
IUP

Portable User Interface

Version 2.2

(iup@tecgraf.puc-rio.br)

IUP is a portable toolkit for building graphical user interfaces. It offers a configuration API in three basic languages: C, Lua and LED. **IUP**'s purpose is to allow a program to be executed in different systems without any modification, therefore it is highly portable. Its main advantages are:

- high performance, due to the fact that the system uses native interface elements
- fast learning by the user, due to the simplicity of its API.

This work was developed at Tecgraf/PUC-Rio by means of the partnership with PETROBRAS/CENPES.

The **IUP** Team:

Antonio Escaño Scuri
Mark Stroetzel Glasberg

Tecgraf - Computer Graphics Technology Group, PUC-Rio, Brazil
<http://www.tecgraf.puc-rio.br/iup>

Overview

IUP is a portable toolkit for building graphical user interfaces. It offers APIs in three basic languages: C, [Lua](#) and [LED](#).

Its library contains about 100 functions for creating and manipulating dialogs.

IUP's purpose is to allow a program to run in different systems without changes - this toolkit provides the applications a high portability. Supported systems include: Motif, Microsoft Windows 98, Microsoft Windows NT, Microsoft Windows 2000 and Microsoft Windows XP.

IUP uses an abstract layout model based on the boxes-and-glue paradigm from the T_EX text editor. This model, combined with the dialog-specification language ([LED](#)) or with the Lua binding ([IupLua](#)) makes the dialog creation task more flexible and independent from the graphics system's resolution.

Currently available interface elements can be categorized as follows:

- **Primitives** (effective user interaction): **dialog, label, button, text, multi-line, list, toggle, canvas, frame, image.**
- **Composition** (ways to show the elements): **hbox, vbox, zbox, fill.**
- **Grouping** (definition of a common functionality for a group of elements): **radio.**
- **Menu** (related both to menu bars and to pop-up menus): **menu, submenu, item, separator.**
- **Extended** (additional elements built outside the library): **dial, gauge, matrix, tabs, valuator, GL canvas, color chooser, color browser, toolbar.**
- **Dialogs** (useful predefined dialogs): **file selection, message, alarm, data input, list selection.**

Hence IUP has some advantages over commercial interface toolkits available in the industry:

- **Simplicity:** due to the small number of functions and to its attribute mechanism, the learning curve for a new user is often faster.
- **Portability:** the same functions are implemented in each one of the platforms, thus assuring the interface system's portability.
- **Customization:** the dialog specification language (LED) and the Lua binding (IupLua) are two mechanisms in which it is possible to customize an application for a specific user with a simple-syntax text file.
- **Flexibility:** its abstract layout mechanism provides flexibility to dialog creation.
- **Extensibility:** the programmer can create new interface elements as needed.

IUP is free software, can be used for public and commercial applications.

Availability

The library is available for several **compilers**:

- GCC and CC, in the UNIX environment
- Visual C++, Borland C++, Watcom C++ and GCC (Cygwin and MingW), in the Windows environment

The library is available for several **operating systems**:

- UNIX (SunOS, IRIX, AIX and Linux)
- Microsoft Windows NT/2K/XP

Support

The official support mechanism is by e-mail, using **iup AT tecgraf.puc-rio.br** (replace " AT " by "@"). Before sending your message:

- Check if the reported behavior is not described in the user guide.
- Check if the reported behavior is not described in the specific format characteristics.
- Check the History to see if your version is updated.
- Check the To Do list to see if your problem has already been reported.

If all these points were checked, you can report your problem. Please specify in your message: **function**, **attribute**, **callback**, **platform** and **compiler**.

Announcements of new versions are done by the read only list **iup-l AT tecgraf.puc-rio.br** (replace " AT " by @). Send a request to the support e-mail to be added or removed from the list.

Credits

This work was developed at Tecgraf by means of the partnership with PETROBRAS/CENPES.

People who took part in IUP's development:

André Carregal
 André Costa
 André Derraik
 Antonio Scuri
 Carlos Augusto Mendes
 Carlos Henrique Levy
 Carlos José Pereira de Lucena
 Claudio Coutinho de Biasi
 Danny Reinhold
 Diego Nehab
 Diogo Martinez
 Enio Emanuel Russo
 Guilherme Fonseca Alvarenga
 Henrique Dalcin Mendes Pinheiro
 Leonardo Constantino Oliveira
 Luiz Henrique de Figueiredo
 Marcelo Gattass
 Mark Stroetzel Glasberg
 Mauricio Oliveira Carneiro
 Milton Jonathan
 Neil Armstrong
 Renato Borges
 Renato Cerqueira
 Roberto Beauclair
 Vinicius Almendra

We must also mention engineer Enio Emanuel Russo, from PETROBRAS, who effectively contributed to the system's specification and project.

The initial version of the present document was developed by Carlos Henrique Levy, Neil Armstrong and André Carregal, being supervised and oriented by Luiz Martins, Luiz Henrique de Figueiredo, Marcelo Gattass and Carlos José Pereira de Lucena at Tecgraf, PUC-Rio for the Data Processing Sector (SEPROC) at CENPES/PETROBRAS.

Documentation

This toolkit is available at <http://www.tecgraf.puc-rio.br/iup>.

The full documentation can be downloaded from the [Download](#) by choosing the "Documentation Files" option.

The documentation is also available in Adobe Acrobat ([iup.pdf](#) ~1.0Mb) and Windows HTML Help ([iup.chm](#) ~900Kb) formats.

The HTML navigation uses the WebBook tool, available at <http://www.tecgraf.puc-rio.br/webbook>.

Publications

This product stimulated the following scientific publications:

- Levy, C. H.; Figueiredo, L. H.; Gattass, M.; Lucena, C.; and Cowan, D. "IUP/LED: A Portable User Interface Development Tool". *Software: Practice & Experience*, 26 #7 (1996) 737-762. [[spe95.pdf](#)]
- Levy, C. H. "IUP/LED: Uma Ferramenta Portátil de Interface com Usuário". M.Sc. dissertation, Computer Science Department, PUC-Rio, 1993. [[levy93.pdf](#)]
- Figueiredo, L. H.; Gattass, M.; and Levy, C.H. "Uma Estratégia de Portabilidade para Aplicações Gráficas Interativas". Proceedings of VI SIBGRAPI (1993), 203-211. [[sib93.pdf](#)]
- Oliveira Prates, R.; Figueiredo, L. H.; and Gattass, M. "Especificação de Layout Abstrato por Manipulação Direta". Proceedings of VII SIBGRAPI (1994), 165-172. [[sib94.pdf](#)]
- Oliveira Prates, R.; Gattass, M. ;and Figueiredo, L. H. "Visual LED: uma ferramenta interativa para geração de Interfaces gráficas". M.Sc. dissertation, Computer Science Department, PUC-Rio, 1994. [[prates94.pdf](#)]

Tecgraf Library License

This product is free software: it can be used for both academic and commercial purposes at absolutely no cost. There are no royalties or GNU-like "copyleft" restrictions. It is licensed under the terms of the [MIT license](#) reproduced below, and so is compatible with [GPL](#) and also qualifies as [Open Source](#) software. It is not in the public domain, Tecgraf and Petrobras keep its copyright. The legal details are below.

The spirit of this license is that you are free to use the library for any purpose at no cost without having to ask us. The only requirement is that if you do use it, then you should give us credit by including the copyright notice below somewhere in your product or its documentation. A nice, but optional, way to give us further credit is to include a Tecgraf logo in a web page for your product.

The library is designed and implemented by a team at Tecgraf/PUC-Rio in Brazil. The implementation is not derived from licensed software. The library was developed by request of Petrobras. Petrobras permits Tecgraf to distribute the library under the conditions here presented.

Copyright © 1994-2004 [Tecgraf](#) / [PUC-Rio](#) and [PETROBRAS S/A](#).

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.

History of Changes

Version 2.2.2 (07/Oct/2004)

General

- Fixed bug in IupGetFile FILTER initialization.
- Improved IMINACTIVE automatic generation algorithm.
- New zip package for download with iup images in LED format.
- New application IupView to load and display LED files.
- Fixed some attribute storage in iupMask and IupGetParam. Fixed bug when several masks are used in the same dialog.
- Replaced the internal Lua4 code for a smaller hash table module. Thanks to Danny Reinhold.
- Fixed IupGetParam invalid memory access.
- IupNextField and IupPreviousField now only changes the focus for the checked toggle inside a radio.
- IupGetAttributes now returns the pointer address if attribute is a known internal pointer data.
- Now pressing Enter over a button activates it, even if it is not the DEFAULTENTER button.
- Esc and Backspace keys now will be translated even if CapsLock is active.

Windows

- New ENTERWINDOW_CB and LEAVEWINDOW_CB for buttons.
- Fixed double click for button, toggle and list were not being considered as two clicks.
- removed FLAT style from toggles with IMPRESS image. Fixed size of toggle with image.

- New attribute SHOWDROPDOWN to open the dropdown list programmatically.
- Removed a black border around IupMultiline and IupText.
- Removed the TABSTOP for non marked Toggles inside a Radio.
- Fixed invalid memory access when menu item is activated and all dialog controls are disabled.
- Fixed IupFileDialog ignored the x,y parameters of IupPopup.

Motif

- Enter in IupMultiline activated the DEFAULTENTER button instead of adding a new line.
- Fixed invalid memory access when set FONT to NULL.
- Fixed ACTION callback called for IupList when list contents were cleared.

IupControls

- IupTree and IupTabs did not propagate to the parent the K_ANY callback for non used keys.

IupMatrix

- The TITLES, BGCOLORs, FGCOLORs and FONTs attributes were incorrectly set after a DELLIN, ADDLIN, DELCOL or ADDCOL.
- In Windows when the user double click a dropdown list now will start opened.
- The user callback scroll_cb was incorrectly registered.
- New "HIDEFOCUS" attribute to hide the focus mark when drawing.
- Now in MARK_MODE=CELL and MULTIPLE=YES you can click on the title area to mark a full line or collumn at once.
- New BGCOLOR_CB and FGCOLOR_CB callbacks.
- Fixed when MARKMODE=LIN/COL/LINCOL if the first cell in the line/column is selected the click in the title area was ignored.

IupLua

- Removed "print" debug calls in internal code.
 - IupGetAttribute/iup.GetAttribute now returns an user data if attribute is a known internal pointer data.
 - New IupGetAttributeData/iup.GetAttributeData that returns the data always as an used data.
 - Fixed incomplete initialization of image object returned by IupLoadImage.
-

Version 2.2.1 (25/Aug/2004)

General

- Fixed some minor bugs introduced in version 2.2.
- Fixed HTML help navigation.
- For disabled buttons and toggles when the IMINACTIVE is not defined by IMAGE is defined, we replace the non transparent colors by a darker version of the background color creating the disabled effect.
- New key K_PAUSE.

Windows

- Fixed dynamic cursor creation.
- Toggle with inactive image could be enabled/disabled only once.
- Fixed toggle in Radio behavior.
- Some keys were not being treated correctly.
- Improved key codes management.

Motif

- Fixed IupList setattribute VALUE and list items activated the ACTION callback.

Controls

- Circular IupDial now uses absolute angle.
- CARET did not work when set inside EDITION_CB in IupMatrix.
- Check for double initialization of IupControls.
- Better resize management for IupVal and IupDial.
- IupControls now depends on the CD library version 4.3.3 in Motif.

IupLua

- Wrong implementation of DROPCHECK_CB.

Version 2.2 (11/Aug/2004)

INCOMPATIBILITIES

- Definition of K_parenleft changed to K_parentleft in C and all Lua bindings.

- Major IupLua5 change (see IupLua section below).
- IupLua4 is not supported.
- Motif 1.x is not supported.

General

- Documentation in Portuguese removed from the manual.
- Changed and documented the default palette used in IupImage.
- IupImage can now have up to 256 colors.
- New mouse wheel callback "WHEEL_CB" for Windows and Motif. If not defined the wheel will automatically scroll the canvas vertically.
- Changes on global attributes:
 "COMPUTERNAME", "USERNAME" - now implemented also in Motif.
 "COPYRIGHT" - not documented
 "SCREENDEPTH", "SYSTEMVERSION" - new for Windows and Motif
 "SYSTEM" - Implementation were different from the documentation
 "CURSORPOS" was documented as if it was only for Windows.
 "LOCKLOOP" now implemented also in Motif..
- The definitions IUP_SBDRAV and IUP_SBDRAH were not documented.
- Callback MENUSELECT_CB changed to HIGHLIGHT_CB. Now implemented also in Motif.
- New menu callback MENCLOSE_CB.
- New utility functions IupMessagef and IupGetInt2.
- Improved visual appearance of IupScanf, IupAlarm and IupListDialog.
- New creation attribute "SEPARATOR" for IupLabel so you can create vertical or horizontal line separators.
- New IupGetText predefined dialog.
- Now all the predefined dialogs consult the global attribute IUP_PARENTDIALOG.
- New "HELP_CB" callback for all interactive controls.
- The "KEYPRESS_CB" callback now will be called repeatedly if the key is pressed and held.
- IupList can now have an edit box associated.
- The OLD newfocus parameter of the KILLFOCUS_CB is now NULL always, in Windows and Motif.
- The BGCOLOR color for IupImage transparency was not according to the documentation. It was using the default background color of the dialog. Now it uses the BGCOLOR of the control where it is inserted.

Windows

- Menus for notification icons (system tray) were not working correctly.
- Cursors in Windows now accept more than 2 colors and can have size different from 32x32.
- IupImage was rewritten in Windows to be more simple and flexible. This also solved some weird button backgrounds in gcc3.
- New global attributes "SHIFTKEY" and "CONTROLKEY" can be "ON" or "OFF", return the

the key state (windows only).

- The default size for buttons in Windows was increased by 2 characters.
- Returning IUP_CLOSE in a SHOW_CB of an IupPopup wasn't closing dialog.
- IupOpen instead of initializing OLE, now only initializes COM (CoInitialize).
- The border of buttons are now drawn by a system function instead of simulated.
- New attribute "PLACEMENT" to show the dialog maximized or minimized.
- In IupFileDlg when browsing for folder it will use a new interface, with a resizable dialog and other features.
Also in IupFileDlg fixed start position for IupPopup. New file selection callback and preview area. IupFileDlg was not using the IUP_PARENTDIALOG attribute. Default value for IUP_NOOVERWRITEPROMPT was wrong. ALLOW_NEW was inconsistent with the documentation.
- The button callback now is called only when the button is released inside the button area.
- WOM callback renamed to WOM_CB.
- New "HELPPBUTTON" attribute for the dialog.
- The menu item now accepts auxiliary bitmaps.
- When the dialog has a multiline and the user press ESC the window was improperly closed.
- Fixed combobox resize feedback. When resizing the dialog the combobox was temporarily opened.
- IupCanvas was not receiving arrow keys events correctly in keypress_cb.
- IupHide now can close popup dialogs.
- Attribute TABSIZE for IupMultiline in Windows was not documented.
- Default value for attribute BGCOLOR for IupCanvas in Windows was not documented.
- Direction keys now are processed by the ACTION callback for IupText.
- The GETFOCUS_CB and KILLFOCUS_CB management for the controls was reviewed and optimized.
GETFOCUS_CB now works for toggle and button.
- First RESIZE_CB of the canvas received a wrong canvas size.
- Label alignment for images was always center.

