

Far Manager macro system manual

Since 30.09.2012 (Far 3.0.2851), Far Manager uses [Lua](#) programming language (version 5.1) instead of the built-in macro language.

Since 23.04.2014 (LuaMacro build 310), added support for [MoonScript](#) programming language.

A macro is a script (written either in Lua or MoonScript) and its meta-data such as:

- Areas from which the script may be executed
- Keys that initiate execution of the script from those areas
- Prerequisites for the execution of the script
- Run-time flags: EnableOutput and NoSendKeysToPlugins

Functionality

- Macro recording and playback
- Standard Lua libraries
- LuaFAR libraries (“plugin API”)
- [Libraries of macro functions](#) (“macro API”)
- For plugins: function [MacroControl](#)

Loading macrofiles

- [Regular macros](#) and [event handlers](#) are loaded from Lua-files (extension *.lua) and/or MoonScript files (extension *.moon). Further we will call these files *macrofiles*. The macros are loaded when macrofiles are executed by [LuaMacro](#) plugin.
- Macrofiles are loaded from sequence of directories (recursively with their subdirectories), determined by one of the following ways (in order of priorities):
 1. A sequence of directories, specified explicitly (in a command or a function argument)
 2. The value of MacroPath variable in the file [luamacro.ini](#)
 3. The directory %FARPROFILE%\Macros\scripts
- Far Manager itself **never** makes any changes to the above mentioned directories. The files are added, removed, renamed and edited solely by the user.
- One macrofile can contain arbitrary number of macros and event handlers.
- When a macrofile is executed it receives 2 arguments: (1) the full pathname of this macrofile and (2) the value of execution counter in the current session of plugin LuaMacro. If we define 2 variables at top of the macrofile, e.g.

```
local MacroFileName, ExeCounter = ...
```

then these variables will be available to all the macros and event handlers defined in that file.
- If macrofiles are loaded from a sequence of trees `root1;root2;...`, that means that:
 - The tree `root2` loads only after loading `root1`
 - For each tree `rootN` the first macrofile run is `rootN_macroinit.lua` if such a file exists. For the rest of macrofiles in the given tree order of

execution is not defined.

Format of macros and event handlers

Regular macros

A macro is loaded by the global function `Macro` that receives one argument – a table containing parameters of the macro. On successful completion the function returns `true`.

```
Macro {
  area          = "Shell Info Tree";           -- str
  key           = "CtrlF11 ShiftHome";        -- str
  description   = "Macro example";           -- str
  flags        = "NoPluginPanels EmptyCommandLine"; -- str
  filemask     = "*.txt,*.cpp";              -- str
  priority     = 50;                          -- num
  sortpriority = 50;                          -- num
  selected     = true;                        -- boo
  condition    = function(key) return Far.Height>30 end; -- fun
  action       = function() msgbox("", "Macro example") end; -- fun
  id           = "F0109446-AA63-4873-AEC3-17AEE993AA53"; -- str
}
```

1. The field `area` should contain names of one or more areas, separated by whitespaces.
2. The field `key` can contain names of one or more keys, separated by whitespaces.

Keys can contain modifiers: `Ctrl`, `LCtrl`, `RCtrl`, `Alt`, `RAlt`, `LAlt`, `Shift`. `Ctrl` means “any of `LCtrl`, `RCtrl`”, the same goes for `Alt`. The order of modifiers can be arbitrary.

Alternatively, the field `key` can be specified as a regular expression, enclosed in slashes (/).

- In this case `/Ctrl/` will not work when `RCtrl` is pressed, it should be specified explicitly `/[LR]Ctrl/` etc.
- Also in this case it is necessary to maintain the order in the sequence `Ctrl,Alt,Shift`, e.g. `/[LR]Alt[LR]CtrlF1/` would never work.

3. Optional field `flags` may contain a set of [flags](#) separated with spaces. Some changes in names or interpretation of flags are described [here](#).
4. Optional field `priority` – a number in the range 0 to 100. The default value = 50.
Macros added via [MCTL_ADDMACRO](#) have `priority = 50`.
5. Optional field `sortpriority` – a number in the range 0 to 100. The default value = 50.
This field affects the order of macros in the macro selection menu.
6. Optional field `selected` – a boolean.
This field assigns this macro to be initially selected in the macro selection menu.
7. Optional field `filemask` – a string.
It is applicable only for `Editor` and `Viewer` areas. It is processed according to the same rules that Far Manager applies for file masks when searching from panels, etc. If name of the file open in editor or viewer does not match the given mask, the macro will not execute.
8. Optional field `condition` – a function.
 - It is called with one argument: the name of pressed key. For auto-started macros it is called with no arguments.
 - If the function returns `false/nil/nothing`, the macro will not execute.
 - If it returns a number then this number is used instead of `priority`.
 - In other cases of return value (e.g., `true`) `priority` is used.
9. Field `action` – a function.
If the macro has passed successfully all preliminary checks (`area`, `flags`, `filemask`, `priority`) then this function is called.
10. More than one macro for (`key,area`) combination is allowed. In this case a macro with highest `priority` is executed. If there are multiple macros having the same `priority` then the macro selection menu is displayed.
The auto-starting macros are executed all, one by one, independently from `priority`s. The order of their execution is not defined.

Keyboard macros

- Keyboard macros are usually used for quick recording and replaying key-press sequences. These macros are less powerful than regular macros and for the long-term use regular macros should be preferred.
- Keyboard macros are loaded from Lua-files (extension *.lua), residing in the directory
%FARPROFILE%\Macros\internal.
- Far Manager automatically creates, modifies and deletes files in this directory in accordance with operations conducted on keyboard macros. It is **not** recommended to manually edit these files, except for their deletion.
- Modifications to keyboard macros become permanent either after executing [MacroControl\(MCTL_SAVEALL\)](#), or automatically (when the “Auto save setup” option is on).
- The field “area” may contain only one area name.
- The field “key” may contain only one key name.
In key names only modifiers Ctrl, Alt, Shift may be used, modifiers LCtrl, RCtrl, LAlt, RAlt are not supported. Regular expressions are not supported.
- It is not allowed to have more than one keyboard macro for a (key,area) combination.
- Keyboard macros have higher priority than all other macros. This avoids the need of resolving conflicts when one creates a temporary macro (the typical use case of keyboard macros).

Event handlers

Like [regular macros](#), event handlers are loaded from Lua-files (extension *.lua), residing in the directory

%FARPROFILE%\Macros\scripts and its subdirectories. Each Lua-file may contain both macros and event handlers.

Loading a handler occurs when the global function Event is called. It receives one argument – a table containing parameters of the event handler. On successful completion the function returns true.

```
Event {
  group      = "EditorEvent";           -- string
  description = "Event example";       -- string (o
  filemask   = "*.txt,*.cpp";         -- string (o
  priority   = 50;                    -- number (o
  condition  = function() ..... end;  -- function
  action     = function() ..... end;  -- function
  id         = "F0109446-AA63-4873-AEC3-17AEE993AA53"; -- string (o
}
```

The field group can have one of the following values:

"DialogEvent", "EditorEvent", "EditorInput", "ExitFAR",
"ViewerEvent", "ConsoleInput".

These names are derived from names of the corresponding functions, exported by the plugin, e.g.:

```
export.ProcessDialogEvent -> DialogEvent
```

The functions condition and action are called with the same parameters as the corresponding exported functions are called (see LuaFAR manual).

When there are multiple event handlers for the same event (i.e. handlers with the same group value), these handlers will be called one after one: a handler having higher priority is called first. Priorities are evaluated dynamically accounting for condition() results if any, the same way it is done for macros.

The ExitFAR handler is called in the following cases: (a) exit from Far Manager, (b) unloading LuaMacro plugin, (c) unloading or reloading macros. The handler

receives one argument of boolean type: false for cases (a) and (b); true for case (c).

Adding items to plugins' menus

Like the [regular macros](#), the added menu items are loaded from Lua- and MoonScript-files, residing in the directory %FARPROFILE%\Macros\scripts and its subdirectories.

A menu item is loaded by the global function `MenuItem` that takes one argument – a table with parameters. The function returns `true` on success.

```
MenuItem {
  description = "Menu item";           -- string (optional)
  menu       = "Plugins Disks Config"; -- string
  area       = "Shell Editor Viewer Dialog Menu"; -- string (optional)
  guid       = "A435D567-AD64-4DD1-8C61-28CB90358817"; -- string
  text       = function(menu,area) return "Hello!" end; -- string, or function
  action     = function(OpenFrom,Item) ..... end; -- function
}
```

The fields `description` and `area` are optional with the default values being an empty string. Other fields are mandatory.

