area needed for displaying the console to the POOL runtime. Both sizes need not to be the the same. If e.g. the area needed for displaying the console to the POOL runtime is bigger, Scrollbars are used in the I/O-Window

x: The new x-Position in the main Commander window..

y: The new y-Position in the main Commander window..

width: The width of the I/O-Window in pixels.

height: The height of the I/O-Window in pixels.

---

config.setIcon, [String name]

Sets an icon to be used in the main window. Alternative icons in different resolutions to be used by the operating system in different scenarios, e.g. window title, task bar, task switching can than be added by *config.AddIcon*.

name: The file name of the icon file. Image formats known to work are gif, png. The ico format is not supported.

---

config.setMainBounds, [int x], [int y], [int width], [int height]

Sets size and position of the main Commander window on the screen.

x: The new x-Position.

y: The new y-Position.

width: The new window width.

height: The new window height.

---

config.setPos, [String name], [int x], [int y]

Sets the position of a window in the main Commander window.

name: The name of the window. The fully qualified name is: *desktop.[title of the window]*.

x: The new x-Position in the main Commander window..

y: The new y-Position in the main Commander window..

---

config.setSize, [String name], [int width], [int height]

Sets the size of a window in the main Commander window.

name: The name of the window. The fully qualified name is: *desktop.[title of the window]*.

width: The width of the window in pixels.

height. The height of the window in pixels.

---

config.setStartBatch, [String batch]

Sets the commands which will be executed prior to connection to the POOL runtime, when the Commander is started with a configuration file.

batch: The commands which will be executed prior to connection with the POOL runtime, when the related configuration is started.

---

config.setStatusText, {String text}

Sets the text in the status line (bottom left) of the Commander window. With the next status change of the system, the standard status message from the system will be displayed..

text: The text, which should be displayed in the status line. An empty string means, that the standard status message corresponding to the system state will be displayed.

---

config.settooltipdismissdelay, [int time]

Sets the maximal time span, for which a Tooltip will be displayed. This value is valid globally for the whole Java Virtual Machine (JVM), which means for both Commander and Visual Objects, if they are running on the same JVM (Standard, for Commander and Visual objects running on the same computer). If the cursor leaves the area of the object, the displaying of the Tooltip will be stopped earlier.

time: The maximal time span in Milliseconds, for which the Tooltip will be displayed.

---

config.settooltipinitialdelay, [int time]

Sets the time span after which a Tooltip will be displayed. This value is valid globally for the whole Java Virtual Machine (JVM), which means for both Commander and Visual Objects, if they are running on the same JVM (Standard, for Commander and Visual objects running on the same computer).

time: The time span after which the displaying of a Tooltip begins. Starting time is when the cursor stops moving over the object.

---

config.settooltipsenabled, [int modus]

Switching on or off of all the Tooltips. This value is valid globally for the whole Java Virtual Machine (JVM), which means for both Commander and Visual Objects, if they are running on the same JVM (Standard, for Commander and Visual objects running on the same computer).

modus: With a value of **0** the Tooltips will be switched off, with a value of **1** the Tooltips are switched on.

---

config.show

Switches the Commander window to be visible.

Commands of class *core*

This class contains basic commands concerning the state of the Java Virtual Machine, the state of the Commander, and for the communication of the Commander with other programs.

---

core.about

Shows a window containing the version numbers.

---

core.addClass, [String objectName], [String className]

Adds to the Commander a new object of a Java class. This can be either a user defined class, or a Menu Structure or an input area of the classes CommanderMenu or UserInput. The methods of the new object can be addressed by internal commands of the commander. As an example an object of class *AidaTest* should be loaded and afterwards the parameterless method *debug* should be executed. In the example the object gets the name *test*.

```
core.addClass, test, AidaTest
test.debug
```

For the instantiation of the new object the standard constructor will be used, thus ignoring explicit constructurs. If an initializiation of the object is needed, this has to be done in an additional method.

objectName: The name of the new object. With this name the object can be addressed later. The first character of the object name must be an alphabetic character.

className: The name of the class of the new object.

---

core.close, {String option}

Closes the connection between POOL runtime and Commander. For security reasons without the string option a confirmation dialog is shown, when a connection is established.

option: If the keyword "force" is supplied, no confirmation dialog will be shown, even when a connection is established.

---

core.closeNexit

Closes the connection between POOL runtime and Commander and ends the Commander.