Motif

- New global attribute: "MOTIFVERSION".

- IUP_SBDRAV and IUP_SBDRAH were not implemented.
- HIGHLIGHT_CB menu item callback.
- "COMPUTERNAME", "USERNAME" and "LOCKLOOP" global attributes.
- IupMessage now uses native XmMessageBox.
- The overwrite confirmation dialog was closing the file open if the user answered "No".
- Implemented the IUP_NOOVERWRITEPROMPT attribute for IupFileDialog.
- The dropdown list now uses the Motif 2 combobox widget. So IUP is not compatible with Motif 1.x anymore.
- Now the GETFOCUS callback is also invoked when the list is dropdown.
- KEYPRESS_CB is now called only for IupCanvas.

Controls

- DEFAULTESC and DEFAULTENTER were missing in IupGetColor.
- New function IupLoadImage that uses the library IM to load an image file (implemented in an additional library).
- New dialog IupGetParam, similar to IupScanf but uses variable controls for fields.
- IupTabs now uses the FG_COLOR for the text color.
- ICTL_DASHED was missing in the documentation of IupGauge.
The control now has the attributes MIN and MAX just like the valuator.
- For IupVal and IupDial, new keyboard and mouse wheel support.
New attribute "SHOWTICKS" to show tick marks around the valuator.
New attribute "UNIT" to change the angle unit to degrees in the dial.
Completely changed visual of the controls.
The controls can now be deactivated and it displays focus feedback.
- Updated visual for the IupGauge and IupTabs controls.
- In IupTabs the popup menu to select a tab sometimes did not set the new tab.

Matrix

- Documentation reviewed and reorganized.
- Returning IUP_CLOSE in CLICK_CB was not closing application.
- The scrollbar drag will now simultaneously scroll the matrix.
- New callback "DROPCHECK_CB" to aid the dropdown feedback in the cell.
- New utility functions: IupMatSetAttribute, IupMatStoreAttribute IupMatGetFloat, IupMatSetfAttribute, IupMatGetAttribute, IupMatGetInt.

- Fixed some display errors in Windows because of an error in the size of the scrollbar.
- In Windows pressing a key in a menu activates the `k_any` of the last active element. In the matrix this turns into an infinite loop. The matrix now uses the `keypress_cb` instead of the `k_any` callback.
- Fixed empty selection in the dropdown list if the user presses a regular key to start editing the cell.
- Fixed invalid dropdown value if the user changed focus to the scrollbars.
- `CLICK_CB` was called twice in a double click (press+release).
- In Motif, the textbox and the dropdown did not open when you double click a cell. But now the user still needs to click again in the control to put it into focus.
- After editing the cell in the last line, now the focus goes to the column on the right at the last line, instead of the first line.
- `BG_COLOR` now works also for titles.
- `FONT` attribute now can be set/get just like `BG_COLOR` and `FG_COLOR`. But the cell size is calculated always from the matrix attribute `IUP_FONT`.

Tree

- Documentation reviewed and reorganized.
- `CTRL` and `SHIFT` accept only values `IUP_YES` and `IUP_NO`. Default value of `SHIFT` and `CONTROL` is `NO`, it was `NULL`.
- Pressing Space without Control now activates the `RENAMENODE_CB` callback.

IupLua

- The selection callback wasn't working in Lua 5 binding.
- `MOUSEMOVE_CB` in Dial control was receiving wrong angle parameter in Lua 5 binding.
- `IupGLCanvas` wasn't working in Lua 5 binding.
- Major IupLua5 change.
It now complies to LTN7 (namespaces). All exported functions are accessed only through **iup.FunctionName** (no Iup prefix anymore)
All callbacks in Lua are now accessed through their exact name in the C API. Mostly add suffix `"_cb"` to name (most common callbacks renamed for ex: `getfocus_cb`, `killfocus_cb`). Also some names were fixed: `valuecb` >> `value_cb` and `mapcb` >> `map_cb`.
Numeric definitions also changed: `IUP_DEFAULT` >> `iup.DEFAULT`
String definitions for values are no longer supported, use `"YES"`, `"NO"`, etc.
`iupcb` changed to `iup.colorbar`.
- Use `LoadLibrary` to load IUP from Lua.
- There was no stack pop in color processing loop for `IupImage` in IupLua5.
- IupLua4 is not supported anymore.

LEDC

- Added support for `IupTree` and `IupSbox`.
- Fixed include for `IupColorBrowser`.
- Fixed small invalid memory access.

Version 2.1 (18/Feb/2004)

General

- New split-panel control: IupSbox
- IupTree and IupMatrix libraries are now part of iupcontrols
- New functions to traverse IUP controls: IupGetNextChild, IupGetBrother, IupGetParent
- IupAppend accepts elements other than predefined internal controls (allowing CPI containers)
- Focus now may go to CPI controls
- Attribute IUP_X, IUP_Y are now valid for every control that has a native representation (returns the position of the control in screen coordinates)
- CURSORPOS global attribute is now returned from the driver
- IupGetFile was not allowing new files and should not change user directories
- IupGetFile was not accepting long directories
- IupAlarm does not take [ENTER] as button1 click anymore
- IupScanf does not accept "," when option is float
- Windows 95 is no longer supported

IupTree

- Trying to get attribute NAME for an invalid ID returns NULL
- Fixed attributes IUP_CTRL and IUP_SHIFT for mouse interaction

IupMatrix

- Special keys such as backspace, control+c, etc. are now ignored when not in edit mode
- leaveitem/enteritem were not being generated when the focus was leaving or entering the matrix
- leaveitem/enteritem should not be called when the cell enters edition mode through the mouse

Windows

- IupOpen/IupClose now initializes OLE (OleInitialize/OleUninitialize)
- ENTERWINDOW/LEAVEWINDOW reimplementation. LEAVEWINDOW does not fail anymore
- Mouse hook removed. Better performance
- New attributes TRAY, TRAYTIP and TRAYIMAGE and new callback TRAYCLICK_CB which allows a dialog to be put in the tray
- Action in IupText now responds to the [ENTER] key
Some keys were not working with keypress callback: \] [' ; / . ,
- New attribute NATIVEPARENT, which makes any dialog in Windows able to be parent of a IUP dialog (even from other toolkits)
- Better protection dealing with other processes messages

- IupFileDialog when used to get directory was not updating STATUS attribute correctly
- IUP_APPEND small memory problem fix
- atexit removed
- KILLFOCUS_CB and GETFOCUS_CB were not being called when focus goes to the menu
- MAP_CB in a canvas is now called before RESIZE_CB (like the Motif driver)
- ALT-F4 was not working to close application
- Images sometimes show black using Visual C: do not use option in Visual C 6.0 /NODEFAULTLIB:libcd
- IUP_TIP does not show when the fade effect is on: MS fixed the problem, use autoupdate

IupLua 3.2, 4.0, 5.0

- Functions exported to Lua: IupGetType, IupGetParent, IupGetNextChild, IupGetBrother
- IupTimer, IupSbox binding
- IupTreeGetTable, IupTreeSetTableId, IupTreeGetTableId functions created
- Several bug fixes in IupLua 5.0
- New function iuplua_pushihandle, iuplua_dofile and iuplua_dostring, IupGetFromC
- If iuplua_dofile and iuplua_dostring are used errors are reported through _ERRORMESSAGE function
- Default _ERRORMESSAGE function shows a dialog with the error
- IupLua5: Removed Lua redefinitions of dofile and dostring
- Minor bug in IupTree function TreeSetValue
- IupListDialog was not returning a table as it should when in multiple mode

IupVal

- Attribute IUP_VALUE wasn't taking effect when set before mapping
- CD canvas was being altered during mouse movement event

Manual

- CPI manual revision
- IupLua manual revision
- Several examples revised
- Controls section rearranged

Distribution

- README on how to compile IUP with tecmake

Version 2.0.1 (31/Jul/2003)

General

- Attribute IUP_TYPENAME replaced by IupGetType function

- minor bugs introduced in 2.0 because of internal old misuse of the hash table.
- Following controls were not working with LED: val, dial, gl, matrix, tree.
- New canvas attribute "DRAW_SIZE" that returns the drawing area of the canvas (in Windows we may have an additional border included in "RASTER_SIZE").

Windows

- Memory invasion when eliminating an item from an IupList with multiple items.
- Callback IUP_OPEN_CB sometimes was not being called.
- New dialog attribute "BRINGFRONT" which forces dialog to be the window in the front. Useful for multithreaded applications.
- Attribute ACTIVE was not working with radio control.
- Now folder selection in IupFileDialog uses IUP_DIRECTORY as a start path.
- Now when ESC or ENTER is pressed KEYPRESS_CB is generated

Motif

- Dropdown were becoming unstable when VALUE attribute is set after IupMap.
- Dropdown were not being positioned accordingly.
- IupList was not selecting the first item.
- IupTimer callback were called only once.
- The value "BG_COLOR" in a value of an image color table index appeared with erroneous color.
- keyboard and mouse callbacks were not being called when in full screen.

LEDC

- Updated to reflect 2.0 changes like "iupmatrx" to "iupmatrix".
- Now tests if name is not NULL before using IupSetHandle.

IupLua

- New binding for Lua 5. This is beta version since uses old notation "iuplabel" instead of "iup.label".

Version 2.0 (23/Jun/2003)

General

- IUP has undergone a large internal reorganization, but no structural or algorithmic changes have occurred. The purpose of this reorganization was to standardize function, variable and module nomenclature. This process is not yet complete, but the few remaining details will be solved in the next version.
- Table Hash was completely replaced with a modified version of Lua 4. This version is internal of IUP and does not affect applications. This has brought us a better management of the memory used by attributes.
- The CPI was changed to allow the creation of native controls, as well as controls based on IupCanvas. The internal controls were not yet rewritten over the new CPI - this will be done progressively in the next versions.
- The Ihandle definition changed from "void" to "typedef struct Ihandle_ Ihandle;". This has direct implications on C++ applications that did not do pointer typecast. In C++, code errors might occur and, in C, there might be warnings.

- New control `IupTimer`. Allows creating timers in Windows and Motif.
- New callback `"KEYPRESS_CB"`. Allows intercepting any key and replacing all callbacks `"K_xxx"`.
- `IupHelp` was rewritten in a simpler way. In Windows, it simply uses the system's configuration to open a URL and, in UNIX, it directly runs Netscape or another executable configured by an environment variable.
- New attribute `"FULLSCREEN"`, allows creating a dialog that occupies exactly the whole screen.
- Dialog `IupGetFile` was rewritten using `IupFileDlg`.

Windows

- New attribute `"CURSORPOS"`, allows programmatically changing the cursor's position on the screen.
- New attribute `"NOOVERWRITEPROMPT"` for `IupFileDlg`. It prevents `IupFileDlg` in Save mode from asking the user if s/he really wishes to overwrite a file.
- Problem corrected in the file list in the use of attribute `"MULTIPLE_FILES"` for `IupFileDlg`. When only a folder was selected, it was not setting the `"STATUS"` attribute in a cancelled action.
- Greater driver stability - `Ihandle` is no longer dependant on the native handle (HWND).
- New global attributes `"HINSTANCE"`, `"SYSTEMLANGUAGE"`, `"COMPUTERNAME"`, `"USERNAME"`.
- Global attribute `IUP_SYSTEM` now returns a more complete string.
- Cursor now changes instantly - it only changed before returning to IUP.
- In an inactive `IupToggle`, the `IMINACTIVE` image is now correct.

Motif

- The `iupmot` library no longer exists. Tecmake has been updated, but those who use their own metafiles must remove this file from the list of libraries in the application.
- New attribute `"AUTOREPEAT"` allows turning on and off the automatic repetition mode of pressed keys.

IupLua

- [4/5] `IupListDialog` when selection type is 1 (single) was not returning any value.
- [4/5] Callbacks `mapcb` and `showcb` had their names wrong: `map_cb` and `show_cb`
- [3] Callback `action` in `IupMultiline` was not passing the parameter `"after"`.
- [4/5] In `IupTree`, callbacks `"afterselection"` and `"beforeselection"` were replaced with the callback `"selection"`.

IupControls

- We have joined seven libraries in one: `dial`, `gauge`, `cb`, `gc`, `mask`, `tabs` and `val`. But neither the initialization functions nor each control's inclusion files were changed. The source code does not need to be altered, except for the makefiles. Tecmake was given a flag `USE_IUPCONTROLS` to automatically include this library.

IupMatrix

- The name of the library was changed from `"iupmatrx"` to `"iupmatrix"`. The same for the inclusion files. Therefore, all applications that use `IupMatrix` must change the source code and the makefile to reflect these changes.

IupTree

- In one case, the active CD canvas was not being returned to the old canvas before drawing.

IupGL

- In Linux, the additional GLw library was added to the control library.
 - New attributes for query in UNIX: `CONTEXT` (`GLXContext`), `VISUAL` (`XVisualInfo*`), `COLORMAP` (`Colormap`).
-

Version 1.9.1 (17/Oct/2002)

General

- Version number now resides in `iup.h` (it is also included in the library during compilation.)

Windows Driver

- `IupLabel` with `\n` was not working.
- Line-break in attribute `IUP_TIP` is now accepted.
- Double-click in the Windows top-left corner made the program crash.
- `IUP_READONLY` was only accepted if used before `IupMap` in a `IupText` or `IupMultiline`.
- Windows Driver was limiting initial elements of a `IupList` to 999.
- New attribute `FULLSCREEN` created.
- The codes of the numeric keyboard when the CapsLock was turned on were not mapped correctly to IUP.
- New callback added `MENUSELECT_CB` (called when the mouse hovers over a menu or item.) - not fully implemented.

Motif Driver

- `IUP_MOTFONT` did not accept IUP fonts. Now it accepts both native fonts and IUP fonts.
- It is acceptable now to select an option in a popup menu with any mouse key.
- Attribute `IUP_STATUS` in a `filedlg` was not working in a silicon.

IupLua

- Better error messages.
- In the `iuptree` control, the callback `BRANCHOPEN_CB` was not passing the node parameter.
- In the `iuptree` control, new functions were implemented to associate and retrieve a Lua Table from a node or leaf.
- `IupGLCanvas` binding.

IupTree

- Expand and collapse no more alters selection of elements.
- When all nodes were deleted using `"DELNODE0"`, `"CHILDREN"` inside a `tree_selection` callback, the program crashed.
- `BRANCH_OPEN` now passes parameter node.
- `IUP_DEPTH` now works for folders and leaves. Attention: the depth works only with the

appointed element, not with its children.