- The field `menu` is the list of Far Manager menus the given menu item should be added to. The valid values of list elements are "Plugins", "Disks" and "Config" that stand for plugins menu, disk menu and plugins configuration menu respectively.
- The field `area` is the list of areas, where the given menu item should be added to the **plugins menu** when it is invoked. This field is used only if the field `menu` contains `Plugins`. The valid values of list elements coincide with the names of macro areas.
- The field `guid` contains a unique identifier (GUID) of the given menu item.
- The field `text`: if it is a string then that string is used as the menu item's text. Otherwise it should be a function. The menu item is added only if the function returned a string value.
The function receives 2 arguments:

1. *menu* – type of the menu (either of: "Plugins", "Disks" or "Config")
2. *area* – name of the current macroarea.

- The field `action` is the function that is called upon activation of the given menu item. It takes the same arguments as the function `export.Open` (see `luaфар_manual.chm`), except `Guid`. If the function is called from the plugins configuration menu then both the arguments have `nil` value. The returned value is not used.

Adding command line prefixes

Like the [regular macros](#), the added command line prefixes are loaded from Lua- and MoonScript-files, residing in the directory %FARPROFILE%\Macros\scripts and its subdirectories.

Prefixes are loaded by the global function `CommandLine` that takes one argument – a table with parameters. The function returns the number of successfully loaded prefixes.

```
CommandLine {  
  description = "Adding prefixes";           -- string (optional)  
  prefixes = "abc:def:1234";                -- string  
  action = function(prefix,text) ..... end; -- function  
}
```

- The field `prefixes` is a list of prefixes delimited by colons. Spaces are not allowed.
- The field `action` is the function that is called when the command line begins with one of the registered prefixes. It takes 2 arguments: `prefix` is the actual prefix in lower case; `text` is the rest of the command line with leading and trailing spaces stripped.

Adding panel modules

“Panel module” is a set of Lua functions placed in a table and loaded with `PanelModule` function.

- Names of those functions and their parameter sets coincide with functions from export table (see LuaFAR manual).
- Here is the list of module-exported functions that are supported by the plugin:
Analyse, ClosePanel, Compare, DeleteFiles, GetFiles, GetFindData, GetOpenPanelInfo, MakeDirectory, Open, ProcessHostFile, ProcessPanelEvent, ProcessPanelInput, PutFiles, SetDirectory, SetFindList.
- Every panel module must contain a table `Info` with a mandatory field `Guid`. Other fields are optional.

```
-- Create a panel module
local mod = {}
mod.Info = {
    Guid      = win.Uuid("FBBC5FBF-AE9F-46EC-999C-C744F7D898B6"); --
    Version   = "";
    Title     = "";
    Description = "";
    Author    = "";
}

-- Add only those "exported" functions that are needed for this pane
mod.Analyse   = function(...) ..... end
mod.Open      = function(...) ..... end
mod.GetFindData = function(...) ..... end
.....

-- Load the module
PanelModule(mod)
```

Notes:

1. To create a panel from the command line or from the plugins menu, the existing function [CommandLine](#) and [MenuItem](#) should be used. Their `action()` should return 2 values: (1) the module table and (2) the panel

object (any non-false Lua value).

2. Function `mod.open` is called by the plugin only with the following values of `OpenFrom` parameter:
OPEN_ANALYSE, OPEN_FINDLIST and OPEN_SHORTCUT.

See also: [Demo Example](#)

Demo example

```
if ({ far.AdvControl("ACTL_GETFARMANAGERVERSION", true) })[4] < 5171

local F = far.Flags
local Title = "Demo panel in LuaMacro"
local mod = {}

mod.Info = {
  Guid = win.Uuid("715E5E90-DEB9-470A-84CE-7CF8D92A7B05")
}

local function FileToObject(FileName)
  FileName = far.ConvertPath(FileName, "CPM_FULL")
  local fp = io.open(FileName)
  if fp then
    local obj = { HostFile=FileName; List={} }
    for line in fp:lines() do
      table.insert(obj.List, {FileName=line})
    end
    fp:close()
    return obj
  end
end

function mod.Analyse(Data)
  return Data.FileName and Data.FileName:sub(-5):lower() == ".abcd"
end

function mod.Open(OpenFrom, Guid, Item)
  if OpenFrom == F.OPEN_ANALYSE then
    return FileToObject(Item.FileName)
  elseif OpenFrom == F.OPEN_SHORTCUT then
    return FileToObject(Item.HostFile)
  elseif OpenFrom == F.OPEN_FINDLIST then
    -- If we uncomment the line "return {}", then this module will b
    -- used instead of TmpPanel for displaying search results.
    ---- return {}
  end
end

function mod.GetFindData(object, handle, OpMode)
  return object.List
end
```



```

function mod.GetOpenPanelInfo(object, handle)
  return {
    HostFile = object.HostFile;
    PanelTitle = Title;
    StartSortMode = F.SM_UNSORTED;
    StartSortOrder = 0;
    ShortcutData = "";
    Flags = bit64.bor(F.OPIF_SHORTCUT, F.OPIF_ADDDOTS);
  }
end

```

```

function mod.SetFindList (object, handle, Items)
  object.List = Items
  return true
end

```

```

MenuItem {
  description = Title;
  menu      = "Plugins";
  area      = "Shell";
  guid      = "5E1ECBD6-F6E1-4A02-AC90-DB49DB6E350C";
  text      = Title;
  action = function(OpenFrom, Item)
    return mod, FileToObject(APanel.Current)
  end;
}

```

```

CommandLine {
  description = Title;
  prefixes = "abcd";
  action = function(prefix, text)
    if text then return mod, FileToObject(text); end
  end;
}

```

```

PanelModule(mod)

```

External modules

Lua modules can be placed in %FARPROFILE%\Macros\modules and its subdirectories, as
%FARPROFILE%\Macros\modules\?.lua;%FARPROFILE%\Macros\modules\?
\init.lua;
is automatically added to package.path.

Binary modules (DLL) can be placed in %FARPROFILE%\Macros\lib32 and %FARPROFILE%\Macros\lib64 and their subdirectories as those paths are automatically added to package.cpath.

LuaMacro plugin

This plugin is necessary for macros to work, therefore it should be installed. The same is true regarding the runtime (*lua51.dll*, *lua51.dll*, *lua51.dll* and *lpeg.dll*) that is necessary for plugin's work.

When Far Manager exits the LuaMacro plugin is unloaded after all other plugins, in order to be able to process MCTL_XXX requests from [ExitFARW](#) functions of other plugins.

Command line operations

- `macro: load [path]` (Re)load macrofiles. An optional parameter `path` has the same meaning as the field `Path` in struct [FarMacroLoad](#).
- `macro: save`
Save the created or modified [keyboard](#) macros.
- `macro: unload`
Unload macros (except those created with operation [MCTL_ADDMACRO](#)) and event handlers.
- `macro: about`
Show versions of the plugin and the libraries it is using.
- `lua: [=] <code>`
- `moon: [=] <code>`
Execute the code `<code>` written correspondingly in Lua or MoonScript. If `<code>` is preceded with a character `=` then `far.Show()` is called, e.g.:
`lua:=5+2,6,"foo"` is equivalent to `lua:far.Show(5+2,6,"foo")`.
- `lua: [=] @<filename> [<args>]`
- `moon: [=] @<filename> [<args>]`
Execute the script `<filename>` written correspondingly in Lua or MoonScript.
 - For passing arguments to the script they should be specified after the file name, separated with whitespace.
 - Arguments are a sequence of expressions delimited with commas.
 - The expressions must be written in the same programming language as the script.
 - The global (within the environment of the script) variable `_filename` contains the file name.

Note 1:

Prefix `lm:` can be used instead of prefix `macro:` – they are equivalent.

Note 2:

There are also `luas:` and `moons:` prefixes that can be used instead of respectively `lua:` and `moon:`. In that case no macro is created and the code is executed immediately (“synchronously”). If the code terminates in a normal way then `CmdLine.Result` is a table containing an array of returned values and the

field `n` of the table is the number of returned values. If the code execution is interrupted by an error the value of `CmdLine.Result` is `nil`.

Note 3:

[Additional](#) command line prefixes can be defined and loaded from macrofiles.

File luamacro.ini

The file `luamacro.ini` contains some plugin's settings. If the file is missing, or some setting is missing then the plugin will use the default value for the given setting.

- `MacroPath` Defines paths from which macrofiles are loaded. It is a sequence of 0 or more paths separated with semicolons. The default value is `%FARPROFILE%\Macros\scripts`.

File luafar_init.lua

Plugin LuaMacro runs the file %FARPROFILE%\luafar_init.lua (if that file exists) before loading its default script. See description of this feature in LuaFAR manual.

Libraries of macro functions

The description of the APIs in this document is **not** self-contained, it is *supplementing and clarifying* the description of the Far objects' properties and functions in the [Macro language](#) section of Far Encyclopedia.