---

core.connect, [String dest], {user}, {domain}, {password}, {File workDir}, {String parameters}

Tries to connect to a running POOL runtime. For the BSKD2000 commander it connects to a BSK diagosis program. Some of the parameters are useful just for the latter.

user The user name (on Windows OS the window user name). If a connection should be made with the local host, *user* can be replaced by '*'.

domain: The domain name for the user name.

password: The password associated to the user name. If just a local connection should be established and the user name '*' is specified, a password is not necessary.

dest: This identifies the destination host. This might be a host name like *localhost* or an IP address like *127.0.0.1*.

workDir: A working directory into which the BSK diagnose program changes after its start.

parameters: Parameter values for the BSK diagnosis program. As parameter values all options can be used, which can be used while starting the BSK diagnosis (see in the help file for BSK diagnosis chapter 2.1).

---

core.gc

Triggers the garbage collector of the Java Virtual Machine. Normally this is done automatically if needed. E.g. when searching for "memory leaks" (Object Retention), it might be useful together with the command core.heapInfo to trigger the garbage collection manually.

---

core.heapInfo

Returns information about the currently free, the requested, and maximally requestable Java heap memory.

Return value: +[core.heapInfo]0,[free],[total],[max]

free: The currently free Java heap.

total: The currently requested Java heap.

max: The Java heap, which maximally can be requested. Because of Java Bug ID #4686462 a value, which is to high, will be returned).

---

core.kill

Terminates the Java Virtual Machine immediately. With this hard exit, e.g. no history data will be saved and no configuration data will be saved.

---

core.killOnLostConnection,[int mode]

Specifies if the Java Virtual Machine will be terminated, if a connection to a POOL runtime is lost.

mode: With a value of **0** the Java Virtual Machine will not be terminated, if the connection to a POOL runtime is lost. With a value of **1** upon loss of connection to the POOL runtime the Java Virtual Machine will be terminated like for the command core.kill.

---

core.listen

Starts a server thread, which waits for an incoming connection from a POOL runtime. The number of the port, which can be used for connecting is shown in the status field (lower left corner of the main Commander window).

---

core.listenfor,[String winMode], [File exeFile], {File directory}, {String parameters}

Starts a server thread, which waits for an incoming connection from a POOL runtime, as the command core.listen. Then similar to the command core.run an external program is started. This program is expected to connect as client to the server port. Because of this, the additional options –client und –port:<portno> are inserted into the parameters. This currently works only under Windows operating Systems and PI (the POOL runtime) as the program to start. Internally this function uses the Windows function *ShellExecute*.

winMode specifies window modes for the new application window. Possible values are the numbers 0 - 10, and several key words:

No.	key word	Denotation
0	none	The application gets no window.
2	min	The application starts minimized with an icon in the task bar.
3	max	The application starts with a maximized window.
4	back	The application starts with its window at normal size, but the window doesn't get the focus on start.
5	normal, front	The application starts with its window at normal size, and the window gets the focus on start.

Using other numbers than those listed makes no sense. A complete listing of all modes can be found in the Windows documentation under the keyword *ShowWindow*.

exeFile: The file which should be started. *exeFile* either itself bust be an executable file or it must be assigned to an executable file.

directory: The working directory of the new application.

parameters: Command line parameters for calling the new application.

---

core.pause,[long time]

Pauses the execution of this thread for the duration specified in the parameter *time*. The pause can be ended previoulsly by an InterruptedEXception thrown in Java (e.g. in the AWT-EventQueue, which is responsible for the execution of commands triggered by using Terminal Buttons or Menu Items, this command doesn't lead to a pause of the specified duration).

time: The waiting time in milliseconds. Attention!: With a value of **0** it will be waited indefinitely, the pause can be ended only by an InterruptedException.

---

core.run,[String winMode], [File exeFile], {File directory}, {String parameters}

Starts an external program. This currently works only under Windows operating systems. Internally this function uses the Windows function *ShellExecute.* It is e.g. possible to start Batch files, also it is possible to 'start' files, which are connected with an application.

winMode specifies window modes for the new application window. Possible values are the numbers 0 - 10, and several key words:

No.	key word	Denotation
0	none	The application gets no window.
2	min	The application starts minimized with an icon in the task bar.
3	max	The application starts with a maximized window.
4	back	The application starts with its window at normal size, but the window doesn't get the focus on start.
5	normal, front	The application starts with its window at normal size, and the window gets the focus on start.

Using other numbers than those listed makes no sense. A complete listing of all modes can be found in the Windows documentation under the keyword *ShowWindow*.

exeFile: The file which should be started. *exeFile* either itself bust be an executable file or it must be assigned to an executable file.

directory: The working directory of the new application.

parameters: Command line parameters for calling the new application.

