LuaCmd

The Lua Commander

Version 1.0

Print Date: 5-May-00

Tecgraf/PUC-Rio Rua Marquês de São Vicente, 225 Prédio Velloso - CEP 22453-900 - Rio de Janeiro - Brasil

Tel. +55 21 512-5984 - Fax. +55 21 259-2232 - E-mail: tecgraf@tecgraf.puc-rio.br - URL: http://www.tecgraf.puc-rio.br

Summary

Overview *3*
Usage *3*
Screen Shots *3*
History *3*
To Do's *4*
Support *5*
Thanks *5*
Author *5*
Copyright *5*
Download *5*
About LuaCmd Online *5*
LUACMD COPYRIGHT NOTICE 6
CONSOLE 7
Output *7*
Print/Error *7*
Command Line *7*
Run File *8*
Initialization *8*
FUNCTIONS 8
`lcListFunc()` *8*
`lcListVar()` *8*
`lcAbout()` *9*
Internal Functions *9*
NOTEPAD 10
String Editor *10*
Do String (Run) *10*
Initialization *10*
FUNCTIONS 10
`lnpadGet() -> (text: string)` *10*
`lnpadSet(text: string)` *10*
`lnpadInsert(text: string)` *10*
`lnpadAdd(text: string)` *10*
`lnpadSave(filename: string)` *10*
`lnpadLoad(filename: string)` *10*
`lnpadClear()` *10*
DEBUGGER 11
State *11*
Controls *11*
Source *11*
Breakpoints *12*
Call Stack *12*
Local Watch *12*
Initialization *12*

LuaCmd

The Lua Commander

Version 1.0

Overview

Lua Commander is a 3 in 1 application/library for Lua programmers. It is composed by a **Console**, a **Notepad** and a **Debugger**.

The **Console** receives every print or error output. It also has a command line where you can type Lua based commands and other sugars.

The **Notepad** is a simple text editor or string editor, NOT a text FILE editor. But you can load text from files and save it to files. It will not keep the file name.

The **Debugger** is an interactive Window based debugger for Lua. It shows the source code been executed, together with the Call Stack and Local Variables Watch. You can set breakpoints, step into/over/out functions, stop everything, an even animate the execution steps over the code.

Usage

The Console source code is independent from the other two, and the two are dependent only on the Console. All the 3 parts were written in C, with no auxiliary Lua files, so there are very few registered functions in Lua.

It uses 2 libraries: Lua (of course) and IUP. IUP is a portable user interface toolkit developed at Tecgraf, but it is copyright Petrobras, one of our partners. So if you download the source code probably you will not be able to compile it. But IUP, as other libraries, soon will be available with a binding for Lua. More information about IUP check the online manual at http://www.tecgraf.puc-rio.br/manuais/iup (in Portuguese). In UNIX, IUP uses the Motif libraries, so you will need them installed on your system.

To minimize this limitation with the source code, the application is capable of **dynamic loading** a C library. As an example we distribute together with the binaries the LuaLib shared library and its .LUA that dynamically registers the library. Check the **LUALIB_REG.LUA** file for the Lua command. You just have to pass the file name and its initialization function as parameters.

The source code is split in a library and in an application. The library is more useful than the application it self because you can embed it in your own application do add a Lua debugger or a Lua console output. But in this case the application interface must be done using IUP. We are studding ways to contour this situation.

Screen Shots

The Console, the Notepad and the Debugger.

History

05 May 2000 - Version 1.0.

To Do's

Known Problems

Some times the debugger state is corrupted and even when inactive pauses every dofile or dostring, but if you step it goes all the way, very strange. I am trying to track this, It does not happen often.

The multiline control used in the Debugger and in the Notepad does not have some functionality we desire. So I can not catch mouse clicks and I can not update the text without moving to the top of the file. We are working on solutions for these restrictions to improve LuaCmd interface.

Some Ideas

Global Watch: just like the Local Watch, but you can manually insert and remove global variables that are updated every step. You can do some of this using the console, but it will be nice to have this feature.

Breakpoints: a better breakpoint management with support for conditional breaks, temporary breaks (ex: run to cursor). Today we do not check if a breakpoint is at a valid line, but using part of the code of **LuaC** we can write a function that can filter invalid breakpoints at lines that never will have a line hook call.