Ideally, the API must match the original macro language API, with the exception of the cases listed in article [API changes in comparison to the macro language](#).

See also: [Restrictions in the use of some functions](#)

API changes in comparison to the macro language

1. All identifiers are case sensitive.
2. Instead of directly specifying the keys you use function [Keys](#).
3. [\\$AKey](#) → `Keys("AKey")`
4. [\\$SelWord](#) → `Keys("SelWord")`
5. [\\$XLat](#) → `Keys("XLat")`
6. [\\$Exit](#) → `exit()`
7. Logical properties have type *boolean* (so it does not make sense to compare them with the number 0).
8. Function [prompt](#) can return either a string or a false (but never a number 0).
9. Many functions are placed in the table [mf](#) (abbreviation from *macrofunctions*), e.g.: `mf.abs`, `mf.fsplit`, etc.
10. Functions [Far.Window Scroll](#), [mf.beep](#), [mf.fexist](#) and [Panel.SetPath](#) return a boolean rather than a number.
11. Functions [mload](#) and [msave](#) have changed, see their descriptions.
12. Context dependent properties are placed in the table [Object](#), e.g.: `Object.CurPos`, `Object.Empty`.
13. [CheckHotkey](#) → `Object.CheckHotkey`
14. [GetHotkey](#) → `Object.GetHotkey`
15. Logical properties for testing execution areas are in table [Area](#), e.g.: `Area.Editor`, `Area.Shell`.
16. [Macro.Area](#) → `Area.Current`.
17. `Dialog.AutoComplete` → `Area.DialogAutoCompletion`
18. `Shell.AutoComplete` → `Area.ShellAutoCompletion`
19. `CallPlugin` → [Plugin.Call](#). This call is always asynchronous. For synchronous calls use [Plugin.SyncCall](#).
20. Functions [Plugin.Exist](#), `Plugin.Menu`, `Plugin.Config` and `Plugin.Command` return a boolean value.
21. [mmode](#)(3, x) no more affects synchronicity/asynchronicity of calls to plugins; it does nothing and returns 0.
22. [Dlg.Info.Id](#) → `Dlg.Id`
23. `Dlg.Info.Owner` → `Dlg.Owner`
24. [Far.Cfg.Get](#) → `Far.Cfg_Get`. This function returns a string in case of success and false in case of failure.

25. [Far.Cfg.Err](#) does not exist anymore.
26. [FullScreen](#) -> `Far.FullScreen`
27. [IsUserAdmin](#) -> `Far.IsUserAdmin`
28. [History.Disable](#) -> `Far.DisableHistory`
29. [KbdLayout](#) -> `Far.KbdLayout`
30. [KeyBar.Show](#) -> `Far.KeyBar_Show`
31. [Window.Scroll](#) -> [Far.Window.Scroll](#)
32. [Menu.Info.Id](#) -> `Menu.Id`
33. [MsX](#), [MsY](#), [MsButton](#), [MsCtrlState](#), [MsEventFlags](#) — see table [Mouse](#)
34. [RCounter](#) does not exist anymore.
35. [Macro.Const](#), [Macro.Func](#), [Macro.Keyword](#) и [Macro.Var](#) do not exist anymore.
36. Macro specification: flag `DisableOutput` does not exist; screen redraw is disabled by default; added flag `EnableOutput` (apply it if screen redraw is needed during macro execution).
37. Macro specification: flags `Selection` and `NoSelection` are no more applicable for editor/viewer/dialog areas. For these areas one should use respectively `EVSelection` and `NoEVSelection` flags.
38. Macro specification: flag `RunAfterFARStart` is in effect also when Far Manager is run with `/e` or `/v` command line switch. In those cases only macros whose area field contains respectively `Editor` or `Viewer` are run.

See also: [Restrictions in the use of some functions](#)

Global properties and functions

Properties: none.

Functions:

[akey](#)
band, bnot, bor, bxor, lshift, rshift
[eval](#)
[exit](#)
[Keys](#)
[mmode](#)
[msgbox](#)
[print](#)
[prompt](#)

Notes:

1. Functions *band*, *bnot*, *bor*, *bxor*, *lshift* и *rshift* (bitwise operations) are global aliases of the same-named functions of *bit64* library. (see LuaFAR manual).
2. Functions [akey](#) and [mmode](#), when called from within function *condit* return *false*.
3. For uniformity sake, all the above functions (except the bitwise operations) are duplicated in table [mf](#), for example: [eval](#) и *mf.eval* reference the same function.

exit

exit ()

Parameters:

none

Returns:

nothing

Description:

Exit macro.

See also:

[Restrictions in the use of some functions](#)

Keys

Keys (...)

Parameters:

One or more arguments of string type.

Each argument may contain multiple keys separated by whitespace characters.

The arguments are case insensitive.

Returns:

nothing

Description:

Send one or more keys to Far Manager.

Notes:

1. Special keys:

"AKey" - send Far Manager the key that started this macro.

"SelWord" - select the word under cursor.

"XLat" - convert the word under cursor.

"EnOut" - enable screen output (same effect as *mmode(1,0)*)

"DisOut" - disable screen output (same effect as *mmode(1,1)*)

2. Each key may be preceded with a multiplier, e.g. "3*Down" is equivalent to "Down Down Down".

Example:

```
mykeys = "CtrlF5 Esc"
```

```
Keys("AKey A b CtrlC ShiftEnter", mykeys)
```

See also:

[Restrictions in the use of some functions](#)

mf

Properties: none.

Functions:

abs	len
acall	max
AddExitHandler	mdelete
akey	min
asc	mload
atoi	mmode
beep	mod
chr	msave
clip	msgbox
date	postmacro
env	print
eval	prompt
exit	replace
fattr	rindex
fexist	size2str
float	sleep
flock	string
fmatch	strpad
fsplit	strwrap
GetMacroCopy	substr
iif	testfolder
index	trim
int	ucase
itoa	usermenu
key	waitkey
Keys	xlat
lcase	

acall

```
... = mf.acall (func, ...)
```

Parameters:

func: function
...: 0 or more Lua values

Returns:

...: 0 or more Lua values

Description:

This function calls "asynchronously" the function *func*, passing it arguments.

mf.acall is a sort of specialization of [Plugin.Call](#) for the [LuaMac](#) but unlike `Plugin.Call` it allows code execution in the context of as well as passing and returning any Lua values.

Like `Plugin.Call`, *mf.acall* is "asynchronous": when the function *fu* a dialog or a menu on the screen, *mf.acall* immediately terminates

If the function *func* pops up no dialogs or menus on the screen, it operation mode: in this case *mf.acall* returns all values, returned

See also:

[Restrictions in the use of some functions](#)

AddExitHandler

`mf.AddExitHandler (handler)`

Parameters:

handler: function

Returns:

nothing

Description:

1. This function adds a handler that will be called at the end of the macro execution.
2. The handler will be called both in the case of normal macro completion and in the case of error completion.
3. If multiple handlers were added during macro execution then the will be called in the order reverse to the order of their addin

Usage example:

```
local fp = assert(io.open("some file.txt"))
mf.AddExitHandler(function() fp:close() end)
-- use fp; return from multiple places; do not care about closing
```

See also:

[Restrictions in the use of some functions](#)

eval

```
ret = eval(S[, Mode[, Lang]])
```

This function corresponds to the [description](#) in Far Manager Encyclopedia, with the following extensions:

1. eval can execute either Lua or MoonScript code

Added an optional 3-rd parameter `Lang` specifying the programming language of the parameter `S` in modes 1, 2 and 3. Acceptable values are "lua" and "moonscript". The default value is "lua".

2. Parameter S can specify a script-file

In modes 0, 1 and 3 parameter `S` can refer to a script-file, if this parameter begins with a `@` character. In this case the `S` parameter must be in the following format:

```
@<script-file name> [<script parameters>]
```

- The file name can contain environment variables, they will be expanded.
- Optional script parameters are a list of expressions separated by commas.
- The expression should use the same programming language as the script.

Example:

```
eval("@%MyFarScripts%\calc.moon 'factorial', 3+5", 0, "moonscri
```

3. New return codes of eval(S, 2)

- 0 (normal return) : is followed by any additional values that might be returned by the “evaluated” macro.
- -3 : if the macro selection menu was displayed and cancelled by the user.
- -4 : if the “evaluated” macro was interrupted by a run-time error.

GetMacroCopy

```
macro = mf.GetMacroCopy (index)
```

Parameters:

```
index: integer
```

Returns:

```
macro: table or nil
```

Description:

GetMacroCopy returns a copy of a loaded macro (or event handler) t by its index in the internal array (starting from 1). If the index than the size of the array, the function returns nil, so one can d the end of the array.