- Some conditions necessary for a DEPTH change were wrong.
- Redraw optimization.
- When a tree was big, the scrollbar was not working properly.
- When the tree was totally expanded and the scrollbar was all down, collapsing folders made the thumb be wrongly calculated.
- PGDN and PGUP were stopping in any folder that was closed.
- Even when the user did not want a folder or leaf to be selected, sometimes the tree allowed it.
- When the tree's folder does not have children, an empty box is shown next to it (instead of the + and - symbol.)
- Sometimes an error occurred in selection when a double click was done in a tree.
- Callback RENAMENODE_CB now works correctly.
- When the TreeSetValue function was used to define a tree, using a folder with no leaves made the program crash.
- New attribute "COLORid" allows the text color to be changed.

IupTabs

- IUP_REPAINT was not repainting the elements in its interior.

IupMatrix

- The attributes IUP_DEFAULTESC and IUP_DEFAULTENTER of a dialog were not working in Windows (they work only when the matrix is not in edition mode.)
- The matrix did not show the selected elements when the focus passed to another interface element.
- In a dropdown, when the user left edition mode changing the focus away from the matrix, the previously entered value was lost.
- Selection with the control key now works for selecting and deselecting.
- The cell with the input focus now draws the selection status.
- The attribute IUP_MARKED now works after the matrix is mapped.
- The matrix now starts with no cell selected.
- Clicking on the first column of a marked line with MARK_MODE LIN now also deselects the line.
- When MARK_MODE is LIN, COL or LINCOL the selection is not done on the focused cell.
- When MARK_MODE is CELL and MULTIPLE is NO the whole line cannot be marked.
- When MARK_MODE is NO nothing can be selected.
- The [TAB] key in the matrix now changes focus to next element.
- When MARK_MODE was NO (default), after leaving the edition mode with [ENTER] the cell was being marked.

IupVal

- Mousemove is now standardized.
- Idle is not used anymore (better optimization and code simplicity.)
- Minimum and maximum value when different from 0 and 1 now work.
- Clicking a position in the middle of the IupVal now work correctly.

Version 1.9.0 (18 Dec 2001)

General

- The K_ANY callback now considers the state of the CAPSLOCK key. The native behavior of the combination of the keys CAPSLOCK and SHIFT was kept.
- New binding for IUP: Lua 4.0.
- New binding for IupMask.

Windows Driver

- Driver Windows now deals only with messages generated for IUP elements (this used to be a problem with CD's print dialog).
- Label fonts did not work when set before IupMap.
- Attribute IUP_FILTERUSED now can be set on before the creation of IupFileDialog.
- Tip in Windows now accepts \n.
- Tip in Windows is now modified immediately after it is set through programming.
- Tip now can be removed immediately.
- In a SubMenu, the attribute ACTIVE was not working properly.
- The OPEN_CB callback was implemented in the SubMenu.

Motif Driver

- Callback OPEN_CB in a SubMenu was providing wrong parameter.
- Attribute IUP_BORDER in a dialog was working differently from the manual when the window manager was sawfish.

IupMask Control

- IupMask was becoming unstable when the user set the attribute IUP_SELECTION in a IupText.
- There was a bug in the IupMask-IupMatrix combination.

IupMatrix Control

- Adding a new column or line is now correctly dealing with color inheritance.
- There was IUP_MARK_MODE defined but not: IUP_LIN, IUP_COL, IUP_LINCOL and IUP_CELL.
- The drop_cb callback was being called for any focus change. It is now being called just when the matrix enters edition mode.
- The matrix was not showing the selected cells when the user changed focus from the matrix.
- The matrix was not calling K_ANY from the parent if the callback had been set after matrix creation.
- IUP_RIGHTCLICK_CB is now called IUP_CLICK_CB. This callback is now called for every mouse button.
- New callback IUP_MOUSEMOVE_CB.

IupTree Control

- Attribute IUP_MARKED now also sets.
- IupTree's binding now exports functions to set and get ID.
- Redraw is now done with one attribute. This avoids unnecessary redraw when the user wants to insert a lot of data.
- IupTree now takes leafs and nodes before IupMap.
- Clicking to select a LEAF was not always working in Windows.
- BRANCHOPEN and BRANCHCLOSE callbacks were not testing the return value correctly.

- Double clicking was not working properly. When the user clicked a node, while the timer was still waiting for the second click, it was impossible to click a nother node.
- Hitting the space button with CTRL pressed now marks the element immediatly.
- SELECTION_CB callback was created. This callback is called when any type of mark is made on the Tree. The return value blocks this action.
- Removed callbacks BEFORESELECTION_CB and AFTERSELECTION_CB.
- Setting IUP_VALUE though programming does not activate callbacks anymore.
- Keyboard control, including arrow keys, PGUP, PGDOWN, HOME e END were not working properly.
- Clicking + or - was not activating the SELECTION_CB callback.
- SELECTION_CB is now in the binding. BEFORESELECTION_CB and AFTERSELECTION_CB are not.
- The IUP_MARKEDid attribute now returns IUP_YES or IUP_NO depending on the state of the node's mark. If the node does not exist, the returned value is NULL.
- IupTree was breaking when it tried to erase a marked node inside BRANCHCLOSE_CB.
- The BRANCHCLOSE_CB callback was not being called for the correct node.
- SELECTION_CB was included in the binding.
- Including a new leaf now does not alter selection.

IupGL Control

- Created attribute "ERROR" indicating error in a GL canvas.

IupCB Control

- User canvas was not being reactivated after the mouse callbacks.

IupLua

- IupGetGlobal and IupSetGlobal were not doing toupper.
- New function created to get an Ihandle created in C: IupGetFromC.
- The IUP_BUTTON_CB callback was not being called.
- Functions isshift, iscontrol, isbutton1, isbutton2, isbutton3 and isdouble are now exported.
- IupPreviousField and IupNextField were not implemented.
- The OPEN_CB callback was implemented in the binding with the name OPEN.
- New callback IUP_MOUSEMOVE_CB for matrix.

Version 1.8.9 (07 May 2001)

IupMatrx Control

- If the user defined FG_COLOR while the matrix was in edition mode, the application crashed.
- Hitting Esc was causing garbage to be written in the matrix field.
- A bug that made the value_edit callback be called several times was fixed (it was called several times because the matrix kept trying to exit the edition mode with other events).

IupTree Control

- New IupTree control.

- Scrollbar.
- Multiple selection.
- Default image size: 16x16.
- Lua Binding.

IupCB Control

- The name of the Lua `colorbrowser` element has changed. Now it is called `iupcb`, not `cb`.

Windows Driver

- The `IUP_MULTIPLEFILES` attribute was created. Now it is possible, in Windows, to select several files in a `FileDialog`.
 - `IupHelp` now only initializes DDE when it is used.
-

Version 1.8.8 (15 Mar 2001)

- The `global.h`, `macros.h`, `rgb.h` and `hls.h` files are no longer exported by IUP.
- Some keys were in conflict among themselves (shift-home and 4, for instance). Shift-space and Ctrl-space were added to the `K_ANY` callback (Windows and Motif).
- `IUP_VISIBLE` was returning `NULL` on IUP when the dialog was not mapped.
- `IupSetLanguage` can now be called before `IupOpen()`;
- `iuptoolbar` and `iupfiletext` were removed from the distribution.

CPI

- Several defines (such as `stricmp`) are no longer exported from `iupcpi.h`
- Functions `iupAddSymbol`, `iupGetSymbol`, `iupgetdata` and `iupsetdata` are no longer exported from the CPI.

Motif Driver

- The `Tip` font is now inherited from the element it belongs.
- Inserting a text (`IUP_INSERT` or `IUP_APPEND`) on Motif was ignoring the maximum number of characters.
- Some `ITALIC` fonts were not working.
- Several visibility problems were fixed for `ZBOX` inside a `ZBOX`.
- The default value of the `ALLOWNEW` attribute (in `fileopen` mode) allowed creating a new file (now standardized).

IupTabs Control

- `IupTabs` was not considering attribute `IUP_ALIGNMENT`.
- `Tabs` was not showing the selected element if it was selected while the `Tabs` was invisible (it was a Motif bug).
- The `<TAB>` key was neither passing the focus to `IupTabs` nor taking the focus off it.
- The `SIZE` attribute is now defined for the tabs of `IupTabs` – `ICTL_TABSIZE`.
- Changing the text value for `Tabs` was not recomputing the `Tabs` size.
- The appearance of `IupTabs` was enhanced.
- `IupTabs` now sends the focus back to the first element when the user tries to shift right after

the last element.

- Now a redraw can be forced on Tabs with the IUP_REDRAW attribute.

IupMatrix Control

- Ctrl+arrows was not working properly.
- The behavior of the DEL key to delete a set of cells now also considers the return of the IUP_EDITION_CB callback.
- The mark is now shown (not the focus) when matrix loses the focus (users were having problems when wishing to hit a button to cause an action over the matrix).
- On the NT platform, the fields of the created matrix had the wrong values when an automatic scroll occurred.
- Right-clicking the matrix now passes the control parameter (as in BUTTON_CB) `isshift(r)`, `iscontrol(r)`, `isbutton1(r)`, `isbutton2(r)`, `isbutton3(r)`, `isdoube(r)`
- Vertically scrolling by dragging the thumb now works properly.
- The focus is now correctly drawn inside the matrix (when only half the cell appears, half of the focus is drawn).
- When leaving the edition mode by clicking an element outside the matrix, the focus was remaining on the IupText in the matrix.
- Colors and alignments are now moved when a cell is moved either by adding new lines or columns or by deleting lines or columns.
- The matrix now leaves the edition mode whenever lines or columns are removed.
- When the user clicked a cell near the end of the matrix (on the x coordinate) an automatic scroll was made and the cell beside the desired cell was marked.

Windows Driver

- KEY in IupItem was replicating the underlined KEYs (and some times adding the wrong values because of that).

IupLua.exe

- Now works properly with all controls.

IUP Manual

- All elements now have examples at least in IupLua and C.
- The IupMask manual was created.

Version 1.8.7 (23 Nov 2000)

- The alignment of composition elements can now be changed on-the-fly.
- Current language treatment has been changed. ATTENTION: previous `putenv` no longer works! Use new functions `IupSetLanguage` and `IupGetLanguage`. Default language: Portuguese.
- IupAlarm's design was reformulated. Now all buttons have the same size.
- Functions `IupUnMapFont` and `IupMapFont` were created to make the use of the driver's fonts easier.
- Attribute IUP_FONT now accepts a string either with the native font or the IUP font, and always returns the native font (attributes WINFONT and MOTFONT are now obsolete).

Motif Driver

- Motif did not have `K_ANY` for `IupList` in dropdown mode.
- The `IUP_VISIBLE` attribute now works for `FRAME`, `ZBOX`, `VBOX`, `HBOX` and `RADIO` (all elements were tested). Now it is no longer lost for internal `HBOX` elements when the `HBOX` visibility is changed.
- When the user changed from one `ZBOX` to another, the first one was forgetting which elements were visible.

Windows Driver

- When `Toggle 1` (default) begins deactivated, it no longer remains marked forever.
- `Toggle` with image now accepts images `IUP_IMPRESS` and `IUP_IMINACTIVE`, but it follows the Windows standard for `Toggle` manipulation.
- `Toggle` was not verifying whether it was active or not when it was created.
- Canvas redraw was optimized. The canvas now uses transparent color as default. The user is in charge of drawing the canvas, but now it no longer “blinks” when a redraw is made. Tip: To avoid unnecessary canvas redraws, do not put it inside a frame and use the `IUP_CLIPCHILDREN` attribute.
- Initializing `Toggle` (or `Radio`) with a value and then modifying it via callback was marking both toggles.
- Changing `Toggle`’s color (`IUP_FGCOLOR`) was not working on Windows unless its background color was also changed.
- `IupItem` outside a submenu was not calling the callback.
- On Windows, the `IUP_HOTSPOT` attribute was being read incorrectly (the correct form is with `:'`).

IupMatrix Control

- `DROPDOWN`’s function in `Matrix` was corrected. Now the user fulfills the dropdown values, which always start at position 1. If the user wishes, he/she can set the initial dropdown value by checking the `IUP_PREVIOUSVALUE` attribute about the dropdown element passed as parameter. This attribute returns the previously selected string value.
- `Dropdown` now enters “edition mode” just as regular fields do.
- `Dropdown` can automatically close after the user’s choice. Simply return `IUP_CONTINUE` for the callback chosen by the dropdown.
- Now the dropdown accepts the `ESC` key, restoring its previous value.
- An element with focus is now drawn with double focus.
- The color of a selected element is now 20% attenuated.
- When the user entered edition mode using the mouse and exited it hitting `ENTER`, the cell remained selected.
- `Matrix` no longer gets lost when it has 0 lines.
- `Matrix` was not accepting a user to return a constant string with `‘\n’` from a callback.
- A `Matrix` that loses the focus does not lose the selection (but it is not apparent).
- `TAB` no longer changes cells in the `Matrix` (it now changes `IUP` elements).
- Hitting ‘delete’ on a marked element deletes everything.
- `Matrix` leaves the edition mode when `IupText`’s exit arrows are used.
- There was a computation mistake in cell size when the `Matrix` was in edition mode.
- When the user scrolls, the `Matrix` exits the edition mode.
- ALL problems caused by `cdActivate` in `Matrix` were solved.

Other Extended Controls

- The element from `IupGL` was not getting the focus when it was the only element in the dialog.
- In `IupGL`, `OpenGL` now synchronizes its functioning with Motif (`glXWaitX`) at resize.

- IupGC now works with IUP_ENGLISH's variable set (cancel/cancela, red/Verm, etc.)
- IupGauge now accepts changing text or percentage values on-the-fly.
- Tabs font now has a differentiated color when it is inactive.

IupLua

- IupScanf at IupLua was not performing the final dialog's popup.
- IupSetLanguage, IupGetLanguage, IupMapFont and IupUnMapFont were created at IupLua.
- It now considers the IUPLUA_QUIET attribute.
- The callbacks in IupLua are now inherited (eg.: k_any from a dialog is called when IupCanvas does not have k_any).
- The library's opening message now follows a standard.
- IupLua was passing Lua's pointer to IUP instead of copying its value in IupSetHandle (making it crash).

IupLua Program

- iuplua was not running with IupVal and IupGetColor.
- iuplua now accepts several files as a parameter.
- iuplua is now joined with iupluafull
- iuplua now shows line number and cursor column.

Version 1.8.6 (21 Jun 2000)

- All libraries were generated for AIX 4.3.2, which is available in new IBM machines.
- A series of memory management problems was solved for all platforms.
- Attribute IUP_SELECTEDTEXT now can also be used to change the selected text in a IupText and IupMultiline field.
- The IupLabel element now takes the IUP_ALIGNMENT attribute into account.
- The IupList (dropdown) element now always leaves some option selected (unless there is none to select).
- When the selected elements value in IupList (dropdown) is changed, it now remains selected with the new value.