---

core.textEvent,[String text]

Transfers the parameter *text* to the POOL runtime. Interpretation of the parameter is under the responnibility of the POOL runtime.

text: will be transferred as string to the POOL runtime.

Commands of class *debug*

This class contains commands for debugging of a running POOL program.

---

debug.getIdleUpdateInterval

Gets the time interval, after which the variable values displayed in an Item Window will be requested internally in the Commander and be redrawn. See debug.setIdleUpdateInterval

Return value: +[debug.getIdleUpdateInterval]0,[interval]

interval: The time interval from last displaying a variable value to a redisplay and automatic invalidation of the value internally in the Commander in milliseconds.

---

debug.getRequestInterval

Gets the duration, how long after an answer to a variable value from the POOL runtime will be waited until the content of the variable will be requested again. After expiration of this time, for a value request internally in the Commander, the Commander will request the variable value again from the POOL runtime. See also debug.setRequestInterval

Return value: +[debug.getRequestInterval]0,[interval]

interval: The time, a variable value is assumed to be valid in milliseconds.

---

debug.itemWindow

Opens a new window for the inspection of objects of the POOL runtime.

debug.setIdleUpdateInterval, [long interval]
Sets the time interval, after which the variable values displayed in an Item Window will be requested internally in the Commander and be redrawn. If, from the internal request in the Commander a request to the POOL runtime is generated, depends on the request interval set by debug.setRequestInterval.

interval: The time interval from last displaying a variable value to a redisplay and automatic invalidation of the value internally in the Commander in milliseconds.

debug.setRequestInterval, [long interval]
Sets the duration, how long after an answer to a variable value from the POOL runtime will be waited until the content of the variable will be requested again. After expiration of this time, for a value request internally in the Commander, the Commander will request the variable value again from the POOL runtime.

interval: The time, a variable value is assumed to be valid in milliseconds.

Commands of class *history*

This class contains commands concerning the command line stacks.

history.clear, [int id]
Clears command line stack No. *id*.

id: The number of the command line stack, which should be cleared.

history.clearAll
Clears all command line stacks.

history.load, {File name}
Loads a history file and writes the entries into the command line stacks. All former entries will be deleted.

name: The name of the history file. If no file name is provided a Fileselect-Box will be displayed. The current history name, under which the command line stacks will be saved upon ending the Commander, will be changed accordingly (see history.setName).

history.save, {File name}
Savest he current command line stacks in a history file.

name: The file name under which the command line stack should be saved. If no file name is provided a Fileselect-Box will be displayed. The current history name, under which the command line stacks will be saved upon ending the Commander, will be changed accordingly (see history.setName).

history.setMaxEntries, [int start], [Array values]
Sets the maximal number of entries in a sequence of command line stacks beginning with *start*. For the rules, when entries are deleted from command line stacks, see history.setMaxEntry.

start: The id of the first command line stack, whose maximal number of entries should be set.

values: A byte array containing the maximum values, which should be set for the

command line stacks starting *start*. With this command only values up to 255 can be set. For setting higher values the command history.setMaxEntry has to be used.

---

history.setMaxEntry, [int id], [int value]

Sets the maximum number of entries in a command line stack. If the maximum number of entries is exceeded by a new entry, the first entry of the stack will be deleted. If by using *history.setMaxEntry* the maximum number of entries will be reduced such that is smaller than the actual number of entries in the a command line stack, the additional entries will not be deleted. It is just guaranteed, that the number of entries will not grow further. Also if by loading a new history file, the number of entries goes beyond *value*, no entries will be deleted. To be really sure, that after a *history.setMaxEntry* the command line stack is smaller than *value*, it should be cleared with history.clear.

id: The id of the command line stack, whose maximum entry number should be set.

value: The new maximum entry number of the stack.

---

history.setName, [File name]

Sets the currrent history file name, under which the command line stacks will be saved upon ending the Commander name. The current history name will also be set by the commands history.load and history.save. The default history file name upon start of the program name is *Aida.hst*.

name: The new history file name.

Commands of class *log*

This class contains commands for managing log windows in the Commander. For ease of use, in most commands the parameter *name* can be omitted. Then the command is applied to the log window last referenced. If no log window yet exists, a log window with a default name will be created.

---

log.append, [String text], {String name}

Appends the string *text* to the contents of the log window named *name* or the last referenced log window.

text: The text to be appended.

name: The name of the log window, into which the output should be made.