Table Print: if you use a "parent" index in a table to implement object oriented support the print out of a table can have many non interesting levels. The print function can has a mechanism to check for a reserved word such "parent" in a table print out and do not print its contents.

Lua Tools: We can create a new dialog for the Lua Commander called Lua Tools, where **LuaC** and **ToLua** will be available.

Profile: the debugger can have a new state called profile that does not debug, instead it profiles the time for a function to execute. (Any Volunteers?)

LuaShell: the console can be used to implement a simple shell interface in Lua. (mkdir, remove, move, copy, list, attrib/chmod, find, where, ...) (Any Volunteers?)

Some Wishes

Multiple File Editor: a string editor is too simple. A multiple document manager with support for many open files in multiple windows is not so hard to do, but a really good text editor can be. Maybe It is more convenient to use an external editor such VI, EMACS, Word, TextPad, UltraEdit or Notepad. Probably this will be the main window of the Lua Commander.

Stdout/Stderr: I would like to redirect the output from *stdout/stderr* to the console window, but this seems to be impossible or near that. This affects the LuaLib **write** function that can be used with no file specified and it will write to *stdout*.

States: Since you can debug different scripts in the same Lua Commander section. Some functions to reset Lua state to the application initial state would be very interesting. Maybe also some general sate management.

Support

If you interested in help, send comments, critics, suggestions, etc to me. Please, specify platform, compiler, version you are using in your message. And I'm far from been a Lua expert...

Looking for Lua? http://www.tecgraf.puc-rio.br/lua.

But remember: This program is free for every usage. The source code is public available, but the libraries used are copyright their authors and IUP is not public available. The author does not offer any guaranties, nor support, etc...

Thanks

This work was developed at Tecgraf/PUC-Rio and home under my own motivation, but with help and support by many friend and colleges. I would also like to thanks *lhf*, *roberto* and *celes* for the offline help and support using Lua.

Author

Antonio E. Scuri (messages to scuri@tecgraf.puc-rio.br, home page in Portuguese at http://www.tecgraf.puc-rio.br/~scuri).

Copyright

Download

The program source code, HTML pages and tests, and pre-compiled binaries (including IUP and Lua dynamic libraries):

luacmd10.tar.gz
luacmd10.zip

luacmd10_Linux.tar.gz,
luacmd10_Solaris.tar.gz,
luacmd10_IRIX6.tar.gz,
luacmd10_Win32.zip

About LuaCmd Online

This home page should be at http://www.tecgraf.puc-rio.br/~scuri/luacmd. This manual works only in Netscape 4 or above, and Intemet Explorer 4 or above because of the JavaScript 1.1 code used.

There is a printable version of this manual in Adobe Acrobat (LUACMD.PDF 1Mb).

This manual was created using the manual toolkit ManJS, that can be found at http://www.tecgraf.puc-rio.br/~scuri/manjs.

.. *"Make it Reusable, Make it Simple, Make it Small"* ...

LuaCmd COPYRIGHT NOTICE

LuaCmd is free and non-proprietary. It can be used for both academic and commercial purposes at absolutely no cost. There are no royalties or GNU-like "copyleft" restrictions. **LuaCmd** (probably) qualifies as Open Source software. Nevertheless, **LuaCmd** is not in the public domain and TeCGraf keeps its copyright.

If you use **LuaCmd**, please give us credit (a nice way to do this is to include a logo in a web page for your product), but we would appreciate *not* receiving lengthy legal documents to sign.

The legal details are below.

The **LuaCmd** implementation have been entirely designed and written by **Antonio Escaño Scuri** at TeCGraf, PUC-Rio.

This implementation contains no third-party code.

Console

Output

The output multiline receives all the output generated by prints and errors. Also serves as a command history of the command line. When a command is executed by the command line, its strings appears preceded by the ">" character indicating that the command execution was started.

The interface toolkit limits the size of the text in the multiline in 32768 byes. When the output becomes full the oldest text is overwritten. When it reach near full capacity also becomes very slow, you can minimize this reducing even more the size of the output buffer.

The following functions can be called from Lua code to access the console output:

```
lcClearOutput()
lcEnterMessage(msg: string)
lcEnterCommandStr(cmd: string)
```

Print/Error

The Lua functions print and error are redefined to send the output text to the console. The print function can have more than one parameter, and prints tabs between them. Tables, functions and user data are not printed, instead the type is printed "<table>", "<function>", "<cfunction>", "<userdata>".