User Manual

- The user manual is now also available in several Windows Help formats, including the help format for Visual C++ (5 and 6). To configure your account for Visual C++ to access IUP's Help, run W:\iup\help\iuphelp.reg (ATTENTION: On Visual Studio, IUP's manual must be activated and deactivated through option "Help -> Use extension Help"). Other available formats can be found at W:\iup\Help.
- A general revision of the user manual is being made.
- The CPI manual was rewritten.
- Several examples were included.
- An application called "iupluatest" (W:\iup\bin) was created to run the IupLua examples included in the manual (it works with the controls using the installed DLLs).

Windows Driver

- There is no longer any restriction for the number of dialogs created using IUP (the only limitation now is Windows' capacity to create native elements).

- Events of `IupButton` and `IupToggle` were being improperly called when a `IupHide` or a `IupShow` was made on the dialog.
- A bug when drawing an image associated to a `IupToggle` element was fixed.
- The functioning of attributes `IUP_DEFAULTENTER` and `IUP_DEFAULTESC` was corrected.
- Now, when a user changes the selection of a multiple `IupList` via programming, IUP internally updates the selection.
- The `IUP_BGCOLOR` attribute to define a new cursor was not standardized with the Motif driver, and color 0 in the Windows image was never allowed to be transparent.
- A bug in the dropdown list was fixed. It was not calling callback `GETFOCUS_CB`, causing instability in the `IupMatrix` element).
- The transparency color in a cursor now must be color number 0 (according to the manual, this is the way it was supposed to be).
- The `IupList` (dropdown) callback is no longer called for element 0 (which does not exist).
- A button in a `Popup` dialog was only allowing to be pressed via mouse. Now it can be pressed with the space key.
- The “`IupSetAttribute(x, IUP_VISIBLE, IUP_YES)`” call, when `x` was a dialog, was not working.
- Calling `IupHide` with a frame, with `[hvz]box` or with `radio` was not the same thing as calling “`IupSetAttribute(n, IUP_VISIBLE, IUP_NO)`”.
- The `IUP_MOUSEPOS` position in a dialog’s `IupPopup` was not functioning.

Motif Driver

- Several memory leaks were fixed. They occurred when `IupGetAttribute` called functions from XM which allocated memory to store the attribute’s value. This change may cause problems for applications which did not copy the value returned from `IupGetAttribute` and used the returned string. This usage of the return value from `IupGetAttribute` is not appropriate, because the user has to copy this string if he/she intends to remain using it (the returned string is intern to IUP).
- The dialog’s `Close` callback was not closing the application when it returned `IUP_CLOSE`.
- The `IUP_ACTION` callback from `IupMultiline` was not returning the new text value if the key was validated (parameter *after*).
- The dropdown list was not automatically showing the first element when it was opened.
- The Motif driver now returns the default font when “`IupGetAttribute(n, IUP_FONT)`” is performed.

IupLua

- The names of callbacks `show_cb` and `map_cb` were corrected.
- A bug that made a toggle image not appear was fixed.

Extended Controls

- The default cursor of the `IupMatrix` element now looks like the MS Excel cursor. (Remember to call `IupMatrixOpen()` even when using `IupLua`!)
- Alignment (center) of the field in column 0 of the `IupMatrix` element.
- The user can now return `IUP_CONTINUE` at the action callback of element `IupMatrix` to allow IUP to go on treating pressed keys in the conventional IUP way.
- The dropdown list at `IupMatrix` was losing its current value when the user changed cells.
- The `IupGetColor` element was being drawn outside the canvas (old problem in `cdActivate`).
- The font in `IupTabs` is now inherited.
- Attributes `ICTL_ACTIVE_FONT`, `ICTL_INACTIVE_FONT`, `ICTL_FONT` were

implemented in the IupTabs element.

- Attribute IUP_MARGIN was implemented for the IupGauge element.

Version 1.8.5 (18 Apr 2000)

- The versions of libraries IUP and IupLua were synchronized. From this version on, these tools will be distributed together.
- The library generation mechanism was changed to use libmake. All DLLs are available and following the same standard as the DLLs of other Tecgraf libraries.
- A FAQ was created for IUP: <http://www.tecgraf.puc-rio.br/~mark/iup/faq-iup.txt>.
- Several memory management problems were fixed.
- Attribute IUP_DIALOGTYPE can now assume three values: IUP_OPEN, IUP_SAVE and IUP_DIR. Due to the creation of IUP_DIR, the IUP_ALLOWDIR attribute is no longer used.
- One more value was added to attribute BGCOLOR: IUP_TRANSPARENT (used only by the Canvas to avoid unnecessary drawing).
- Function IupGetError was removed from iup.h.
- Function IupDataEntry was removed from iup.h.

Windows Driver

- Function iupdrvSetIdleFunction was added to make the Windows driver compatible with Motif.
- The bug that made IUP crash when using MessageBox inside a button callback was fixed.
- IupDestroy now reconfigures the button control function (it was making IUP crash).
- The IUP_READONLY attribute was implemented (valid for Text and Multiline).
- The IUP_FILTERUSED attribute was implemented: it informs which is the filter selected by the user (1, 2, 3...).
- A bug that caused IupPopup(IupMenu(item)) not to call the item's callback was fixed.

Motif Driver

- IupDestroy was corrected. In a IupFrame, it made IUP crash.
- IupList was corrected. It crashed when the user changed its elements and tried to set IUP_VALUE.
- The memory leak at IupGetFile was removed.
- List elements were not being correctly deleted.

IupMatrix Element

- The bug in the NT matrix was fixed. It was not refreshing added elements (the values on the cells were wrong).
- The bug in the scroll matrix was fixed.

Version 1.8.4 (09 Dec 1999)

Windows Driver

- A problem, which called the dropdown callback even for an already-deleted element, was fixed.
- Function IupHelp is now available.

- A bug was fixed; it caused excessive system resource usage when dialogs with several elements were used.
- The size of the version dialog was corrected.
- A bug was fixed; it made IUP crash depending on the use of `MessageBox`. Same for `IupFileDialog`.
- Callback `IUP_BUTTON_CB` was added for the `IupButton` element.
- A bug was fixed; it made `IupGetInt(d, IUP_X)` return a wrong value when the dialog was maximized.

CPI Controls

- The color inheritance problem was fixed.
 - Corrections were made to the `Dial` size.
 - Attributes of colors `FGCOLOR`, `BGCOLOR`, and fonts `FONT`, `WINFONT`, `MOTIFFONT`.
-

Version 1.8.3 (15 Jun 1999)

Windows Driver

- The `IUP_ACTIVE` attribute now also works in the frame.
- The `action` callback in `Multiline` now also accepts the DEL key.
- `Toggle` element now accepts an image.
- The `IUP_TOOLBOX` attribute was implemented for dialogs.
- A bug was removed; it made a second `IupShow` in a dialog reset its position to the center of the screen.
- Treatment of the `SIZE` and `RASTERSIZE` attributes was changed.
- The `IUP_ACTION` callback now treats the DEL key and commands and keys from the Cut and Paste menu.
- A conflict was solved; it made the key – generate a call to the callback as if it were key ` (plic).
- Keyboard accelerators for menus now work, since the focus is no longer on the dialog. When a dialog receives the focus back, it sets the focus to the last control inside it that had the focus.
- `IUP_K_ANY` no longer issues beeps when keys are pressed on the canvas.
- When the `IUP_STARTFOCUS` attribute is not defined, the focus is set for the first control in the dialog that accepts it, thus preventing the dialog from keeping the focus and allowing the menus to be called via accelerator.
- Attribute `IUP_SELECTION` was implemented.

Motif Driver

- Color management for 8bpp displays (256 colors) was re-implemented. Basic colors used by IUP (black, white and the grays used for highlight and shadow) are now reserved, and the search for colors in the palette was optimized.
- Elements such as `IupCanvas` now have their own visual, independent from their “parent’s”. If allowed by the display, the default visual of a canvas will be `TrueColor` (24bpp); if not, it will be the same as the default display visual.
- The `IupToggle` element now processes the `IMAGE` attribute differently: it now shows the toggle with the same appearance as the `IupButton` element, but maintaining its functionality – the button remains pressed until the user clicks it again. The `IMPRESS` attribute can be used to define the image used for the pressed button. In this case, the user is in charge of giving it a 3D appearance.
- **IMPORTANT:** The size of the dialog can be adjusted after being mapped, by means of the

SIZE and RASTERSIZE attributes

- The size of the dialog has now precedence over the smallest size required by its children (either having been specified in its creation or in run-time).
- Attributing a NULL value to the SIZE or RASTERSIZE (in C) of a dialog will re-compute its size according to the size of its children.
- Partial dimensions (###x and x###) are now treated correctly.
- Therefore, applications that define sizes for dialogs (either in LED or in C) smaller than the minimum size required by their children will show truncated dialogs. To force a computation based on the size of the children, set any of these attributes to NULL (in C) or simply do not define them in LED. As a general rule, avoid specifying a dialog size unless there is a real need for such – in this case, be careful to specify a sufficient size.
- IupFileDlg:
 - The default value for the DIALOGTYPE attribute was not being recognized (the program aborted when there was no defined value).
 - When ALLOWNEW = NO, the dialog informs if the user is specifying a non-existing file (instead of simply returning, as was happening).
 - When the dialog type was OPEN, the returned value was -1 (Cancel) even when the user confirmed the operation.
 - If DIALOGTYPE is SAVE, a confirmation is required if the file already exists.
 - A new dialog was created for each popup without destroying the previous dialog.
 - The NOCHANGEDIR attribute was implemented.
 - The dialog does not return if the user specifies a new file when attribute ALLOWNEW = NO. The same happens when attribute ALLOWDIR = NO and a directory is specified. In these cases, alerts are shown.
- The iupGetColor function for CPI controls was replaced in functionality by the iupGetRGB function (iupGetColor is maintained for compatibility purposes, but it should no longer be used).
- TRUECOLORCANVAS was created. It indicates if the display allows the creation of TrueColor windows (> 8bpp), even if the default is PseudoColor.
- Tabs: a problem was fixed concerning the use of the VISIBLE attribute for elements belonging to a non-selected tab.
- IupHelp: allows using a browser (default = Netscape) for viewing HTML pages.
- The ACTION_CB callback, from IupText, now receives, apart from Ihandle* and int, a char* pointing to the new text value in case the key is confirmed.
- Dropdown lists were not correctly processing the VISIBLE attribute.
- A problem with the initialization of multiple-selection lists was solved: the VALUE attribute was not being respected in some cases.
- Attributes FGColor and BGColor from the dropdown list were not being correctly updated.
- IupLoopStep was re-implemented: now it no longer blocks when there are no events to be processed (it simply returns DEFAULT).
- The dropdown list is closed when the associated textbox is totally or partially darkened.
- The dropdown list was not being closed when the dialog lost the focus if IupIdle was registered.
- A problem in the exhibition of CPI controls was fixed.
- New return code (CONTINUE) was created, specific for key callbacks, to be used when the event is to be propagated to the parent of the element receiving it.
- In some situations, elements destroyed by means of IupDestroy were receiving events, making the application abort.
- The redefinition of items in the main menu was making the dialog return to its original size.
- Consulting attribute BGColor in a dropdown list was aborting the application.
- Consulting attributes BGColor and FGColor of a canvas with a different visual from the default was generating an X-Windows error message.
- The problem with IupFileDlg was fixed (the application was aborting).

- `IupDestroy` in a bar menu was inducing an infinite loop to the application.
 - The list now matches the documentation: it calls the action callback for the de-selected element (with the `v = 0` parameter).
 - Bug correction: The use of a Motif attribute instead of a function was making Motif lost control of memory management (memory already liberated was liberated again, which aborted the application).
 - `ACTION` in `IupText` caused `SIGSEV` when the user pressed `ENTER`.
 - New `IupMapFont` for mapping IUP fonts -> Motif.
-

Version 1.8.2

Windows Driver (12 Jan 99)

- Function `char* IupMapFont(char* font)` converts a IUP font describer (used by the `IUP_FONT` attribute) into a native font describer (used by `IUP_WIN_FONT`).
- File Drag & Drop was implemented in dialogs and canvases, via the `IUP_DROPFILES_CB` callback.
- Attribute `IUP_EXTFILTER` was implemented for the `IupFileDlg` control, allowing the use of more than one filter.
- Changes were made to allow the creation of CPI elements other than `CANVASES` or `dialogs`.
- The `IUP_ACTIVE` attribute of a dialog can now be changed after it was mapped.
- List callback correction: the callback is now called both for selected and not selected items.
- New function `void IupHelp(char *url)` shows a URL in a Netscape window.
- The treatment of the new return value for keyboard callbacks, `IUP_CONTINUE`, was implemented.
- `IUP_CURSOR` attribute was implemented.
- A code was added to treat the case of toggle de-selection via `IupSetAttribute`.
- `IUP_CARET` now uses `' , '` as a separator instead of old `' : '`.
- A restriction was eliminated that prevented the function `iupGetTextSize` from being called passing a dialog or frame as a parameter.
- New text callback was implemented; it receives the text both before and after the change, and receives the code of the typed key.
- It was possible to set two activated radio toggles by selecting `VALUE` for one of them on the radio and `VALUE = ON` on the other toggle.
- Attributes `IUP_STARTFOCUS`, `IUP_DEFAULTENTER` and `IUP_DEFAULTESC` were implemented.
- The `IUP_VALUE` of a `IupRadio` was not allowing to be changed if it was not visible.
- A problem was corrected for the lists, which were being reset between a `IupShow/IupPopup` and another.
- Attribute `IUP_SELECTEDTEXT` was implemented. It returns the selected text (if there is any), with the `'\r'` already filtered.
- A bug was corrected; it caused an "Assertion Failed" when the mouse was moved after a window was destroyed.
- The value of `IUP_VALUE` of a `IupText` and a `IupMultiline` now does not contain `'\r'`.

Motif Driver v1.8.2 (14 Aug 98)

- `IupFileDlg` was corrected: the `IUP_FILE` and `IUP_DIR` attributes were not being treated correctly.
- In some specific situations, closing a dialog could lead to the end of `IupMainLoop`, causing an abortion of the application.

Version 1.8.1

Windows Driver v1.8.1 (17 Jul 98)

- Correction: IUP's Matrix element was being shown with different fonts from the ones used by IUP, especially on UNIX platforms.
- A bug related to ZBOX was fixed.
- `IupAppend` on `Multiline` now includes '\n' at the end of the text.
- A font set by `CD` no longer affects canvas size computation.
- `IupSetAttribute` from a `IupRadio`'s `VALUE` with the name of a toggle with more than one name now works.
- Default attributes now store values that match the documentation.
- Function `IupFlush` was implemented.
- Small errors in dialog size computations were corrected.
- Now the dialog size is changed when the size of one of its children increases.

Motif Driver v1.8.1 (16 Jun 98)

- Correction: IUP's Matrix element was being shown with different fonts from the ones used by IUP, especially on UNIX platforms.
- Dropdown list (combo box) remained opened if the element was hidden or destroyed.
- The use of popup dialogs was sometimes preventing the last `IUP_CLOSE` (or `IUP_DEFAULT`) from ending `IupMainLoop`.
- [LINUX] The `button_press` event was not being received by the canvas when the `CTRL` key was pressed.