---

log.buffer, [int rows], {String name}

Changes the buffer size of the log window named *name* or the last referenced log window to *rows* rows.

rows: The buffer size in rows.

name: The name of the log window, whose buffer size should be changed.

---

log.clear, {String name}

Clears the contents of the log window named *name* or the last referenced log window.

name: The name of the log window.

---

log.clearall

Clears the contents of all log windows.

---

log.create, {String name}

Creates a new log window named *name* or with a default name.

name: The name of the log window. By omitting this parameter, a default name will be used.

---

log.dispose, {String name}
Disposes the log window named *name* or the last referenced log window.

name: The name of the log window.

---

log.disposeall
Disposes all log windows.

---

log.heapInfo, {String name}
Shows information about the currently free, the requested, and maximally requestable Java heap memory . Because of Java Bug ID #4686462 a value, which is to high, will be shown.

name: The name of the log window, into which the output should be made.

---

log.insertEnvCodes, [String raw], {String name}
Replaces the system environment variables and Java system variables embedded into the string *raw* and appends the output to the contents of log window named *name* or the last referenced log.

raw: The raw text, whose embedded system environment variables and Java system variables will be replaced and then appended.

name: The name of the log window, into which the output should be made.

---

log.message, [String message], {String name}
Appends the message *message* together with current date and time in a separate row to the contents of a log window.

message: The message to be appended.

name: The name of the log window, into which the output should be made.

---

log.packages, {String name}
Show all currently active packages including version information if available.

name: The name of the log window, into which the output should be made.

---

log.properties, {String name}
Shows all Java system variables in alphabetic order. The Java system variables might be set by the Java system itself, can be set with the parameter "-Dpropertyname=value" upon start of the Java Virtual Machine, can be set explicitely set by the user, or are set internally by the Java application (AIDA Commander or Visual Objects, running on the same Java Virtual Machine.

name: The name of the log window, into which the output should be made.

---

log.property, [String key], {String name}
Shows the content of the Java system variable *key*.

key: The name of the Java system variable.

name: The name of the log window, into which the output should be made.

---

log.setWrapColumn, [int column], {String name}

Specifies if and how long rows should be wrapped for display in the log window.

column: The column, after which the line should be wrapped. With *column* = **0**, no additional wrap will be made. With *column* > **0** the additional line wrap will be made character based after *column*. For *column* < **0** the wrap will be made at word borders (white spaces are considered).

name: The name of the log window,whose wrap column value should be changed.

---

log.severe, [String message], {String name}

Appends the message *message* together with current date and time in a separate row to the contents of a log window. The line is highlighted by a special colour.

message: The message to be appended.

name: The name of the log window, into which the output should be made.

---

log.systemEnvironment, {String name}

Show the system environment of the java process.

name: The name of the log window, into which the output should be made.

---

log.warning, [String message], {String name}

Appends the message *message* together with current date and time in a separate row to the contents of a log window. The line is highlighted by a special colour.

message: The message to be appended.

name: The name of the log window, into which the output should be made.

Commands of class *tools*

tools.echo, [String param]

Returns the entered parameter as answer. Mainly used for test purposes.

param: The parameter used for the echo.

Return value: +[tools.echo]0,[param]

---

tools.exec, [String commandLine]

Executes the external command *commandLine* on operating system level. This internal command has major differences compared to the internal command core.run. Whereas *core.run* uses a dll (dynamic link library), *tools.exec* uses the Java internal method *Runtime.getRuntime().exec(String command)*. For the use of these commands these means the following differences:

1. *core.run* needs a special dll (or similar library) for every operating system. Currently only the dll for the Windows OS is contained in the Commander package.. *tools.exec* works for all operatings systems, which have a working Java environment.
2. Under Windows OS, with *tools.exec* onley exe files can be started, which means no batch files or other files which are assigned to an application (e.g. Word documents or html files.

3. Under Windows OS files needing a shell window for execution cannot be started with *tools.exec*.

commandLine: The command line to be executed, including all its eventual parameters.

---

tools.getEnv, [String name]

Returns the content of the system environment *name. getEnv* doesn't look for the Java system variables, but only for the environment variables of the operating system (e.g. Windows).

name: The name of the system environment variable.

Return value: +[tools.getEnv]0,[content]

content: The content of the system environment variable.

---

tools.getEvaluatedSting, [String raw]

Replaces the system environment variables and Java system variables embedded into the string *raw* and returns the result. The expressions, which should be replaced have to be entered as $(Variable) .

raw: The raw text, whose embedded system environment variables and Java system variables will be replaced and then returned.