Notes:

- * Inactive (unloaded or deleted) elements have the field "disabled"
- * To distinguish between macro table from event handler table: the "area" of type string that is present only in macro tables.

mdelete

```
ret = mf.mdelete (key, name [, location])
```

Parameters:

```
key:      string  
name:     string  
location: string ("roaming" or "local"; default: "roaming")
```

Returns:

```
ret:      boolean
```

Description:

Function *mdelete* deletes a value or key from the database.
To delete a key, specify *name* == "*" (asterisk).

mload

```
value, errmsg = mf.mload (key, name [, location])
```

Parameters:

```
key:      string
name:     string
location: string ("roaming" or "local"; default: "roaming")
```

Returns:

```
value:    number, string, boolean, table, int64 or nil.
errmsg:   nil on success, string on failure.
```

Description:

Function *mload* reads a value from the database.
If the second return value is nil then the first return value is v

Note:

int64 - a distinguished type of userdata, created by the bit64 lib

msave

```
ret = mf.msave (key, name, value [, location])
```

Parameters:

```
key:      string
name:     string
value:    nil, boolean, number, string, table, int64
location: string ("roaming" or "local"; default: "roaming")
```

Returns:

```
ret:      boolean
```

Description:

Function *msave* saves the specified value into the database.

When you save the table the following will be preserved:

```
keys of types:  number, string, boolean, table.
values of types: number, string, boolean, table, int64.
```

Nested tables and recursive references are correctly processed.

The link between a table and its metatable is preserved.

Note:

int64 - a distinguished type of userdata, created by the bit64 lib

postmacro

```
result = mf.postmacro (func [, ...])
```

Parameters:

```
func:    function  
...:    0 or more Lua values
```

Returns:

```
result:  boolean
```

Description:

The function places a new macro in a queue for execution. When the execution begins *func* is called with arguments ...

usermenu

mf.usermenu (mode, filename)

Parameters:

mode: number (0 by default)
filename: string or nil

Returns:

nothing

Description:

Opens or creates a user menu.

Function behavior depends on the least significant byte of *mode*:

- 0: equivalent to pressing F2 in panels; *filename* is ignored.
- 1: displays the dialog for user menu selection; *filename* is ignored.
- 2: opens user menu from the file "as is", i.e. by specified *file*.
- 3: opens user menu from the file *filename* in %farprofile%\Menus (the directory is created automatically).

If the bit 0x100 of *mode* is set the function will return only upon the menu (synchronous call). If that bit is cleared the function will return immediately when the menu is opened (asynchronous call).

See also:

[Restrictions in the use of some functions](#)

Area

Area - a table with the following fields:

Properties:

Current:	string
Other :	boolean
Shell :	boolean
Viewer :	boolean
Editor :	boolean
Dialog :	boolean
Search :	boolean
Disks :	boolean
MainMenu :	boolean
Menu :	boolean
Help :	boolean
Info :	boolean
QView :	boolean
Tree :	boolean
FindFolder :	boolean
UserMenu :	boolean
ShellAutoCompletion:	boolean
DialogAutoCompletion:	boolean

Functions:

None.

APanel, PPanel

APanel, PPanel - tables with the following fields:

Properties:

Bof :	boolean
ColumnCount :	number
CurPos :	number
Current :	string
DriveType :	number
Empty :	boolean
Eof :	boolean
FilePanel :	boolean
Filter :	boolean
Folder :	boolean
Format :	string
Height :	number
HostFile :	string
ItemCount :	number
Left :	boolean
LFN :	boolean
OPIFlags :	number
Path :	string
Path0 :	string
Plugin :	boolean
Prefix :	string
Root :	boolean
SelCount :	number
Selected :	boolean
Type :	number
UNCPath :	string
Visible :	boolean
Width :	number

Functions:

None.

Panel

Panel - a table with the following fields:

Properties:

None.

Functions:

[FAttr](#)
[FExist](#)
[Item](#)
[Select](#)
[SetPath](#)
[SetPos](#)
[SetPosIdx](#)
[CustomSortMenu](#)
[LoadCustomSortMode](#)
[SetCustomSortMode](#)

Note:

Functions [CustomSortMenu](#), [LoadCustomSortMode](#) and [SetCustomSortMode](#) are available only if the Lua engine is LuaJIT 2.x.

CustomSortMenu

Panel.CustomSortMenu ()

Parameters:

None

Returns:

Nothing

Description:

Displays a menu containing the list of loaded custom sort modes.
Pressing **Enter** will set the selected sort mode in the active panel
pressing **CtrlEnter** - in the passive panel, **CtrlShiftEnter** - in bot

Keys **Add** and **Subtract** work as in the "Sort by" menu in Far Manager
Also supported are modifiers **Ctrl** and **CtrlShift** that determine cho
of panels for setting sort mode on them.

LoadCustomSortMode

Panel.LoadCustomSortMode (Mode, Settings)

Parameters:

Mode: sort mode; integer ≥ 100 and $\leq 0x7FFFFFFF$

Settings: a table containing the following fields:

Condition:

Function. If it is specified it will be called with one arg
If the return value is false then the sort is cancelled.

Note that this function can reload all sorting parameters b

Compare:

Function, see its description below.

DirectoriesFirst, SelectedFirst, RevertSorting, SortGroups:

These optional fields specify corresponding sorting options
0 - the option is off, 1 - the option is on. Any other valu
mean "use the current setting of Far Manager".

InvertByDefault:

Whether the default sort direction is the inverse one.

Indicator:

Indication of sort mode on the panel, a two character strin
(1-st character for the direct sort mode, 2-nd for the inve

NoSortEqualsByName:

By default the elements equal from the sorting algorithm's
are sorted by name. If that is not desired, set this field

Description:

Textual description of the sorting mode. If this field is s
it is used in the custom sort menu (see [Panel.CustomSortMen](#)

SortFunction:

Specify the sorting algorithm out of the 2 available ones.
It is a string: "shellsort" (default value) or "qsort".

InitSort:

Function. If specified, it will be called before the sortin
It receives one argument: *FarOptions* table (see the same-na
of the function *Compare*).

EndSort:

Function. If specified, it will be called after the sorting

If the value of *Settings* is nil or false, it means unloading (re
sorting mode.

Returns:

Nothing

Description:

This function loads (or unloads) a custom panel sort mode. Once the mode is loaded it can be set in the panel by means of call [Panel.SetCustomSortMode](#).

Function Compare

```
result = Compare (Pi1, Pi2, FarOptions)
```

Parameters:

Pi1 и Pi2 - panel elements being compared, structures of [Sortin](#)
FarOptions - a table containing the current Far Manager panel so (all values are boolean): *DirectoriesFirst, Selecte RevertSorting, SortGroups, NumericSort, CaseSensiti*

Returns:

result - if the 1-st element should appear after direct sort a negative number should be returned, if below - a and if the elements are equal by sorting criteria -

Note 1:

Custom panel sorting uses the LuaJIT's FFI library. The use of FFI with its [documentation](#).

Note 2:

Custom panel sort modes are automatically restored after Far Manag that the configuration has been saved and the corresponding Panel. calls are done during the process of loading macros.

Restoring of custom panel sort modes takes place after the macros before the execution of auto-starting macros.

Note 3:

The custom panel sort modes are forcibly unloaded when the macros

Example:

```
-- Load the sorting by file name length.
local ffi = require "ffi"
local C = ffi.C
Panel.LoadCustomSortMode (100,
{
  Compare = function(p1, p2, opt)
    local l1, l2 = C.wcslen(p1.FileName), C.wcslen(p2.FileName)
    return l1<l2 and -1 or l1>l2 and 1 or 0
  end;
  Indicator = "bB";
})
```

SetCustomSortMode

Panel.SetCustomSortMode (Mode, whatpanel [, order])

Parameters:

Mode: sorting mode, an integer ≥ 100 and $\leq 0x7FFFFFFF$
whatpanel: 0=active panel, 1=passive panel
order: "auto" - standard choice of sort direction (default)
 "current" - keep current sort direction in the panel
 "direct" - set direct sort mode
 "reverse" - set reverse sort mode

Returns:

Nothing

Description:

If the specified sorting mode is loaded (see [Panel.LoadCustomSortM](#)) will be set in the specified panel. Otherwise, no actions will be

Example:

```
-- Set the given custom sort mode in the active panel.  
Macro {  
  description="Sort files by their name lengths";  
  area="Shell"; key="CtrlShiftF1";  
  action=function() Panel.SetCustomSortMode(100,0) end;  
}
```

BM

BM - a table with the following fields:

Properties:

None.