Version 1.8 (29 May 98)

- General (also includes changes to both drivers)

- BUG: `Valuator`, `Dial` and `Gauge` could cause an invalid memory access on `resize` or `destroy`.
- BUG: The parse of CPI elements described in `LED` was corrected.
- BUG: `Valuator` was removing the application's idle action.
- NEW: `FILEDLG` control.
- NEW: `IupStoreAttribute` function.
- NEW: `IupSetfAttribute` function.
- NEW: `IupSetGlobal`, `IupGetGlobal` and `IupStoreGlobal` functions for global attributes.
- NEW: `K_sCR` key; shift-enter combination is now treated by IUP (callback: `IUP_K_sCR`, code: `K_sCR`).
- NEW: `IUP_TYPENAME` attribute returns the name of the element type.
- NEW: CPI popup method.
- NEW: Definition of global attributes (verification only) `IUP_VERSION`, `IUP_DRIVER`, `IUP_SYSTEM` and `IUP_SCREENSIZE`.
- NEW: Attributes `IUP_X` and `IUP_Y` were implemented, for dialogs only. They provide the dialog's upper left corner coordinates in relation to the upper left corner of the screen.
- NEW: `IUP_SHRINK` attribute to change the computation of the position and size of elements.
- NEW: CPI control for an OpenGL canvas.

- CHANGE: The IUP_TYPE attribute of the IupFileDlg control was changed into IUP_DIALOGTYPE, which must contain OPEN, SAVE or NULL.
- CHANGE: The IupSetAttributes function now returns the Ihandle*.
- CHANGE: The IupSetAttribute function no longer returns the old value.
- CHANGE: CPI's create method now creates the handle.
- CHANGE: New function for CPI class creation.
- CHANGE: Some obsolete definitions of iup.h are now only available when the IUP_COMPAT macro is set.
- CHANGE: The ICTL_TYPE attribute of the IupTabs control was changed to ICTL_TABTYPE.

- Lua Binding

- NEW: iupkey_open function allows using IUP's key definitions in Lua.

- Windows Driver

- NEW: Image now accepts "BGCOLOR" color. This turns the color associated to the index into the background color of the element linked to the image.
- BUG: the IUP_TITLE attribute of the IupItem element can now be changed after the element has been mapped.
- BUG: A color problem was fixed; it occurred when the name or path of the executable file contained spaces.

- Motif Driver

- BUG: The dropdown list no longer remains on the screen.
- BUG: The computation of scrollbar attributes POSX and POSY was fixed.
- BUG: Double-click was only being generated for the first button.
- BUG: FRAME layout was corrected.
- BUG: The color of the menu item was corrected.
- BUG: The management of the nested elements of a ZBOX and/or with the VISIBLE attribute defined for its children was fixed.
- BUG: The color remained undefined when the value of attribute FG_COLOR or BG_COLOR was not valid.
- BUG: General cleaning was made to remove memory leaks from the driver.
- NEW: Attributes IUP_X and IUP_Y to provide the pixel position of any element.
- NEW: Attribute IUP_RASTERSIZE can be consulted.
- NEW: Menu item now accepts '\t' to align the text to the right – Windows already allowed it.
- NEW: Version number was added; can be retrieved with tecver.
- CHANGE: Multiline's scrollbar is no longer deactivated with ACTIVE=NO.
- CHANGE: Multiline's and list's BG_COLOR no longer affects the scrollbars.

Version 1.7

- The implemented code was made compatible with manual specifications. iup.h was changed to reflect that. To use old definitions, set IUP_COMPAT before including the iup.h file to the applications.

To Do

General

- Allow the functions `IupAppend` and `IupDetach` to be used for dynamic creation of menus and `IupTabs`.
- To allow `IupShow` after a `IupPopup`.
- To show a border for visual location of `VBOX`, `HBOX` and `FILL`. Can be a dialog attribute.
- Transform all controls into CPI controls, improve the possibility of implementing new drivers.
- Flat button activated by mouse hover.
- a gtk driver in Linux?
- a wxWidgets driver?
- a MacOS X driver
- Reduce flicker when dialog is resized.
- Change all comments in the source code to english. Add comments to the internal includes for Doxygen.
- Attribute for boxes to retrieve the child controls like in `IupLua`. For now you should use `IupGetNextChild` and `IupGetBrother`.
- Fix names of headers, initialization functions and libraries that do not have the "iup" prefix.
- Improve the fullscreen policy.
- Add a Tutorial section to the manual with some "simple" examples.
- Install IUP in a GForge environment.

Motif

- Callback `SHOW_CB` is not called when the dialog is hidden because of `PARENTDIALOG`.
- Sometimes the size of `TEXT` when it is opened is much larger in old SunOS. Resizing makes the text go back to the correct size. (The problem persists even if run on Linux, simply by viewing it on SunOS, seems to be a bug in the SunOS Motif.)
- Several warnings in the SunOS when using the OpenGL canvas.
- When another Window Manager is running the `IupPopup` disable the other windows, but they can be placed in front of the popup window if `PARENTDIALOG` is not used. Also in this case, some window decorations do not work.

Windows

- Enable XP Visual Styles. When using a manifest file, the text in the controls are being mixed with unicode chars.

IupControls

- A vertical `IupGauge`?

IupMatrix

- In Motif, when start editing using a double click, the user must click again to the edit control get the focus.
- When removing a line, if it has the focus an invalid call to `enteritem_cb/leave_item_cb` will occur for the removed cell.
- Copy and paste compatible with Excel?
- Sort for columns?

IupTree

- Drag and Drop.

- Rename node like in IupMatrix.
- Images with variable sizes for nodes.

IupLua

- IupGetParam binding.

New Controls

- Spin Button
- Image Listbox
- Concrete Layout Container (to position elements with absolute coordinates relative to the box)
- Grid Container (to distribute elements in a grid)
- IupOle control

Comparing IUP with Other Interface Toolkits

Why to still maintain IUP if today we have so many other popular toolkits?

This is a question we always ask to ourselves before going on for another year.

To answer that question we must first define the characteristics of the "ideal" toolkit, list the available toolkits and compare them with the "ideal" and with IUP.

We would like a toolkit that has:

- **Portability.** That provides an abstraction for User Interface in Windows, UNIX and Macintosh.
- **Free License and Open Source.** This means that we can also produce commercial applications. The pure GPL license can not be used but the LGPL can but must contain an exception stating that derived works in binary form may be distributed on the user's own terms. This is a solution that satisfies those who wish to produce GPL'ed software and also those producing proprietary software. Many libraries are distributed with this license combination.
- **Small and Simple API.** This is rare. Many libraries assume that an Interface toolkit is also a synonymous of a system abstraction and accumulate thousands of extra functions that are not related to User Interface. At Tecgraf we like many small libraries instead of one big library. Almost all available toolkits today are in C++ only, so C applications are excluded, also this means a hundred classes to include and understand each member function. The use of attributes makes a lot of things more elegant and simpler to understand.
- **Native Look & Feel.** Many toolkits draw their own controls. This gives an uniformity among systems, but also a disparity among the available applications in the same system. Native controls are also faster because they are drawn by the system. But the problem is what's "native" in UNIX? Some commercial applications in UNIX start using Motif as the "native" option. It is the official standard but because of license restrictions, before the OpenMotif event, the system became old and some good alternatives were developed, including GTK and Qt.

Toolkits

With these characteristics in mind we select some of the available toolkits:

--	--	--	--	--	--	--	--	--

Name	License	Last Update	Version	Language	Platforms	Controls	Team	Comments
V	LGPL	1998-2003/03	1.9	C++	Win, X	native	1	
ZooLib	MIT	2000-2003/04	0.9	C++	Win, X, Mac	own	4+9	Still no 1.0 version
Fresco	LGPL	1998-2004/02	Alpha	C++	Win, X, Mac	own	9	License restrictions, Still in Alpha, Use CORBA...
YAAF	BSD*	2002-2004/03	1.1a8	C++	Win, X, Mac	own	1+9	
GraphApp	BSD*	1997-2004/03	3.52	C	Win, X, Mac	own	1	Small and interesting.
FOX	LGPL*	1997-2004/03	1.1.51	C++	Win, X	own	3+15	great look, lincese totally free?
FLTK	LGPL*	1998-2004/04	1.1.5	C++	Win, X, Mac	own	3+16	was from Digital Domain. Easy to learn.
GTK+	LGPL*	1997-2004/03	2.4	C	Win, X	own	9	target for X-Windows, basis of GNOME, Windows is apart
Qt	GPL	1994-2004/03	3.3.1	C++	Win, X, Mac	own	(many)	X is free for Non Commercial, basis of KDE, Windows is paid, Emulates the native look and feel
wxWidgets	LGPL*	1992-2004/02	2.4.2	C++	Win, X, Mac	native	6+11	
IUP	MIT*	1994-2004/08	2.2	C	Win, X	native	2	

More toolkits can be found here: [The GUI Toolkit, Framework Page](#).

An interesting article can be found here: [GUI Toolkits for The X Window System](#).

Conclusions

From the selected toolkits using the defined approach we can eliminate some toolkits:

The 4 first are not updated anymore or the development is very slow.

GraphApp is an one author only project. FOX has a great look but the license can be restrictive in some cases. FLTK promesses a new version with a better look, but until then it does not have a pretty good look. The FLTK documentation also does not help.

GTK+ can be used as a replacement for Motif, but not as a "portable" toolkit since is

target for X-Windows. Qt has several license limitations, although is a very stable and powerful toolkit. Qt can be also used as a replacement for Motif.

The "best" free solution that we choose is **wxWidgets** because of the native controls and its portability.

Developing IUP

IUP uses Native Controls in Windows and Motif. Mac port is outdated and not maintained for long time, MacOS 9 was terrible. With Mac OS X we may have the opportunity to do something better. IUP is in C, is small and powerful.

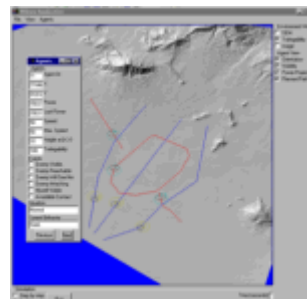
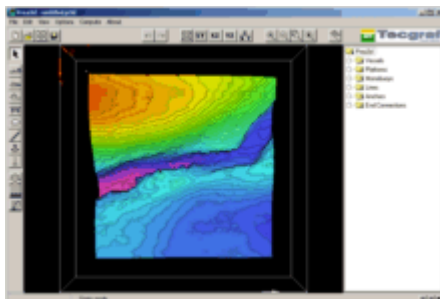
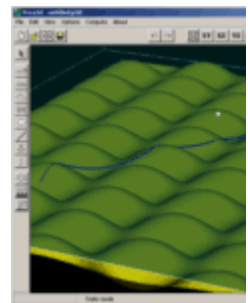
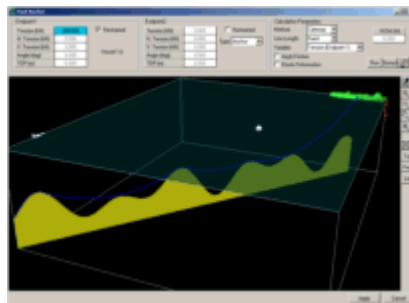
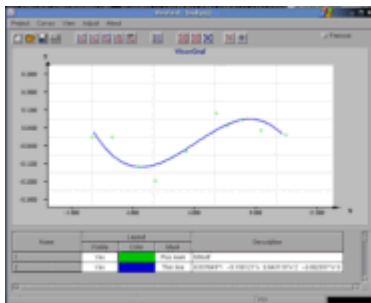
But today only 2 developers, only a few extra help and many other projects to work simultaneously. So it is hard to keep the code updated. One possibility that can reduce our demand is to implement IUP on top of one of these toolkits, so we can focus on the best of our toolkit: its API, the dynamic layout, the Lua binding and the CPI controls. But this is a polemic issue...

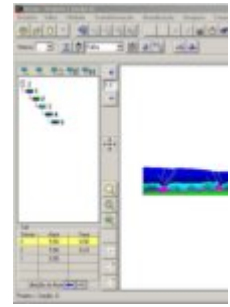
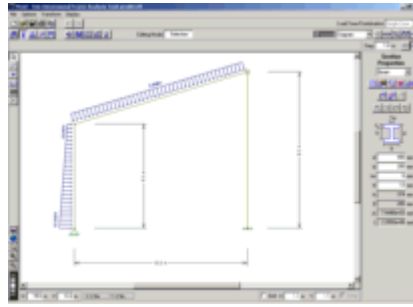
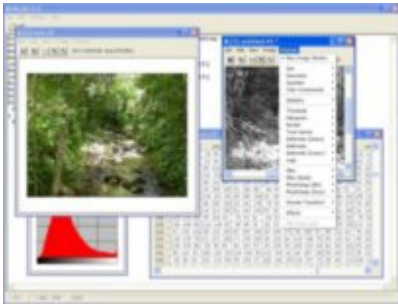
Also IUP does not have a wide localization feature, it only includes support for messages in English and Portuguese. And it does not have support for Unicode characters.

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

Screenshots

Click on the picture to enlarge image.





Guide

System Control

Before running any of IUP's functions, function **IupOpen** must be run to initialize the toolkit.

After running the last IUP function, function **IupClose** must be run so that the toolkit can free internal memory and close the interface system.

Executing these functions in this order is crucial for the correct functioning of the toolkit.

Between calls to the **IupOpen** and **IupClose** functions, the application can create dialogs and display them. However, IUP is an event-oriented interface system, so it will keep a loop "waiting" for the user to interact with the application. For this loop to occur, the user must call the **IupMainLoop** function, which is generally used right before **IupClose**. When the user closes the application, function **IupMainLoop** will return, calling **IupClose** and ending the application's execution. .

Therefore, usually an application employing IUP will have a code in the main function similar to the following:

```
void main(void)
{
    if (IupOpen() == IUP_ERROR)
    {
        fprintf(stderr, "Error Opening IUP.")
        return;
    }
    .
    .
    .
    IupMainLoop();
    IupClose();
}
```

Abstract Layout



Most interface toolkits employ the concrete layout model, that is, element positioning in the dialog is absolute in coordinates relative to the upper left corner of the dialog's client area. This makes it easy to position the elements on it by using an interactive tool usually provided with the system. It is also easy to dimension them. Of course, this positioning intrinsically depends on the graphics system's resolution. Moreover, when the dialog size

is altered, the elements remain on the same place, thus generating an empty area below and to the right of the elements. Besides, if the graphics system's resolution changes, the dialog inevitably will look larger or smaller according to the resolution increase or decrease.

IUP implements an abstract layout concept, in which the positioning of elements is done relatively instead of absolutely. For such, composition elements are necessary for composing the interface elements. They are boxes and fillings invisible to the user, but that play an important part. When the dialog size changes, these elements expand or retract to adjust the positioning of the elements to the new situation.