Command Line

The text written in the command line text box is executed when the user press <Enter>. The execution is NOT a simple dostring. When a command is entered we first try a *getglobal* on it, if it returns a valid object then we print its value, else when evaluate the expression to check if it has a return value, if any its value is printed after the dostring. But it is NOT a shell command line, for example the function "pwd" that show the name of the current directory, must be executed using "pwd()".

So I can write command like these:

```
x = 0
s = "bbb"
t = {4,5,6,{x="aaa"}}
t[1]=3
if x==nil then x = sin(0) else x = sin(10) end
```

and like these:

```
x  (where x is nil) Outputs: nil
s                   Outputs: "bbb"
3+5                 Outputs: 8
t[1]                Outputs: 4
sin(0)              Outputs: 0
```

If a command returns more than one value, all of them are printed:

```
--Take the example:
function f()
  return 1, "name", sin
end
-- When executed:

> f()
```

```
<1° return> = 1
<2° return> = "name"
<3° return> = <cfunction>
```

The command history can be accessed using the up and down keys. The <Esc> key clears the command line.

Check the functions **lcListFunc** and **lcListVar** description bellow to see how functions and variables are printed or in this case inspected.

The following function can be called from Lua code to execute a command as if it was executed from the command line:

```
lcEnterCommand(cmd: string)
```

Run File

Allows you to select a file for a "dofile". After selection the file is automatically executed. The current directory is changed to the file directory.

Initialization

To use the Console inside your application you just have to include the header "lconsole.h" and call the function **luaConsoleCreate**("parent", CloseFlag). Check the LUACMD.C file for an example. And do not forget to call **utlCreateButtonImages** once to initialize the toolbar images, and also **luaConsoleKill** and **utlKillButtonImages** after the program ends the message loop.

Functions

lcListFunc()

Lists all the global functions defined on the current state. C functions are marked with a "(C)" after the function name. Be careful using this function because it can be slow if a large number of functions are defined.

When you inspect a Lua function in the command line instead of just the function name, also the location of the function definition is printed:

```
myfunc = <function> (defined in the file "func.lua" at
line 3)

-- or in the case it is inside of a string
lcListFunc = <function> (defined in a string at line 1)

- a C function
sin = <cfunction>
```

If the function is also a tag method then the print changes to:

```
Tgetglobal = <function> ("getglobal" tag method)
[defined in the file "trace.lua" at line 16]
```

lcListVar()

Lists all the global variables defined on the current state. Be careful using this function because it can be slow if a large number of variables are defined.

When printing variables values, each type is printed differently:

```
1           -- integer number
```

```
1.5000      -- real number
"1"         -- string
"aaa"       -- string
nil         -- nil value

_OUTPUT = <userdata> (tag = -7)  -- user data

t =                -- table
{
  [1] = 4          -- integer number
  [2] = 5
  [3] = 6
  [4] =
  {
    ["x"] = "aaa"  -- string
  }
}
```

Tables are printed up to depth level 5. If a table has more than 5 levels the 6 level has the string "{table too deep}" instead. This avoid an infinite loop in self referenced tables.

`lcAbout()`

Shows information about the version number and authors' name of the used libraries and the application. This function is executed when Lua Commander is started.

Internal Functions

The following functions appears when you execute the lcListFunc function. They are for internal use only, do not use them directly.

`lcHoldCaret`, **`lcPrintFuncVar`**, **`lcPrintVar`**, **`lcPrintTable`**

Notepad

String Editor

This is a very simple text editor, where you can type a Lua expression that is too long for the command line. When you load a file the current directory is changed to the file directory.

Do String (Run)

Simply executes `dostring(lnpadGet())`.

Initialization

To use the Notepad inside your application you just have to include the header "lnotepad.h" and call the function **luaNotepadCreate**("parent", CloseFlag). Check the LUACMD.C file for an example. And do not forget to call **utlCreateButtonImages** once to initialize the toolbar images, and also **luaNotepadKill** and **utlKillButtonImages** after the program ends the message loop.

Functions

`lnpadGet() -> (text: string)`

Returns the text in the Notepad.

`lnpadSet(text: string)`

Replaces the text in the Notepad.

`lnpadInsert(text: string)`

Insert the given text at the cursor position.