Functions:

[Add](#)

[Back](#)

[Clear](#)

[Del](#)

[Get](#)

[Goto](#)

[Next](#)

[Pop](#)

[Prev](#)

[Push](#)

[Stat](#)

CmdLine

CmdLine - a table with the following fields:

Properties:

Bof :	boolean
Empty :	boolean
Eof :	boolean
Selected :	boolean
CurPos :	number
ItemCount :	number
Value :	string
Result :	table, or nil

Functions:

None.

Dlg

Dlg - a table with the following fields:

Properties:

[CurPos](#): number
[Id](#): string (GUID)
Owner: string (GUID)
[ItemCount](#): number
[ItemType](#): number
[PrevPos](#): number

Functions:

[GetValue](#)
[SetFocus](#)

Drv

Drv - a table with the following fields:

Properties:

[ShowMode](#): number
[ShowPos](#): number

Functions:

None.

Editor

Editor - a table with the following fields:

Properties:

[CurLine](#): number
[CurPos](#): number
[FileName](#): string
[Lines](#): number
[RealPos](#): number
[SelValue](#): string
[State](#): number
[Value](#): string

Functions:

[DelLine](#)
[GetStr](#)
[InsStr](#)
[Pos](#)
[Sel](#)
[Set](#)
[SetStr](#)
[SetTitle](#)
[Undo](#)

Far

Far - a table with the following fields:

Properties:

FullScreen :	boolean
Height :	number
IsUserAdmin :	boolean
PID :	number
Title :	string
UpTime :	number
Width :	number

Functions:

[Cfg_Get](#) (deprecated, use [GetConfig](#))
[DisableHistory](#)
[GetConfig](#)
[KbdLayout](#)
[KeyBar_Show](#)
[Window_Scroll](#)

GetConfig

```
val, tp = Far.GetConfig (keyname)
```

Parameters:

```
keyname: string
```

Returns:

```
val:      boolean, string, number, or int64
          This is the value of the queried configuration setting.
          The type conversions between Far Manager and Lua are do
          boolean -> boolean
          3-state -> 0,1,2 are converted respectively into fals
          string  -> string
          integer -> number (if non-lossy conversion is possibl
          userdata (int64) - a value created by the

tp:      string ("boolean", "3-state", "string", "integer")
          The type of the original value in Far Manager.
```

Note:

In cases of failure (e.g an invalid argument, or Far Manager did n the specified option) this function raises an error.

Help

Help - a table with the following fields:

Properties:

[FileName](#): string

[SelTopic](#): string

[Topic](#): string

Functions:

None.

Menu

Menu - a table with the following fields:

Properties:

[Id](#): string (GUID)
[Value](#): string

Functions:

[Filter](#)
[FilterStr](#)
[GetValue](#)
[ItemStatus](#)
[Select](#)
[Show](#)

Mouse

Mouse - a table with the following fields:

Properties:

X :	number
Y :	number
Button :	number
CtrlState :	number
EventFlags :	number
LastCtrlState:	number

Functions:

None.

Note:

Mouse.LastCtrlState differs from *Mouse.CtrlState* by that its value on both mouse events and keyboard events.

Object

Context-dependent values.

Object: - a table with the following fields:

Properties:

Bof :	boolean
CurPos :	number
Empty :	boolean
Eof :	boolean
Height:	number
ItemCount :	number
Selected :	boolean
Title :	string
Width:	number

Functions:

[CheckHotkey](#)
[GetHotkey](#)

Plugin

Plugin - a table with the following fields:

Properties:

None.

Functions:

[Call](#)

[Command](#)

[Config](#)

[Exist](#)

[Load](#)

[Menu](#)

[SyncCall](#)

[Unload](#)

Call

... = [Plugin.Call](#) (PluginId [, ...])

Parameters:

PluginId: string (plugin's GUID in textual representation)
... : zero or more additional parameters

Returns:

... : zero or more return values

Description:

1. The function implements the "asynchronous" plugin call.
If the plugin call turned out to be asynchronous (e.g. if the plugin opens a dialog on the screen) then the function returns *true* without further return, and the macro continues its execution.
2. If the plugin call turned out to be synchronous, the macro gets the values returned by the plugin:
 - If the plugin is not found or returned 0, then *false* is returned.
 - If the plugin returned 1 or `INVALID_HANDLE_VALUE`, then *true* is returned.
 - If the plugin returned a pointer to a [FarMacroCall](#) structure,
3. Arguments are passed to the plugin in accordance with their Lua type:
 - nil -> [FMVT NIL](#)
 - boolean -> [FMVT BOOLEAN](#)
 - number -> [FMVT DOUBLE](#)
 - int64 -> [FMVT INTEGER](#) (int64 - kind of userdata, created by the plugin)
 - string -> [FMVT STRING](#) (automatically converted from UTF-8 to UTF-16LE)
 - {string} -> [FMVT BINARY](#) (in order to pass an arbitrary string, the string should be placed in a table with its key==1)
4. Values returned by the plugin via a [FarMacroCall](#) structure are returned in the following way:
 - [FMVT NIL](#) -> nil
 - [FMVT BOOLEAN](#) -> boolean
 - [FMVT DOUBLE](#) -> number
 - [FMVT INTEGER](#) -> number, if it "fits" in 53 bits, otherwise in a table with key=1
 - [FMVT STRING](#) -> string (automatically converted from UTF-16LE to UTF-8)
 - [FMVT BINARY](#) -> table (the table contains a string as an element with key=1; the string is placed as is, without conversion)
 - [FMVT POINTER](#) -> light userdata
 - [FMVT ARRAY](#) -> table (array of elements; the table contains the elements with keys 1..n)

```
["type"] = "array", and  
["n"] = number of array elements)
```

See also:

[Restrictions in the use of some functions](#)

SyncCall

```
... = Plugin.SyncCall (PluginId [, ...])
```

This function works identically to [Plugin.Call](#), except that:

1. Its call is always synchronous, i.e. the macro continues its execution only after the [OpenW](#) function of the plugin returns.
2. This function does not have [limitations](#) that [Plugin.Call](#) has.

Viewer

Viewer - a table with the following fields:

Properties:

[FileName](#): string
[State](#): number

Functions:

None.

Misc

Macros beginning with @

If a macrosequence begins with the @ character, then the rest of the sequence is treated as name of the file containing Lua script.

- Environment variables in the file name are automatically expanded.
- The global (within the environment of the script) variable `_filename` contains the file name.
- Such scripts are easier for debugging and modifications, as they are automatically reloaded at each macro invocation.

Example 1:

```
lua: @%FARHOME%\test\test.lua 123, "hello"
```

Example 2: [running script from within the editor.](#)

Variables

Macrofile environment variables

Macros that are loaded from the same file share a common environment table. The variables declared without the *local* keyword belong to that environment.

Example: `var = 15`

The macrofile's environment variables keep their values unchanged between macro calls. Their values are reset to initial state upon execution of any macro loading operation: Far Manager start, `macro:load`, `lm:load`, `far.MacroLoadAll`, [MacroControl](#)([MCTL_LOADALL](#)).

Global variables

To set global variables, whose values are stored during the whole Far Manager session and are accessible from any script, one should use the `_G` table.

Example: `_G.var = 15`

Global variables do not change their values even when macros are reloaded, except for Far Manager restart or LuaMacro plugin reload.

When reading a non-existent environment variable, a same-named global variable can be read instead.

```
Example:  
var = 5  
_G.var = 10  
far.Message(var) --> 5  
var = nil  
far.Message(var) --> 10
```

Upvalues

Top-level local variables accessible from functions of one or several macros (upvalues) keep their values unchanged between macro calls. Their values are

reset upon execution of any macro loading operation.

Example:

```
local var = 15  
function inc_var() var = var+1 end  
function dec_var() var = var-1 end
```

Restrictions in the use of some functions

The following functions have certain restrictions on their use in macros:

- [exit](#)
- [mf.acall](#)
- [mf.AddExitHandler](#)
- [mf.usermenu](#) (in the “asynchronous” call mode)
- [Keys](#)
- [Plugin.Call](#)
- [Plugin.Command](#)
- [Plugin.Config](#)
- [Plugin.Menu](#)
- [print](#)

1. If a macro creates coroutines with `coroutine.wrap(f)`, then the above listed functions will not work when called from the body of `f` function. This restriction does not exist if coroutines are created with `coroutine.create(f)`.
2. The above listed functions, when called directly or indirectly with `pcall`, will cause the immediate failure of `pcall`. This restriction does not exist if LuaJIT 2.x is used.
3. The above listed functions will only work when called from a **macro body** (usually it is function *action*). That means those functions will not work when called from:
 - dialog procedures
 - function *condition* of a macro
 - [event handlers](#)
 - etc. etc.

The restrictions of p.3 do not exist, if the above listed functions are called via [mf.postmacro](#) or `far.MacroPost`.