Return value: +[tools.getEvaluatedString]0,[evaluatedString]

evaluatedString: The string with the replaced variables.

---

tools.inputText, [String text], [int execute], {int historyID}

Inputs text into the command line of the Commander (see tools.systemInput) to the value text and eventually executes it. If the command line of the Commander is not active (e.g. during execution of a program), then *text* is written to an internal stack and might be executed, when the command line of the Commander becomes active again.

text: The string which should be used as input into the command line of the Commander. Previously entered characters will be deleted.

execute: Flag to specify, if the input into command line should be immediately executed (no extra key press of "Enter" required) or not. *execute=0* means, that the command line should not be executed immediately, for all other values the command will be executed immediately.

historyID: The ID of the command line history stack, the command will be saved after execution. At the moment command line stacks can have the IDs 0 – 255. *historyID=-1* means that the currently active system command line stack will be used. Other negative values or a value > 255 mean, that no command line stack will be used. If the parameter is omitted, the *historyID=-1* will be used as default.

---

tools.login, [String title], {String user}, {String domain}, {String password}

Shows a login box on the screen, with the values *user*, *domain,* and *password* as proposal in the entry fields. The password will be shown coded (with asterisks).After the user has closed the dialog box, the new values will be returned.

title: The title of the login box.

user: The proposed value for the entry field *Name:*

domain: Der The proposed value for the entry field *Domain:*

password: The proposed value for the entry field *Password:*

Return value: +[tools.login] 0,[code],[user],[domain],[password]

- **code** might have the following values:

  0: The dialog box was ended with *OK*.
  1: The dialog box was ended with the *Close*-Button.
- **user**, **domain,** and **password** are the new values entered by the user. These will be returned, even if the user has closed the box with the *Close*-Button.
- For interpretation of return values see also Return values of internal commands.

---

tools.messageBox, [String text], [String title], [int type], [String buttons], [int initValue]
Shows a message box in the main Commander window.

text: The text of the Message Box

title: The title of the Message Box

type: The icon type, which will be displayed in the Message Box. Allowed values are 0-4:

0 NoIcon
1 Error-Icon
2 Information-Icon
3 Warning-Icon
4 Question-Icon

buttons: The texts for the buttons. Theoretically any number of buttons may be generated. If more than one button is desired, the texts of the several buttons have to be separated by commas. the whole parameter has to be put in quotes in this case.

initValue: The number of the preselected button. Numbering of the buttons starts with 0 for the leftmost button, other buttons are counted from left to right. Specifying -1 means, that no button is preselected.

Example:

```
tools.messageBox, Wrong parameter, Error, 1, "Abort, Ignore", 0
```

Return value: In the moment there is no way to determine the number of the pressed key from the Commander internally. However, if the command is called from the POOL runtime by the command *IDECommand*, the return string contains the number

of the button pressed as ASCII-String. If the leftmost button was pressed the answer +*[tools.messageBox]0,0* will be returned, for the next button +*[tools.messageBox]0,1* and so on. If the *Close*-Button of the message box (or Escape) was pressed, the answer will be +*[tools.messageBox]0,-1*.

---

tools.systemInput

Creates an input area in the Commander console window starting at the current cursor position until the end of the line and returns the result. This command is modal, which means, it will not be ended before the user presses one of the keys *Return, Escape*, or *CTRL-q*. If the user enters more characters, than can be actually displayed in the input area, its content will be scrolled horizontally. The maximum number of characters which can be entered into the input area is 255. This command correlates to the command *systeminput.getLine*. It is also used from the POOL runtime to request the command line input. By parametrizing the *systemInput* object the system command line may be changed.

Return value: +[tools.systemInput]0,[key],[text]

- **key**: The key which triggered the end of the input. *key* is alway 3 characters wide, wheras the last character might be a space. *key* can have the following values:

  "RET" (Return), "UP " (Cursor up), "DWN" (Cursor down), "^Q " (CTRL-q), "PGU" (Page up), "PGD" (Page down), "BRK" (Break)

- **text**: The text entered into the input area..

For interpretation of return values see also Return values of internal commands.

Attention: The command *tools.systemInput* must not be used from the Commander as internal command, as the Commander will freeze.However it is allowed to use it from the POOL runtime like e.g.:

```
IDECommand("tools.systemInput");
```

---

tools.update

Forces redrawing of the complete Commander window. This command can be used if the automatic redrawing doesn't work correctly. After changes in the Menu structure or a Toolbar, config.activate should be used.