Watch the codes below. The first one refers to the creation of a dialog for the Microsoft Windows environment using its own resource API. The second uses IUP. Note that, apart from providing the specification greater flexibility, the IUP specification is simpler, though a little larger. In fact, creating a dialog on IUP with several elements will force you to plan your dialog more carefully – on the other hand, this will actually make its implementation easier.

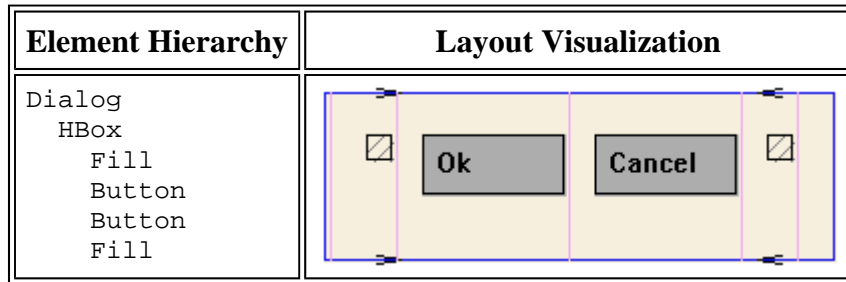
Moreover, this IUP dialog has an indirect advantage: if the user changes its size, the elements (due to being positioned on an abstract layout) are automatically re-positioned horizontally, because of the **iupfill** elements.

in Windows	in IupLua
<pre> dialogo DIALOG 0, 0, 108, 34 STYLE WS_MINIMIZEBOX WS_MAXIMIZEBOX WS_CAPTION WS_SYSMENU WS_THICKFRAME CAPTION "Título" BEGIN PUSHBUTTON "Ok", IDOK, 16, 9, 33, 15 PUSHBUTTON "Cancel", IDCANCEL, 57, 9, 33, 15 END </pre>	<pre> dialogo = iupdialog { iuphbox { iupfill{}, iupbutton{title="Ok",size="40"} iupbutton{title="Cancel",size=""} iupfill{} ;margin="15x15", gap="10" } ;title="Título" } </pre>
	

Now see the same dialog in LED and in C:

in LED	in C
<pre> DIALOG[TITLE="Título"] (HBOX[MARGIN="15x15", GAP="10"] (FILL(), BUTTON[SIZE="40"]("Ok",do_nothing), BUTTON[SIZE="40"]("Cancel",do_nothing), FILL())) </pre>	<pre> dialog = IupSetAttributes(IupDialog (IupSetAttributes(IupHbox (IupFill(), IupSetAttributes(IupButton("O IupSetAttributes(IupButton("C IupFill(), NULL), "MARGIN=15x15, GAP=10"),), "TITLE = Título") </pre>

Following, the abstract layout of this dialog:



The composition elements are vertical boxes (**vbox**), horizontal boxes (**hbox**) and filling (**fill**). There is also a depth box in which layers of elements can be created for the same dialog, and the elements in each layer are only visible when that given layer is active.

Elements

IUP contains several user interface elements. The library's main characteristic is the use of native elements. This means that the drawing and management of a button or menu is done by the native interface system, not by IUP. This makes the application's **appearance** more similar to that of other applications in that system. On the other hand, the application's appearance can vary from one system to another.

Even though IUP is not totally object oriented – because of the interface system's event mechanism, as well as other reasons – the elements follow a certain hierarchy. The creation process for an element occurs before the creation of the dialog in which that element will be inserted. Therefore, when the element is created, its **parent** is not known, but after the dialog is created all elements receive a parent. This mechanism is quite different from that of native systems, who first create the dialog and then the element, using the dialog as a parent. This feature creates some limitations for IUP, usually related to the insertion and removal of elements of an already mapped dialog.

Now we come to the notion of **mapping**. Since the elements are created differently from the native system, native elements can only be created after the dialog – and this can only happen after the programmer has called the **IupShow** function to show the dialog. We often need the elements to be created so we can use some other functionality of those elements before they are visible to the user. For that purpose, the **IupMap** function was created. It maps IUP's elements to native system elements. The **IupShow** function internally uses **IupMap** before showing the dialog on the screen, in case it has not been called.

Each element contains a unique creation function, and all of its management is done by means of **attributes** and **callbacks**, using only functions that can apply to all elements.

Attributes

Attributes are values associated to the elements and modified by means of functions **IupSetAttribute**, **IupSetAttributes** and **IupStoreAttribute**. The values passed for such functions are always strings. In C and in IupLua there are several string definitions, such as **IUP_YES**, which is actually "YES". In LED there is no need to add the prefix **IUP_** or quotation marks.

Since the attributes are strings, there are two functions to store them:

- `IupSetAttribute` stores only a pointer to the string and does not duplicate it.
- `IupStoreAttribute` duplicates the string, allowing you to use it for other purposes.

With `IupSetAttribute` you can also store particular application attributes. This can be very useful, for instance, used together with **callbacks** – which are global functions called by IUP when the user interacts with an interface element, and receive the element relative to the action as a parameter. For example, by storing a C pointer to some element's specific data, the user can retrieve it inside the callback through function `IupGetAttribute`. Therefore, even if the callbacks are global functions, the same callback can be used for several objects, even of different types.

Elements included in other elements inherit their attributes, unless an attribute is defined inside the element. This means there is an **inheritance** mechanism inside a given dialog. Therefore, when you consult the attribute of an undefined element, the inheritance mechanism will check the element containing it, and so forth, until it reaches the dialog. This means, for example, that if you set the `IUP_MARGIN` attribute of a `vbox` containing several other elements, including other `vboxes`, all the elements depending on it will be affected. Please note: not all attributes are inherited. Exceptions are: `IUP_TITLE`, `IUP_SIZE`, `IUP_VALUE`, `IUP_ALIGNMENT` and `IUP_FULLSCREEN`.

In IUP's documentation you will notice several **common attributes** to the elements. Some attributes that serve almost all elements are not mentioned in each one of the elements. We assume that the programmer knows they exist. In some cases, common attributes behave differently in different elements. In such cases, there are comments explaining the expected behavior. This also applies to the callbacks.

In Lua, the elements are implemented as tables, and the attributes can be accessed as indices. For further detail, please see the [Lua Binding](#) documentation.

Events

Events are handled through callbacks. Callbacks are functions the application can register to be called by IUP when a given user action occurs. Please refer to the above comment on the use of attributes and callbacks.

Even though callbacks have different purposes from attributes, they are actually associated to an element by means of an attribute. To associate a function to a callback, the user must employ the `IupSetAttribute` function, linking the action to a name (passed as a string). From this point on, this name will refer to a callback. By means of function `IupSetFunction`, the user connects this name to the callback.

For example:

```
IupSetAttribute(meuIhandle, IUP_ACTION,
"botao_foi_apertado");
IupSetFunction("botao_foi_apertado", (Icallback)
mybtppressed);
```

Therefore, callbacks also have some of the attributes' functionalities. The most important one is **inheritance**. Though many callbacks are specific to a given element, a callback can be set to a composition element, such as a **vbox**, which contains other elements, and

while the composition element does not call that callback all other elements contained in it will call the same callback, unless the callback is redefined in the element.

All callbacks receive at least the element which activated the action as a parameter.

The callbacks implemented in C by the application must return one of the below values:

- `IUP_DEFAULT`: Proceeds normally with user interaction. In case other return values do not apply, the callback should return this value.
- `IUP_CLOSE`: Makes the `IupMainLoop` function return the control to the application. Depending on the state of the application, `IUP_CLOSE` will close all windows.
- `IUP_IGNORE`: Makes the native system ignore that callback action. Applies only to some actions. Please refer to specific action documentation to know whether `IUP_IGNORE` applies to it or not.
- `IUP_CONTINUE`: Makes the element ignore the callback and pass the treatment of the execution to the parent element.

In Lua, the callbacks are implemented as methods, using the language's resources for object orientation. Thus, the element is implicitly passed as the **self** parameter and the functions do not need to return a value, since the binding is in charge of returning `IUP_DEFAULT`. Note that the callbacks in `IupLua` do not contain the suffix “_CB”. For further detail, see the [Lua Binding](#) documentation.

An important detail when using callbacks is that they are only called when the user actually executes an action over an element. A callback is not called when the programmer sets a value via `IupSetAttribute`. For instance: when the programmer changes a selected item on a list, no callback is called.

LED

LED is a dialog-specification language whose purpose is not to be a complete programming language, but rather to make dialog specification simpler than in C. IUP's binding for Lua was made *a posteriori* and completely replaces the LED files. Besides, Lua is a complete language, so a good deal of the application can be implemented with it. However, this means that the application must link its program to the Lua and to the `IupLua` libraries, as well as the IUP library.

The LED or Lua files are interpreted and can be sent together with the application's executable. However, this often becomes an inconvenience. To deal with it, there are the [LEDC](#) and the [LuaC](#) compilers.

In LED, attributes and expressions follow this form:

```
element[attribute1=value1,attribute2=value2,...]  
(...expression...)
```

The names of the elements must not contain the “iup” prefix. Attribute values are always interpreted as strings, but they need to be in quotes (“...”) only when they include spaces. The “IUP_” prefix must not be added to the names of the attributes and predefined values. Expressions contain parameters for creating the element.

In LED there is no distinction between upper and lower case, except for attribute names.

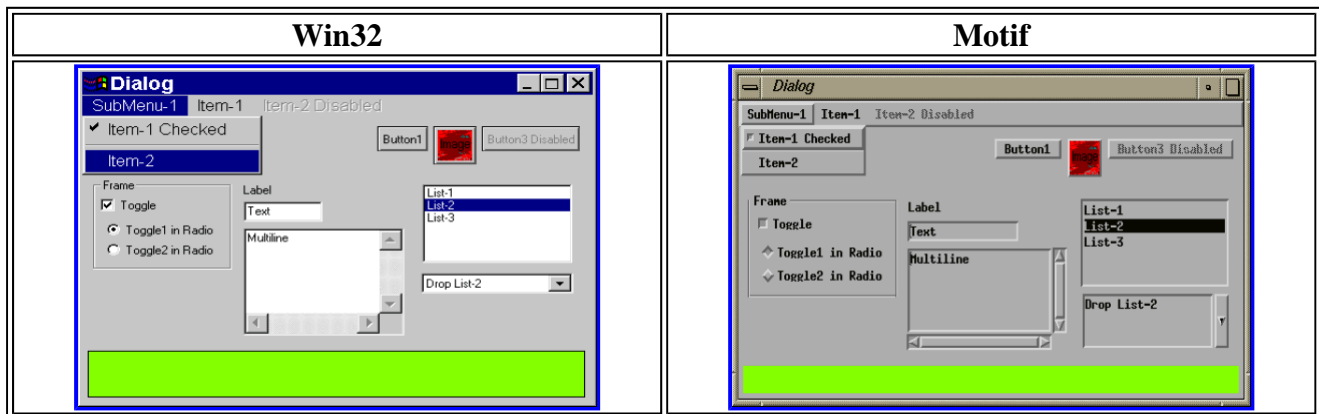
Though the LED files are text files, there is no way to interpret a text in memory – there is only the **IupLoad** function, which loads a LED file and creates the IUP elements defined in it. Naturally, the same file cannot be loaded more than once, because the elements would be created again. This file interpretation does not map the elements to the native system.

To simply view a LED file objects use the LED viewer application, see **IupView** in the applications included in the distribution.

Example

The following example creates a dialog with virtually all of IUP's elements as well as some variations of them, with some attributes changed. The same example is implemented in C, LED and Lua. Both screens presented are from the same example, one in Windows 95 and the other in IRIX.

in C	in LED	in IupLua
sample.c	sample.led	sample.lua



Portability

To compile programs in C, simply include file **iup.h**. If the application only uses functions from IUP and other portable languages such as C or Lua, with the same prototype for all platforms, then the application immediately becomes platform independent, at least concerning user interface, because the implementation of the IUP functions is different in each platform. The linker is in charge of solving the IUP functions using the library specified in the project/makefile. For further information on how to link your application, please refer to the specific driver documentation.

Generating Applications

The generation of applications is highly dependent on each system. Please refer to each of IUP drivers' documentation.

IUP can also work together with other interface toolkits. The main problem is the IupMainLoop function. If you are going to use only Popup dialogs, then it is very simple. But to use non modal dialogs without the IupMainLoop you must call IupLoopStep from inside your own message loop. Also it is not possible to use Iup controls with dialogs

from other toolkits and vice-versa.

There is also a guide on using the [Dev-C++ IDE Project Options](#) and [Visual C++ IDE Project Properties](#).

Dev-C++ IDE Project Options Guide

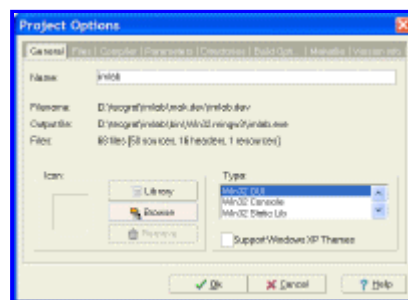
<http://www.bloodshed.net/devcpp.html>

"Bloodshed Dev-C++ is a full-featured Integrated Development Environment (IDE) for the C/C++ programming language. It uses Mingw port of GCC (GNU Compiler Collection) as it's compiler. Dev-C++ can also be used in combination with Cygwin or any other GCC based compiler."

It has many features, and integrated debug and it is free! To use IUP with Dev-C++ you will need to download the "mingw32" binaries in the download page.

After unpacking the file in your computer you must configure your Project Options. In the Project Options dialog there are 3 important places:

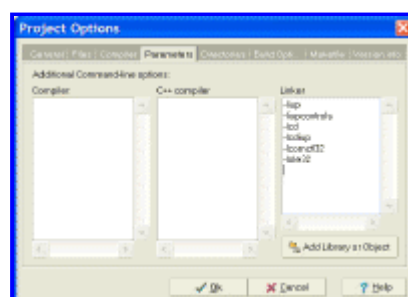
- General / Type - you can configure Win32 GUI or Win32 Console, but if you set to console it will always create a console screen behind your window when the program starts. Do not select "Support Windows XP Themes".



- Parameters / Linker - where you are going to list the libraries you use, for example:

```
-liup
-liupcontrols
-lcd
-lcdiup
-lcomctl32
-lole32
```

In this configuration you are using also the additional library of Controls that uses the [CD library](#), also available at the download page.