Introspection

The global tables of macro API can be examined with `pairs()`, separately for functions and “properties”.

Example:

```
for k,v in pairs(Editor) do .... end -- for functions
```

```
for k,v in pairs(Editor.properties) do .... end -- for properties
```

unicode.utf8.cfind

This function is similar to `unicode.utf8.find`, except that it treats the input offset and returns the output offset in characters rather than bytes. “Positional captures” are still returned in bytes.

editor.SubscribeChangeEvent

LuaFAR library contains the function `editor.SubscribeChangeEvent`. This function is redefined by [LuaMacro](#) plugin for use by [event handlers](#):

- The original function is called only when the internal *subscription counter* changes from 0 to 1 (if argument `Subscribe==true`), or from 1 to 0 (if argument `Subscribe==false`). It is therefore necessary that every event handler containing a call `Subscribe==true`, contained also a matching call `Subscribe==false`.
- Unlike plugins, the [EE_CHANGE](#) event could come to an event handler regardless of whether that handler called `editor.SubscribeChangeEvent`.
- There is a separate subscription counter for each editor session.

package.nounload

`package.nounload` is a table whose keys are module names, that should not be deleted from `package.loaded` when macros are unloaded or reloaded (operations [MCTL_LOADALL](#), `lm:load`, `lm:unload`).

- This table is created automatically by [LuaMacro](#) plugin.
- A use case: place in this table names of modules using LuaJIT FFI that call `ffi.cdef`.

For example, after executing

```
package.nounload.mylib = true
```

the module `mylib` will not be removed from `package.loaded` during macro unload/reload operations.

Examples

Select the word under cursor

```
Macro {
  description="Select/deselect the word under the cursor";
  area="Editor"; key="CtrlM";
  action=function()
    Keys"RCtrl9 CtrlRight CtrlLeft"
    Keys(Object.Selected and "CtrlU" or "CtrlShiftRight")
    Keys"Ctrl9"
  end;
}
```

Invoke the "change drive" menu

```
-- Invoke the "change drive" menu on the opposite panel.
-- Make the panel where drive change occurs visible if it was not.
Macro {
  description="Invoke the 'change drive' menu on the opposite panel"
  area="Disks"; key="CtrlM";
  action=function()
    Keys"Esc"
    if not PPanel.Visible then
      Keys(APanel.Left and "CtrlF2" or "CtrlF1")
    end
    Keys"Tab F9 Enter End Enter"
  end;
}
```

Select files newer than current one

```
-- Select all files newer than the current one in the active panel,
-- using plugin API (LuaFAR).
Macro {
  description="Select all files/folders newer than the current one i
  area="Shell"; key="CtrlM";
  action=function()
    local info = panel.GetPanelInfo(nil,1)
    local curItem = panel.GetCurrentPanelItem(nil,1)
    for i=1,info.ItemsNumber do
      local item = panel.GetPanelItem(nil,1,i)
      if item.LastWriteTime > curItem.LastWriteTime then
        panel.SetSelection(nil,1,i,true)
      end
    end
    panel.RedrawPanel(nil,1)
  end;
}

-- Select all files newer than the current one in the active panel,
-- using macro API (LuaFAR + LuaMacro).
Macro {
  description="Select all files/folders newer than the current one i
  area="Shell"; key="CtrlM";
  action=function()
    d = Panel.Item(0,0,17)
    for i=1,APanel.ItemCount do
      if Panel.Item(0,i,17) > d then
        Panel.Select(0,1,1,i)
      end
    end
  end;
}
```

Create a directory with name = current date

```
Macro {
  description="Create a directory with name = current date";
  area="Shell"; key="CtrlShiftF7"; flags="NoPluginPanels";
  action=function()
    folder = mf.date("%d.%m%Y")
    if Panel.FExist(0, folder)==0 then
      Keys"F7 CtrlY"
      print(folder)
      Keys"Enter"
    end
  end;
}
```

Running script from within the editor

```
-- This macro saves the editor contents (if it was modified)
-- then runs the edited file as Lua-script.
Macro {
  description="Save and run script from editor";
  area="Editor"; key="CtrlF10";
  action=function()
    for k=1,2 do
      local info=editor.GetInfo()
      if bit64.band(info.CurState, far.Flags.ECSTATE_SAVED)~=0 then
        local Flags = info.FileName:sub(-5):lower()==".moon"
          and "KMFLAGS_MOONSCRIPT" or "KMFLAGS_LUA"
        far.MacroPost('@"' .. info.FileName .. "'", Flags)
        break
      end
      if k==1 then editor.SaveFile(); end
    end
  end;
}
```

Third party software used

Lua

Lua is licensed under the terms of the MIT license reproduced below. This means that Lua is free software and can be used for both academic and commercial purposes at absolutely no cost.

For details and rationale, see <http://www.lua.org/license.html> .

Copyright © 1994-2008 Lua.org, PUC-Rio.

Permission is hereby granted, free of charge, to any person obtaining a copy of this software and associated documentation files (the “Software”), to deal in the Software without restriction, including without limitation the rights to use, copy, modify, merge, publish, distribute, sublicense, and/or sell copies of the Software, and to permit persons to whom the Software is furnished to do so, subject to the following conditions:

The above copyright notice and this permission notice shall be included in all copies or substantial portions of the Software.

THE SOFTWARE IS PROVIDED “AS IS”, WITHOUT WARRANTY OF ANY KIND, EXPRESS OR IMPLIED, INCLUDING BUT NOT LIMITED TO THE WARRANTIES OF MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE AND NONINFRINGEMENT. IN NO EVENT SHALL THE AUTHORS OR COPYRIGHT HOLDERS BE LIABLE FOR ANY CLAIM, DAMAGES OR OTHER LIABILITY, WHETHER IN AN ACTION OF CONTRACT, TORT OR OTHERWISE, ARISING FROM, OUT OF OR IN CONNECTION WITH THE SOFTWARE OR THE USE OR OTHER DEALINGS IN THE SOFTWARE.

LPeg

License

Copyright © 2008 Lua.org, PUC-Rio.

Permission is hereby granted, free of charge, to any person obtaining a copy of this software and associated documentation files (the “Software”), to deal in the Software without restriction, including without limitation the rights to use, copy, modify, merge, publish, distribute, sublicense, and/or sell copies of the Software, and to permit persons to whom the Software is furnished to do so, subject to the following conditions:

The above copyright notice and this permission notice shall be included in all copies or substantial portions of the Software.

THE SOFTWARE IS PROVIDED “AS IS”, WITHOUT WARRANTY OF ANY KIND, EXPRESS OR IMPLIED, INCLUDING BUT NOT LIMITED TO THE WARRANTIES OF MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE AND NONINFRINGEMENT. IN NO EVENT SHALL THE AUTHORS OR COPYRIGHT HOLDERS BE LIABLE FOR ANY CLAIM, DAMAGES OR OTHER LIABILITY, WHETHER IN AN ACTION OF CONTRACT, TORT OR OTHERWISE, ARISING FROM, OUT OF OR IN CONNECTION WITH THE SOFTWARE OR THE USE OR OTHER DEALINGS IN THE SOFTWARE.

MoonScript

License (MIT)

Copyright © 2013 by Leaf Corcoran

Permission is hereby granted, free of charge, to any person obtaining a copy of this software and associated documentation files (the “Software”), to deal in the Software without restriction, including without limitation the rights to use, copy, modify, merge, publish, distribute, sublicense, and/or sell copies of the Software, and to permit persons to whom the Software is furnished to do so, subject to the following conditions:

The above copyright notice and this permission notice shall be included in all copies or substantial portions of the Software.

THE SOFTWARE IS PROVIDED “AS IS”, WITHOUT WARRANTY OF ANY KIND, EXPRESS OR IMPLIED, INCLUDING BUT NOT LIMITED TO THE WARRANTIES OF MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE AND NONINFRINGEMENT. IN NO EVENT SHALL THE AUTHORS OR COPYRIGHT HOLDERS BE LIABLE FOR ANY CLAIM, DAMAGES OR OTHER LIABILITY, WHETHER IN AN ACTION OF CONTRACT, TORT OR OTHERWISE, ARISING FROM, OUT OF OR IN CONNECTION WITH THE SOFTWARE OR THE USE OR OTHER DEALINGS IN THE SOFTWARE.

Articles

FAR Manager. Macros and whatnot.

By **Gleb Varenov** (“Acerbic”) » Tue 23 Apr, 2013 22:53

[Revision 1.1](#)

Gather round, boys and girls, for I am about to tell you a story.

Macro in your FAR Manager? It's more likely than you think.