`lnpadAdd(text: string)`

Add to the text in the Notepad.

`lnpadSave(filename: string)`

Saves the text in the Notepad to a file with the given filename.

`lnpadLoad(filename: string)`

Loads the text from a file with the given filename.

`lnpadClear()`

Clears the Notepad contents.

Debugger

State

The Lua debugger works in a different way of the traditional debuggers. You will not start debugging a specified file or string, instead the debugger will be activated or not for next executions. Because of this philosophy we always set the debug compilation flag to 1, independent of the $debug in the source code, but this will not be valid for pre-compiled code.

When the debugger is active, every Lua execution pass trough it. But when the debugger is paused, you can execute any code (for example in the command line of the Console window) that it will not pass trough the debugger. By default the debugger will pause on the first line executed of the main function.

Be careful when the debugger is active, because the interface message loop is handled in a different way than when it is not active. So despite you can do things with the debugger active, some results may be weird, like you terminate the application and it doesn't end, click on buttons and they do not respond immediately and so on.

Controls

"Stop" will stops execution of every Lua dostring or dofile or callfunction being executed. It is far more than a lua_error call.

"Pause" will break the execution in the current line. This is useful when executing a long loop, but because of the slow refresh of interface message queue the command maybe not immediately. This is valid for the 'Stop" button too.

"Play" continues executing from a PAUSED state. The execution will stop only if there is a breakpoint or the user click on "Pause" or "Stop". When the code is being executed the source code is NOT displayed in the source multiline, but if you set the **Animate Delay** for 200ms you will seen a nice animation of the code execution.

"Step" executes until one of the step break conditions is reached. If **Step Mode** is **Lines** then the step break condition is the number of lines specified, that is usually 1. But if a function is executed, the step depends on the **Step Function** state. If it is **Into** the execution will step into the function and continue to count lines. If it is **Over**, the function will be executed and when it returns the step line counting will continue. If **Step Mode** is **Out** the execution will proceed until it steps out of the current stack level. If you step out from main, it will execute until ends. When you step out from a function or step over a function the breakpoits that are set in deeper levels of the stack or inside functions will still break the execution.

Source

The source multiline displays the source code of the current file/string being executed. The current line number will be selected each time you step in the execution. Because you can deselect it when scrolling in the code, the button "->" will return the selection to the current line.

When a dotring or a dofile is excuted during a debug step into, the new file or string is loaded. To check for the current directory you can use the "pwd()" functio, It is an

alias to "print(getcwd())". You can also use "chdir(dir_name)" to change the current directory.

Breakpoints

Breakpoints can be set on any line of any file, even if the line will not be executed, so be careful. You can add a breakpoint clicking on B in the source multiline (in fact this will toggle a break in the current line) or selecting "Breakpoint >>" then "Add File...", this will ask you for a file and after that a line number.

Call Stack

When you are PAUSED somewhere in the code, the call stack is displayed in this list box.

Only the function names appears in the list. if you want more information of one level, select it and click on Print, and will be displayed in the Output muliline of the Console window.

```
Stack Level 0:           -- shows the current level
func, at line 16         -- shows the current line at
that level inside function "func"
    [defined in the file "func.lua" at line 9]   --
shows information about function "func"
```

You can also print information about all the levels in the stack, just click on Print All.

```
Stack:
  0| func, at line 16
    [defined in the file "func.lua" at line 9]
  1| <main of "func.lua">, at line 22
    [defined in the file "func.lua"]
```

Local Watch

When you are PAUSED somewhere in the code, the list of the local variables at the current level of the stack is displayed here. If you select another level of the call stack, then you will see the local variables at that level.

Variables that are numbers or strings are displayed in the list box. But if the string is too large or the variable is not a string you can inspect its value just like entering a command in the Console window, by selecting Print (the output will be in the Output multiline of the Console Window).

You can also change the value of a local variable, just select it and click on Set. The string you entered will be evaluated before the variable is changed, so you can type expressions.

Initialization

To use the Debugger inside your application you just have to include the header "ldebug.h" and call the function **luaDebugCreate**("parent", CloseFlag). Check the LUACMD.C file for an example. And do not forget to call **utlCreateButtonImages** once to initialize the toolbar images, and also **luaDebugKill** and **utlKillButtonImages** after the program ends the message loop.