- Directories / Library Directories and Include Directories - where you are going to list the include path, for example:

```

..\..\iup\lib\mingw3
..\..\cd\lib\mingw3
or
c:\tecgraf\iup\lib\mingw3
c:\tecgraf\cd\lib\mingw3

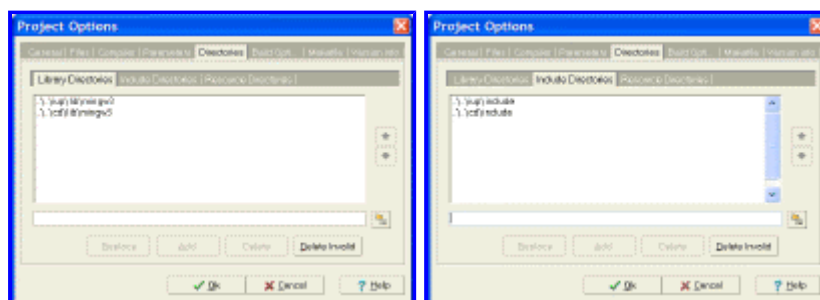
```

And:

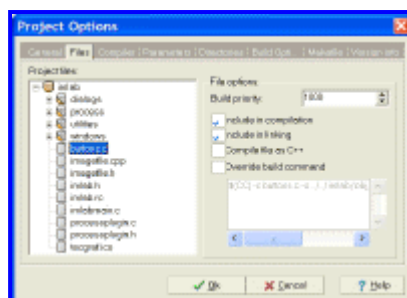
```

..\..\iup\include
..\..\cd\include
or
c:\tecgraf\iup\include
c:\tecgraf\cd\include

```



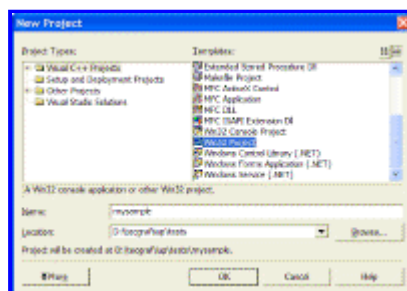
In some cases the IDE may force the compilation of C files as C++. If do not want that then uncheck the option in the settings for each file. Still in the Project Options dialog, in the Files tab, select the file and uncheck "Compile File as C++".



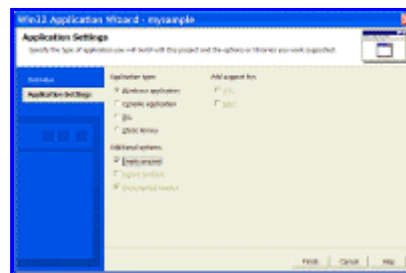
Visual C++ IDE Project Properties Guide

This guide was built using Microsoft Visual Studio .NET 2003, which includes Visual C++ 7.1.

To create a new project go to the menu "File / New / Project":



Select "Win32 Project" on the Templates. Before finishing the Wizard, select "Application Settings". Mark "Windows application" and "Empty project".

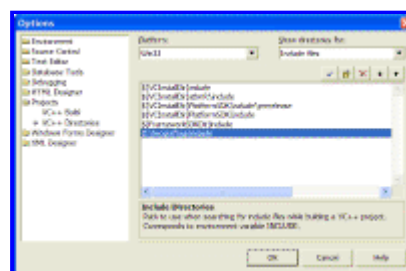


You can also create a "Console application", and whenever you execute your application a text console will also be displayed. But this is a very useful situation so you can use the standard C printf function to display textual information for debugging purposes.

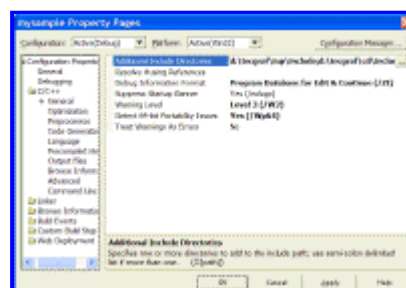
Then add your files in the menu "Project / Add New Item" or "Project / Add Existing Item".

After creating the project you must configure it to find the IUP includes and libraries. In Visual Studio there are two places where you can do this.

One is in the menu "Tools / Options", then select "Project / Visual C++ Directories". Select "Include Files" or "Library Files" in "Show directories for:". In this dialog you will configure parameters that will affect all the projects you open.



Or you can configure the parameters only for the project you created. In this case go to the menu "Project / Properties". To configure the include files location select "C/C++ / General" in the left tree, then write the list of folders separated by ";" in "Additional Include Directories".



To configure the library files location select "Linker / General" in the left tree, then write the list of folders separated by ";" in "Additional Library Directories".

applications the necessary tools so that versions for different platforms can be generated homogeneously. In other words, the **tecmake** user does not need to create a makefile nor be concerned with his/her current platform to create a library or application, which we will here call *product*.

To build **IUP**, first install [Tecmake](#). Note that **Tecmake** defines names for each system. For example: **vc7** (Visual Studio 7) or **Linux24g3** (Linux Kernel 2.4 with gcc 3.x.)

When installing Tecmake you will need to set a few environment variables. You should refer to **Tecmake's** manual, but here are a few tips on how to install it (tested on Redhat 7.0):

- Set environment variables **TCG_HOME** and **TECMAKE_HOME** pointing to where **tecmake.mak** is installed;
- Set variables **TEC_UNAME**, **TEC_SYSNAME**, **TEC_SYSRELEASE**. These are well explained in [Tecmake's manual](#) under Platforms.

In **IUP's** main directory there is a file named *make_uname* (*make_uname.bat* in Windows) that calls **Tecmake** for each **IUP** library. To build **IUP** for Windows using Visual C 7.0 for example, just execute *make_uname.bat vc7*

In order to build **IUP** on Unix systems, you should have **Motif** version 2.1 or greater.

Please send any comments or questions to iup@tecgraf.puc-rio.br

Lua Binding

Overview

The Lua Binding is an interface between the Lua language and IUP, a portable user-interface system. The main purpose of this package is to provide facilities for constructing IUP dialogs using the Lua language. Abstractions were used to create a programming environment similar to that of object-oriented languages, even though Lua is not one of such languages. The concept of event-oriented programming is broadly used here, because the IUP library is based on this model. Most constructions used in IupLua were strongly based on the corresponding constructions in LED.

System Control

Before running any function from the Lua Binding, you must run the **iuplua_open** function to initialize the toolkit. This function must be run after a call to function **IupOpen**. All this is done in C in Lua's host program.

Example:

```
int main(void)
{
    IupOpen();
    IupControlsOpen();

    /* Lua 3.2 initialization (could be Lua 4.0 or Lua 5.0) */
    lua_open();
    lua_iolibopen();
    lua_strlibopen();
    lua_mathlibopen();
}
```

```

iuplua_open();          /* Initialize Binding Lua */
controlslua_open(); /* Initialize CPI controls binding Lua */

IupMainLoop();
IupControlsClose();
IupClose();

return 0;
}

```

Generating Applications

To use the Lua Binding, you need to link the program to the **IupLua** library and to the **Lua** library. IupLua is available in Lua 3.2 and Lua 5.0.

Simple Attributes

Each interface element is created as a Lua table, and **its attributes are indicated as fields in this table**. Some of these attributes are directly transferred to IUP, so that any changes made to them immediately reflect on the screen. By means of Lua's *tag method* system, attribute values defined in IupLua can be transferred to IUP, being immediately updated. However, not all attributes are transferred to IUP. Some are control attributes, such as `handle`, which stores the handle of the IUP element, and `parent`, which stores the object immediately above in the class hierarchy. Attributes that receive strings or numbers as values are immediately transferred to IUP. Other values (such as functions or objects) are stored in IupLua and might receive special treatment (as will be explained later).

For instance, a button can be created as follows (defining a title and the background color):

```
ok = iupbutton{title = "Ok", bgcolor = "0 255 0"}
```

Font color can be subsequently changed by modifying the value of attribute `fgcolor`:

```
ok.fgcolor = "255 0 0"
```

Note that the **attribute names** in C and in IupLua are the same, but in IupLua they can be written in lower case. The names of element creation functions are also in lower case, since they are actually constructors of Lua tables.

Some parameters are required attributes (such as `title` in buttons). Their types are checked when the element is created. The required parameters are exactly the parameters that are necessary for the element to be created in C.

Some interface elements contain one or more other elements, as is the case of dialogs, lists and boxes. In such cases, the object's element list is put together as a vector, that is, the elements are placed one after the other, separated by commas. They can be accessed by **indexing the object** containing them, as can be seen in this example:

```

box = iuphbox{bt1, bt2, bt3}
box[1].fgcolor = "255 0 0"          -- changes bt1 foreground color
box[2].fgcolor = caixa[1].fgcolor  -- changes bt2 foreground color

```

Other Attributes

While the attributes receiving numbers or strings are directly transferred to IUP, attributes receiving other interface objects are not directly transferred, because IUP only accepts strings as a value. The metamethod that transfers attributes to IUP verifies if the attribute value is of a “widget” type, that is, if it is an interface element. If the element already has a name, this name is passed to IUP. If not, a new name is created, associated to the element and passed to IUP as the value of the attribute being defined.

This policy is very useful for associating two interface elements, because you can abstract the fact that IUP uses a string to make associations and imagine the interface element itself is being used.

Callbacks

Some attributes are not directly transferred to IUP. They are either control attributes or attributes in charge of treating the actions associated to objects. Since the use of actions requires registering functions in C to be called when the event occurs, there is a differentiated treatment for such attributes. The IupLua system does not require the creation and registration of C functions for this purpose.

Callbacks of different types of interface events are registered by the library when they are initialized. These default callbacks call methods of the object receiving the event. Each different event calls a different method, which can have a few parameters. The objects are initialized with none of these methods set, so the programmer is in charge of setting them when required. They receive the same parameters as callbacks in C, in the same order, and **they can either return a value or not** (if no return value is set, the IUP_DEFAULT value is returned). The following example shows the definition of an action for a button.

```
function ok:action ()
    local aux = self.fgcolor
    self.fgcolor = self.bgcolor
    self.bgcolor = aux
end
```

Or you can do

```
function myaction(self)
    ...
end

ok.action = myaction
```

The IUP API binding

Even though there are sintatic sugars used to handle callbacks and attributes in Lua, most of the functions defined in C are exported to Lua, such as IupSetAttribute, IupGetBrother among others.

Exchanging Values between C and Lua

Each binding to a version of Lua uses different features of the language in order to implement IUP handles (Ihandle) in Lua. Therefore, functions have been created to help exchange references between Lua and C.

To push an Ihandle in Lua's stack, use the function:

```
iuplua_pushihandle(lua_State *L, Ihandle *n);
```

In Lua 3.2, the `lua_State` parameter does not exist.

To receive an `Ihandle` in a C function called from Lua, just use one of the following functions according to which Lua you are using: `lua_getuserdata` (Lua 3.2), `lua_touserdata` (Lua 4) or `lua_unboxpointer` in (Lua 5).

In order to bring IUP handles created in C to Lua, the user can give the IUP handle a name by means of `IupSetHandle` and call in Lua the function `IupGetFromC`.

Ex:

```
lua_ihandle = IupGetFromC{"element_name"}
```

where `element_name` is the name of the element previously defined with `IupSetHandle`.

Error Handling

Error handling differ between each Lua version. To keep IupLua's API as compatible as possible, functions have been created to execute Lua code:

```
int iuplua_dofile(lua_State *L, char *filename);
int iuplua_dostring(lua_State *L, const char *string, const char *chunk_
```

If the given functions are used, in every IupLua version the errors will be reported through the `_ERRORMESSAGE` function. If this function is not defined by the user, IUP will use its default implementation (shows a dialog with the error message.)

If the user chooses not to use those functions, errors will be handled according to the version of Lua used.

The Architecture Behind IupLua

The Lua API for the IUP system was based on object classes representing the different interface elements. A hierarchy was built among these classes, with the main purpose of reusing code. Code inheritance was implemented precisely as described in the [Lua](#) user guide.

The root of this hierarchy is the `WIDGET` class. It contains the basic procedures for construction, parameter type verification, and allocation of structures for controlling IUP's interface elements. This class also defines the basic parameters of all classes, such as `handle` (which stores the handle of the associated IUP element) and `parent` (used to implement the inheritance mechanism).

Even though almost all classes directly descend from the `WIDGET` class, some other classes serve as mediators in the tree. This is the case of the `COMPOSITION` class, located among the composition element classes: `IUPHBOX`, `IUPVBOX` and `IUPZBOX`.

Some classes use part of the code from other classes, when they are very similar. This happens to `IUPITEM` and `IUPTOGGLE`, which reuse the code related to the verification of parameter types and to the definition of the `action` callback in the `IUPBUTTON` class. Class `IUPMULTILINE` inherits several characteristics from `IUPTEXT`, such as the

definition of the `action` callback and the verification of parameter types.

The complete class hierarchy can be represented as follows:

```

WIDGET
  IUPBUTTON
    IUPITEM
    IUPTOGGLE
  IUPCANVAS
  COMPOSITION
    IUPHBOX
    IUPVBOX
    IUPZBOX
  IUPDIALOG
  IUPFILL
  IUPFRAME
  IUPIMAGE
  IUPLABEL
  IUPLIST
  IUPMENU
  IUPRADIO
  IUPSEPARATOR
  IUPSUBMENU
  IUPTEXT
    IUPMULTILINE

```

Differences in IupLua5

In IupLua5 we follow the same organization of the Lua libraries using the namespace before all the definitions.

- All exported functions are accessed only through **iup.FunctionName**, including control initialization like **iup.label**.
- All callbacks are now accessed through their exact name in the C API.
- Numeric definitions were kept in upper case by without the IUP_ prefix, like: `iup.DEFAULT`.
- String definitions for values are no longer supported, always use "YES", "NO", "ACENTER", etc.

IupLua3 Examples

- [MultiList](#) - Creates a matrix that allows selection of each line at a time.
- [TableTree](#) - Shows a tree given a Lua table.
- [LabelText](#) - Creates a pair Label-Text.
- [AllFonts](#) - Allows you to easily select a font from all possible IUP fonts.

IupLua Test Application

The distribution files include two executables, one for Lua 3 and one for Lua 5, that you can use to test your Lua code. Both applications have support for all the additional controls.

CPI

Introduction

The IUP toolkit was originally designed to support a set of well-defined controls existing in all the destination platforms. With the evolution of native systems (e.g. Windows 95) and new requests from users, IUP needed to evolve with the purpose of making the addition of new interface elements to the toolkit easier.

Thus, to support the development of new controls for IUP, a specific API (Application Program Interface) was created for this purpose: it was called CPI (Control Program Interface). This new API is orthogonal to the original IUP API, that is, its use with a client application does not interfere with the conventional use of IUP. Only a developer wishing to implement a new IUP control is required study this API.

Control Implementation

To create a new CPI control, follow these steps:

- [Initialize the control](#)
- [Create control instances](#)
- Implement the [CPI methods](#) associated to the control
- Make exported information available to the user (function prototypes, definitions, etc.)

General Control Initialization

The initialization function is in charge of passing IUP the necessary information so that the control can be used. Such information is grouped in an `Iclass`-type structure, which from now on is to be called the class of the control.

The first step is to create the structure. This is done by calling the [iupCpiCreateNewClass](#) function. To this function, two pieces of information must be passed: a string identifying the control in a unique way (the "name" of the control), and a string describing the creation parameters when the control is created via LED. The pointer returned by [iupCpiCreateNewClass](#) must be stored in a static module variable, as it will be necessary to create new control instances.

Next step is to replace, if necessary, the control's [CPI method](#). This is done through function [iupCpiSetClassMethod](#), which receives the control's class as an argument and the pointer to the new method.