The “macro” term is used loosely in Far to describe a broad range of user-made modifications of the application's behavior. For the sake of simplicity, let me divide them into several use cases:

1. Assigning **hotkeys** / remapping key bindings. This is the simplest. You want one custom key (or combination with Ctrl, Shift, Alt modifiers) do what another key (combination) already does. It is one for one replacement.
2. Your typical **macro**. Press one key to emulate a sequence of many keys. Its not much different from just a hot key. You press a key – it is treated as if you pressed some fixed predefined string of keys.
3. **Script**. Now this is interesting. Script involves logic and decision making, not just some fixed reaction. A script has means to analyze current situation and affect Far Manager in non-trivial ways. Requires programming skills to create, but luckily the are many scripts already written and you may find one just fitting your needs.
4. **Plugin**. Well, this is a moot point. With the course Far development is following now the gap between a complex script and a full fledged plugin is closing rapidly. As of right now, a script has most of functional capabilities a plugin has, it has access to both macro API and plugin API, in the latest builds a macro script can be started by an event, not just by a key pressed and so forth...

History and identification of obsolete.

FAR Manager has traveled a long way from its early versions and so did its macro capabilities. In Far1 and Far2 the configuration was stored in Windows registry, including macros. In Far3 configuration was moved to SQLite database files located in user's profile folder. But later it was decided that it would be better to allow users simpler access to macros and they were moved from a database to individual files (still inside user's profile folder). Also, parallel to moving macro definitions to external files, the macro language was changed to LUA.

In addition to that, there is a very popular plugin called "MacroLib". It provides extended macro features on top of built-in system. It used to overlay old macro language, but then switched to LUA as well. It always stored macros in external files.

So, what to look out for to spot outdated manuals / macro recipes?

1. *.reg files. Partial and full configurations were distributed as reg import files in times of Far1 and Far2. That included macros. No *.reg files are used by Far3 plugins or Far3 itself, but some old (ANSI) Far1 plugins can still use them. Any macros contained in them won't work in Far3.
2. *.farconfig files. Those are XML text files containing configurations for Far3. They are still used for other parts of configuration (coloring schemes and such) but not for macros anymore.
3. Old macro language. It contained keywords like "\$IF" "\$ELSE" – denoted by dollar sign.
4. Old MacroLib files *.fml – new MacroLib macro files use "fmlua" extension.

It is important to note that internal help in Far (called by "F1") is massively lagging behind development – hence this article.

What now?

At present, by means of storage macros fall in three groups.

1. Files of the built-in macro system. *.lua Located in [%FARPROFILE%/Macros/internal](#) and [%FARPROFILE%/Macros/scripts](#)
2. MacroLib files *.fmlua. By default are located within plugin's folder, but it can be configured to read macros from any user defined path (or several).
3. Other macro processing plugins. "Lua4Editor", for example. I don't know much about these, you are on your own here, folks.

Important warning! There are plugins written in LUA. These are something different from macros. They are legit plugins with all the things a "usual" plugin has (like being listed in plugins' menu "F11"). Except they are written in LUA and distributed as source files. They should not confuse you as they lie in their folders in [%FARHOME%/Plugins](#)

Conversion.

It is best to rewrite your macros in LUA from scratch. If they are few and simple, it won't take much effort. If they are complex, conversion is likely to fail to do it automatically. But if you are still interested in doing things hard way, there are few tools to help you. They were meant as quick fixes for transition period and aren't supported anymore, probably.

1. Far1, Far2 → Far 3 [2x3 perl converter](#) This tool is used to convert old configuration from registry to database. This includes macros. The result will be a bunch of XML files containing far configurations and macros in the old language.
2. Translation from old language to the new one is done by [Macro2Lua Converter](#) plugin. The readme is in Russian, but here is an excerpt regarding main usage via command line

```
M2L: convert <input file> <output file> [<syntax>]
```

where <syntax> is optional input format specifier and is one of following: xml_file, xml_macros, xml_keymacros, xml_macro, fml_file, fml_macro, chunk, expression. General file format is the part before underscore, the specific section of a file is the part after underscore. “chunk” and “expression” are some kind of raw macro pieces of texts. The result should be a XML file (<farconfig>... </farconfig>) with macros translated to LUA inside of it or a MacroLib fmlua file if original was *.fml and corresponding syntax was specified.

3. Far3 2927-3000 → Far3 3001+ Now you need an older version of Far3 (pre3001). You import your macros to Far per usual command, then use a script provided in the following forum thread: [Macros have been moved from macros.db to files](#)

Managing confusion.

While all macros are written in LUA, file contents are not interchangeable as of right now. It means that you can't rename X.fmlua to X.lua, move it to [%FARPROFILE%/Macros/scripts](#) and expect it to work. Likewise you can't just move files from [/internal](#) to [/scripts](#).

Luckily, with few rules I am about to explain, you won't get lost in all of the LUA files lying around.

1. Don't touch your internals! Files in [%FARPROFILE%/Macros/internal](#) are to be manipulated (created/edited/deleted) by FAR Manager itself. And while it is possible to edit them manually, better to leave them alone. Unless you really know what are you doing. Or just feeling adventurous.
2. MacroLib files are always named *.fmlua, so you can never confuse them with native script files, even if you configured the MacroLib so they are located in the same directory.
3. User-made native script files are located in [%FARPROFILE%/Macros/scripts](#) and are named *.lua. They are read at Far launch, but you can make Far to re-read the folder via command line (more on that later).

So, only (2) and (3) are in user's management and they are different in names, location and internal structure.

But there are similarities too.

1. Both MacroLib and native macros have a concept of "Area of execution" – basically, a broad condition limiting macro effect. Typical are "Editor" – when editor is open, "Shell" – when file panels are in focus, etc...
2. In addition to general area, some more conditions might be specified for activation of a macro. Like passive panel being visible, command line not being empty and similar. These conditions/flags are legacy carried over from times when macro language was primitive and things like that were hard to check in script itself. Alternatives are being developed (like custom

function conditionals in native scripts) but there is no sign that old flags will be abandoned yet.

3. And finally, there are two flags that control execution of the macro itself. One is to disable/enable intermediate visual output during macro execution (reduces flicker of menus and dialogs being open/closed, for example), another is to control if plugins can intercept keyboard events generated by macro. No other macro can intercept current macro while it is executing – so you don't have to worry about nasty macro interferences.

Out of the box.

In the beginning of time the macro language was ugly and everyone was sad. And few of the developers raised their voices: “Look! There in the great outside lies shiny LUA. Let us take it for ourselves, let us bind it to our manager and then we won’t be suffering dollar-signed keywords no more.” And so they did. And night turned day, and day turned night, and the Moon died and was born again as they tinkered and meddled and compiled and debugged. Seasons passed by, but finally, the day has come and their labor was over. And they stood proud among men and shouted: “Behold this LuaMacro plugin! We can rework our ugly macros into LUA, we have the technology now. But wait! There’s more: we can write plugins in LUA as well, if we desire so.” And everyone rejoiced. And gathered developers of Far and saw what their brethren did, and saw that it was good. So good in fact, they put the new plugin in the core package and abandoned their old ways of macros.

So, native macro capabilities are provided by LuaMacro plugin, which is distributed with Far itself as part of its core package. The plugin has no configuration dialog, but has a list of commands to manipulate it:

- `lm: unload` – Far forgets all macros. They are still on disk and can be loaded back with next command.
- `lm: load` – makes Far discard all macros and then re-read them from directories anew.
- `lm: post <sequence>|@<filename>` – executes a macro code immediately. Either a “raw” piece of code typed in command line, or same raw code saved in a file. File name is prefixed with “@” symbol.
- `lm: check <sequence>|@<filename>` – same as above. Except the macro is not executed but checked for syntax errors.
- `lm: save` – saves changes made to “internal” macro files. Useful if you have “Auto save setup” option turned off.

When “load” and “unload” operations are concerned, only native macros are affected. I.e. those *.lua that are located in [/internal/](#) and [/scripts/](#). MacroLib

macros are not touched. List of all currently loaded native macros is available in Far built-in help “F1”. That part of help is not translated to English yet, and its not very convenient in operation anyway. Check [this](#) macro out though.

Note. “lm:” commands are similar to ones provided by FarCommands plugin via “macro:” and “far:macro ” prefixes. There was a difference in that FarCommands used “<” symbol to specify filename, but now it supports both “<” and “@” for this.

It is time now to explain why some macros are put in [/internal/](#) and others in [/scripts/](#). Its fairly simple – “internal” is a codename for “recorded” and all the recoded macros go there. More on recorded macros is in the “[Hotkeys / Macro use case](#)” chapter. User-made macros are to be placed in [/scripts/](#). Sadly, there’s no comprehensible manual on how to write them. One is reduced to scavenging for bits and pieces of knowledge by perusing Far’s changelog and dissecting macros written by others ([SimSU macro pack](#) for example, topic in Russian forum: <http://forum.farmanager.com/viewtopic.php?f=15&t=7075>). Here is a script for the Editor that pastes a macro template on “Ctrl+F11” by Shmuel: [InsertMacro.lua.7z](#)

MacroLib.

This is what all the cool kids use. MacroLib is a plugin that provides somewhat extended functionality to macros. It is built on top of native macro system, so 99% of the code working for “regular” macro will work for MacroLib as well. Project’s main page: <http://code.google.com/p/far-plugins/wiki/MacroLib>, download page: <http://code.google.com/p/far-plugins/downloads/list?q=MacroLib>, documentation (Russian): <http://code.google.com/p/far-plugins/wiki/FML>. MacroLib files are named *.fmlua and are located in one or several directories designated by user in configuration dialog. The dialog allows you to update macros from disk and shows you a very neat list of all macros currently loaded with ability to sort, filter, run a macro from the list.

MacroLib used to have many advantages over built-in system, but nowadays Far has caught up for the most part and is ahead in some experimental things (like events). However, there are two things *.fmlua scripts have over *.lua ones.

1. You can use modifiers to your assigned hotkeys, such as “Hold” (macro is invoked after the key was held for a certain period of time), “Double”(on double click or double key tap), “Release” (macro is called on key being released, rather than being pressed).
2. You frame your macro code in double curly brackets for extra swag {{ }}.

Hotkey / Macro use case.

Can't get used to saving edited file by "F2"? Too lazy to run through menus every time you want to view your current Folder Shortcuts? Then this chapter is for YOU. This chapter covers a very simple usage of Far macros – redefining hotkeys for existing actions and creating hotkeys for actions that don't have them by default. The easiest way to do said things is by using "Recorded Macro" feature. The Far Manager has an ability to record your actions (keyboard events) and assign them to a specific key, pressing which will replay your actions. This function is in there from Far1 and is explained in "F1" Help, but I will rehash it for you anyway. You start recording by pressing "Ctrl+." (Control key plus dot key) or "CtrlShift+.", a little red "R" letter appears in the top-left corner and your following key presses will be recorded. You continue to use Far as usual, doing things you want to be put in the macro, or just pressing one key you want to be remapped. Then you finish recording by pressing "Ctrl+." or "CtrlShift+." again. Then you will be asked for a key to which this macro will be assigned, you can select one from a drop-down list with a mouse or just press the desired combination, then "Enter". At this moment an optional dialog might appear to configure additional parameters of you macro.

So,

1. "Ctrl+." or "CtrlShift+."
2. Do stuff on record.
3. "Ctrl+." or "CtrlShift+."
4. Select a desired key to assign to.
5. (optional) Configuration dialog.

If you finish recording with "CtrlShift+." on step 3 you will be shown a dialog on step 5. Otherwise you won't be. To know more about this dialog, press "F1" while in there, its covered in the Help. Starting the recording with "CtrlShift+." puts a "NoSendKeysToPlugins" flag on your macro, which means that during macro playback plugins won't be able to react to keyboard events generated by this macro – it forces plugins to ignore this macro in that regard.

If you made a mistake in your macro during recording you may interrupt the recording by usual "Ctrl+." and then hit "Esc" when asked about desired key. If you select a key that is already taken by a macro, you will be asked if you want

to overwrite previous macro. This means you cannot have two recorded macros on the same key in the same area of execution. You can, however, have one for each area (one in Editor, one in Viewer, etc.) If you need to delete a macro you previously recorded, you create an empty macro for the key you want to free: “Ctrl+.”, “Ctrl+.”, the key. Then select “yes” to confirm deletion.

If you have “F9”→“Options”→“System Settings”→“Auto save setup” option turned on, then every change to your recorded macros (creation, modification, deletion) will be immediately saved to files. Otherwise you can use `lm: save` command to save your recorded macros or press “Shift+F9” to save full setup. If you want to know what macros are already recorded, you can navigate to [%FARPROFILE%/Macros/internal](#). It is possible delete macros in there, just don't forget to use `lm: load` command to update, or restart Far.

Example: lets bind a hotkey to “File associations” menu.

1. (preparation) Make sure you are in the Shell area of Far, its where two panels with files and folders are.
2. Press “Ctrl+.” and make sure the red “R” letter appeared.
3. Press “F9” to move input focus to Far’s menu (usually is the top line of the window), then “c” for commands, then “a”. Now, if done right the “File associations” menu is on screen.
4. Press “Ctrl+.” again, a little "Define macro" box should pop up. Press “Ctrl+Shift+a” and confirm that corresponding key code appeared in the box (“CtrlShiftA”).
5. Press “Enter” and enjoy a new quality of life improvement you just created for yourself. Now every time you press “Ctrl+Shift+a” combo in Far shell, the menu will instantly appear.

Script use case.

“Script” is a program that runs within/by other program (as opposed to one run by CPU/OS). Scripts in Far evolved from macros to a point when macros themselves are considered primitive cases of scripting. Being programs, scripts require “Programming / Coding” skill to be created, therefore, if you intend to use Far to its fullest potential you might want to invest few skill points in it on your next level up. Alternatively, you can utilize macros written by someone else – just copy the files in appropriate folders. As mentioned before, Far uses LUA language for scripting. From within the script you have access to

1. Far (plugin) API – functions of Far that are available to plugins.
2. Far macro API – some specific functions that were available in old language. These overlap “Plugin API” to some extent and considered legacy API. Better use “Plugin API” where possible.
3. Custom functions exported by plugins – some plugins export their functions to be called from macro. Those depend on plugin being installed and loaded, of course.
4. LUA libraries – native to LUA (see language manual) plus few libraries additionally shipped with LuaFar (“bit64”, “win” – gate to Win API, Selene Unicode)
5. Far UI – you can control Far simply by issuing keyboard/mouse commands to it. Why bother finding a function that will open Editor for file under cursor when you can just send “F4” to Far?

Your main source of information about Far APIs is in [%FARHOME%\Encyclopedia](#) files. Lets look at them.

- “FarEncyclopedia.ru.chm” – includes (1) and (2), in Russian. Macro API is outdated (pre-LUA). There’s an online version too: <http://api.farmanager.com/ru/>
- “luafar_manual.chm” – originally a LuaFar plugin manual (writing plugins in LUA), but we can use it in scripts too. Covers (1) in LUA in English.

Very spartan – most of the functions have no textual descriptions, only input parameters and result values (implies ability to read “FarEncyclopedia.ru.chm”). For the most part it is not a problem though, functions' names are self-descriptive.

- “macroapi_manual.chm” – mapping of (2) to LUA. Again, almost no descriptions.

Damn, its kinda depressing, ain't it? Luckily for you, I have a magical artifact that will allow you to understand Russian: [abracadabra](#). Paste a link to Russian website or text fragment and hit “Enter”. And Acerbic saves the day once again! You are welcome.

To sum it up: you will use “luafar_manual.chm” in conjunction with [translated online encyclopedia](#) for Far plugin API reference and “macroapi_manual.chm” in conjunction with [this link](#) for Macro API reference. I found this script very helpful: [lua explorer](#). It allows you to browse Lua tables/values/functions soup available to LUA script. [Thread](#) on the forum.

Native or MacroLib?

MacroLib.

Sample script.

Here's a little demonstration of what you can do in MacroLib: "RCtrl Folder shortcuts.fmlua"

```
;;
;; Folder shortcuts menu
;; RCtrl single pressing or holding will pop-up the shortcuts menu.
;; RCtrl1-0 will go to set shortcut
;;

const FolderShortcutsId = "4CD742BC-295F-4AFA-A158-7AA05A16BEA1"

macro
area="Shell"
description="Folder shortcuts popup"
key="RCtrl:Hold RCtrl:Release" ;; call on holding LCtrl or single p
EatOnRun=0 ;; allows RCtrl:Release in Menu area after RCtrl:Hold was
{{
    Keys("F9 c d");
}}

macro
area="Menu"
description="Folder shortcuts: RCtrl+digit"
key="/RCtrl\d/"
{{
    if (Menu.Id == #%FolderShortcutsId) then
        Keys(regex.match(akey(1), "RCtrl(\\d)?")) -- double escaping
    end;
}}

macro
area="Menu"
description="Folder shortcuts: pass RCtrl+not_digit through"
key="/RCtrl(?!\\d$).+/" ;; Takes RCtrlSOMETHING. Ignores RCtrl0
{{
    if (Menu.Id == #%FolderShortcutsId) then
        Keys("Esc AKey");
    end;
}}

macro
```

```
area="Menu"  
description="Folder shortcuts popup close on second RCtrl or on RCtrl  
key="RCtrl:Release"  
{  
    if (Menu.Id == #FolderShortcutsId) then  
        Keys("Esc");  
    end;  
}
```