Important: Function [iupCpiCreateNewClass](#) fills the class with [default methods](#), which provide the control a default behavior.

This initialization function must be named `IupControlOpen`, where *Control* is the name of the control.

Example (class creation for a control named Dial):

```
#include <iup.h>
#include <iupcpi.h>

static Iclass *classe = NULL;

static Ihandle *DialCreate(...) /* método de criação */
{
    ...
}
```

```

}

void IupDialOpen(void)
{
    classe = iupCpiCreateNewClass("dial","n");

    iupCpiSetClassMethod(classe, ICPI_CREATE, DialCreate);
}

```

Creation of Control Instances

The created control must make a function available whose name must be `IupControl`, where *Control* is the control name. This function is to be used by the user to create a new control instance, and should not receive arguments. If the control is a container, the the arguments are necessarily its children.

In this creation function, if no parameters are necessary, just call [IupCreate](#) with the control's name. If the control allows children, use [IupCreatev](#) to pass them forward to IUP. This function will create the control's `Ihandle`, by means of the function registered by `ICPI_CREATE`.

Example1:

```

Ihandle *IupDial()
{
    return IupCreate("dial");
}

```

Example2:

```

Ihandle *IupBox (Ihandle * first,...)
{
    Ihandle **params = NULL;
    Ihandle *elem = NULL;
    unsigned int i = 0;
    va_list arg;

    if(first)
    {
        va_start (arg, first);
        i = 1;
        while (va_arg(arg, Ihandle *)) i++;
        va_end (arg);

        params = (Ihandle**) malloc (sizeof (Ihandle*) * (i+1));

        i = 0;
        va_start (arg, first);
        elem = first;
        while (elem != NULL)
        {
            params[i++] = elem;
            elem = va_arg(arg, Ihandle *);
        }
        params[i] = NULL;
        va_end (arg);
    }

    elem = IupCreatev(name, params);
}

```

The `Iclass` structure fields are mostly pointers to functions to be called by IUP in certain moments. These pointers to functions play the same parts as objects in languages such as C++. Following the C++ philosophy, the CPI defines a set of functions which can be used to provide the controls a default behavior. The `Iclass` structure stores these function pointers, which are defined right after the call to `iupCpiCreateNewClass`.

<u>iupCpiSetClassMethod</u> parameter Default method used	Description
ICPI_CREATE iupCpiDefaultCreate	This method is called by IUP when an instance for a control is created. When this function is called, IUP already has a reference to the control's instance (represented by the <code>self</code> parameter). The caller must provide all the required attributes, specified in the call to the <code>IUP_CreateControl</code> function. Prototype: Ihandle *(*create) (Iclass* class)
ICPI_DESTROY iupCpiDefaultDestroy	This method is called by IUP when the <code>IupDestroyControl</code> function destroys a control or its dialog. If necessary, this method can free any memory structures maintained by the control. Prototype: void (*destroy) (Ihandle* self);
ICPI_MAP iupCpiDefaultMap	This method is called by IUP to map the control into a window. The <code>map</code> parameter indicates of which window the control will be mapped (it can be a dialog or any other control.) Prototype: void (*map) (Ihandle* self, Ihandle* parent)
ICPI_UNMAP	This method is called by IUP to "destroy" the control's mapping without removing the control from the control hierarchy.

iupCpiDefaultUnmap	Prototype: void (*unmap) (Ihandle* self);
ICPI_SETNATURALSIZE iupCpiDefaultSetNaturalSize	<p>This method is called by IUP for the control to sp function must call functions iupSetNaturalWidth implementation might call the function associated to the Iclass structure (to be described further on) to control. This function must return the same value associated to the getsize field.</p> <p>Prototype: int (*setnaturalsize) (Ihandle* self, int w, int h);</p>
ICPI_SETCURRENTSIZE iupCpiDefaultSetCurrentSize	<p>This method is called by IUP for the control to sp function must call functions iupSetCurrentWidth and iupSetCurrentHeight. Parameters w (width in pixels) and h (height in pixels) are the values the control dimensions can have.</p> <p>Prototype: void (*setcurrentsize) (Ihandle* self, int w, int h);</p>
ICPI_SETPOSITION iupCpiDefaultSetPosition	<p>This method is called by IUP for the control to set the position of the window. Parameters x and y represent the position (x, y) the control must have, computed by IUP. The method need only be changed if the control has a position attribute.</p> <p>Prototype: void (*setposition) (Ihandle* self, int x, int y);</p>
ICPI_SETATTR iupCpiDefaultSetAttr	<p>This method is called to provide a new value to a control attribute. When the method is called, the attribute's value is already updated in the environment. The received parameters mean the attribute name being changed; value is the new attribute value.</p> <p>Prototype: void (*setattr) (Ihandle* self, char* attr, int value);</p>
ICPI_GETATTR iupCpiDefaultGetAttr	<p>This method is called by IUP to verify the value of a control attribute by parameter attr. This method is called before the attribute is updated in the environment. If this method returns null, IUP verifies the value in the environment. If this check also returns null, the getdefaultattr field is called.</p> <p>Prototype: char* (*getattr) (Ihandle* self, char* attr);</p>
ICPI_GETDEFAULTATTR iupCpiDefaultGetDefaultAttr	<p>This method is called by IUP when verifying an attribute value and the method related to the getattr field and the attribute environment fail (returned null).</p>

	Prototype: <code>char* (*getdefaultattr) (Ihandle*</code>
ICPI_GETSIZE <code>iupCpiDefaultGetSize</code>	This method is called by IUP when wishing to ve This function must write to the w and h paramete respectively. The return value for this function ca 0 - The control size does not vary when the dialo 1 - The control width may vary when the dialog 2 - The control height may vary when the dialog 3 - The control width and height may vary when Prototype: <code>int (*getsize) (Ihandle *self, in</code>
ICPI_POPUP -	This method is called by IUP when wishing the c The x and y parameters indicate the position the method must return IUP_NOERROR, if no error c occurs. Prototype: <code>int (*popup) (Ihandle *self, int</code>

Header File

For a IUP application to use the new control, the prototypes of the initialization functions and the definitions of the new attributes must be made available. This is done by means of a header file, which must have the same name as the control.

Function Prototypes

The prototypes of all control functions that might be used by the control's client applications must be provided. Usually there are only two prototypes: the initialization function and the instance creation function. It is important to consider that the control might be used in applications both in C and in C++. Therefore, the prototype declaration should be involved by a "C" extern block (see example below).

Attribute Definition

To match the IUP standard, macros must be defined to reference the strings identifying the new attributes used by the control. For example, if a new control has an attribute named MODE used to identify its operation mode, then the following macro must be defined:

```
#define IUP_MODE "MODE"
```

Note that the attributes used by the control may have already been defined (in another control's header, for instance). Thus, it is advisable to check if this happens to avoid compilation errors. Also refer to section [Attribute Treatment](#).

Example:

```

/*
 * IupControle.h
 */

#ifndef IUPCONTROLE_H
#define IUPCONTROLE_H

#ifdef __cplusplus /* necessário quando controle é utilizado
em código C++ */
extern "C" {
#endif

void IupControleOpen(void);
Ihandle *IupControle(...);

#ifdef __cplusplus
}
#endif

/* Novos atributos */

#ifndef IUP_MODE
#define IUP_MODE "MODE"
#endif

#ifndef IUP_LENGTH
#define IUP_LENGTH "LENGTH"
#endif

#endif /* IUPCONTROLE_H */

```

The Iclass Structure

The `Iclass` structure stores pointers to the control's methods (described in the table above) as well as the following fields:

- `char* name :`

Stores the name given to the control. This name allows the control to be used in LED.

- `char* format :`

Format string used to describe the required attributes defined in LED to create a control instance. If this field is null, then the new control type has no required attributes. The format string can be any sequence with the following characters:

Character	Meaning
n	name of a control instance or an action
s	any string
c	interface control (Ihandle *)
	from this character on, a list of control-instance names or a list of

N	actions will be passed
S	from this character on, a list of strings will be passed
C	from this character on, a list of interface controls will be passed (Ihandle *)

Attribute Treatment

By default, a control inherits the same attributes defined for IUP's Canvas element.

The developer of a new control can define new attributes. For such, he/she must redefine the `ICPI_SETATTR`, `ICPI_GETATTR` and `ICPI_GETDEFATTR` methods, if necessary.

If any attribute-manipulation method is redefined, the standard procedure is:

- identify if the given attribute is part of the set of attributes that must receive any special treatment by the new control;
- if the attribute is part of this set, then the adequate treatment must be provided;
- if not, the [default method](#) must be called to treat the attribute.

Default Methods

In this section, we present the set of functions corresponding to the default behavior of an interface element's methods. Such functions can either be used to fill an [Iclass](#) structure or be called by a new method to execute the default procedure (either before or after the specialized treatment of the new control is executed).

Function corresponding to the `setnaturalsize` method:

```
int ctrsetnaturalsize (Ihandle* self);
```

Function corresponding to the `setcurrentsize` method:

```
void ctrsetcurrentsize (Ihandle* self, int w, int h);
```

Functions corresponding to the `getsize` method:

```
int ctrgetsize (Ihandle* self, int* w, int* h); ou
int ctrgetsizevar (Ihandle* self, int* w, int* h);
```

Note: A control that associates the `ctrgetsize` function to its `getsize` method cannot vary in size when its dialog's size varies. On the other hand, a control that uses function `ctrgetsizevar` can vary in size according to the changes made to its dialog size.

Function corresponding to the `setposition` method:

```
void ctrsetposition (Ihandle* self, int w, int h);
```

Function corresponding to the `create` method:

```
void ctrcreate (Ihandle* self, void** array);
```

Function corresponding to the `destroy` method:

```
void ctrdestroy (Ihandle* self);
```

Function corresponding to the `map` method:


```
void ctrmap (Ihandle* self, Ihandle* parent);
```

Function corresponding to the unmap method:

```
void ctrunmap (Ihandle* self);
```

Function corresponding to the setattr method:

```
void ctrsetattr (Ihandle* self, char* attr, char* value);
```

Function corresponding to the getattr method:

```
char* ctrgetattr (Ihandle* self, char* attr);
```

Function corresponding to the getdefaultattr method:

```
char* ctrgetdefaultattr (Ihandle* self, char* attr);
```

Extra Functions

Following we provide a list of extra IUP functions. They can be used by programmers wishing to create a new control:

Main control-creation functions

```
Iclass *iupCpiCreateNewClass(char *name, char *format);
int iupCpiSetClassMethod(Iclass *ic, char *method, Imethod
func);
```

Functions that call the method in charge for the given action

```
int iupmethSetNaturalSize( Ihandle* self );
void iupmethSetCurrentSize( Ihandle* self, int w, int h );
int iupmethGetSize( Ihandle* self, int* w, int* h );
void iupmethSetPosition ( Ihandle* self, int w, int h );
void iupmethCreate( Ihandle* self, void** array );
void iupmethDestroy( Ihandle* self );
void iupmethMap( Ihandle* self, Ihandle* parent );
void iupmethUnmap( Ihandle* self );
void iupmethSetAttribute( Ihandle* self, char* attr, char*
value);
char* iupmethGetAttribute( Ihandle* self, char* attr );
char* iupmethGetDefaultAttr( Ihandle* self, char* attr );
char* iupmethGetClassName( Ihandle* self );
```

Functions in charge of manipulating an element's size

```
void iupSetCurrentWidth(Ihandle* self, int w);
void iupSetCurrentHeight(Ihandle* self, int h);
int iupGetCurrentWidth(Ihandle* self);
int iupGetCurrentHeight(Ihandle* self);
int iupHorResizable(Ihandle* self);
int iupVertResizable(Ihandle* self);
void iupSetNaturalWidth(Ihandle* self, int w);
void iupSetNaturalHeight(Ihandle* self, int h);
void iupGetCharSize(Ihandle* n, int *w, int *h);
int iupGetSize(Ihandle* e, int* w, int *h);
```

```
void iupGetTextSize(Ihandle* h, char* text, int* size);
void iupdrvResizeObjects(Ihandle *n);
```

Example

The best example possible can be taken from IUP distribution. As an advice, please refer to the `iupgauge` control.

LED Compiler for C

Description

The LED compiler (**ledc**) generates a C module from one or more LED files. The C module exports only one function, which builds the IUP interface described in the LED files. Running this function is equivalent to calling the `IupLoad` function over the original LED files.

One advantage of using the compiler is that it allows the application to be independent from LED files during its execution. Since the interface description is inside the executable file, there is no need to worry about locating the configuration files.

Another advantage is that **ledc** performs a stricter verification than IUP's internal parser. This makes error detection in LED files easier.

Finally, running the function generated by the compiler is faster than reading the corresponding LED file with `IupLoad`, since the parsing step of the LED file is transferred from execution to compilation. However, creating the IUP elements described in LED takes most of the execution time of the `IupLoad` function, so the gain in efficiency may not be very significant.

Usage

```
ledc [-v] [-c] [-f funcname] [-o file] files
```

<code>-v</code>	shows ledc 's version number
<code>-c</code>	does not generate code, just checks for errors in the LED files
<code>-f funcname</code>	uses <funcname> as the name of the generated exported function (default: <code>led_load</code>)
<code>-o file</code>	uses <file> as the name of the generated file (default: <code>led.c</code>)

Error Messages

Several warnings and error messages might be generated during compilation. Errors abort the compilation. The messages can be the following:

```
warning: undeclared control name (argument number)
```

The *name* name was used as an argument where a IUP element was expected, but no element with this name was previously declared.

warning: string expected (argument *number*)

A name (callback?) was passed as a parameter for a string-type argument.

warning: callback expected (argument *number*)

A string was passed as a parameter for a callback-type argument.

warning: unknown control *name* used

An unknown element, called *name*, was used. The compiler assumes the element's creation function is called `IupName`, with *name* capitalized, and assumes the arguments' types based on what was passed on LED.

warning: *elem* declared without a name

An *elem*-type element was declared without being associated to any name. This declaration creates the element, but it will not be accessible, so it cannot be used.

element *name* already used in line *number*

The *name* element was already used in line *number*. In IUP, the same element cannot have more than one parent.

too few arguments for *name*

The *name* element expects more arguments than those already passed.

too many arguments for *name*

The *name* element expects less arguments than those passed.

name is not a valid child

The *name* element cannot be used as a parameter in this case. This happens when trying to insert an image into a vbox, for instance.

control expected (argument *number*)

A string was passed as a parameter for an element-type argument.

string expected (argument *number*)

An element was passed as a parameter for a string-type argument.

number expected (argument *number*)

An element or a string was passed as a parameter for a number-type argument.

callback expected (argument *number*)

An element was passed as a parameter for a callback-type argument.

hotkeys not implemented

Even though it is a LED word reserved to an element, it is not implemented.

Functions

The internal IUP functions allow the user to read and set values, set callbacks, destroy elements, show and hide dialogs, etc., deciding how the library must work.

System Control

Allows initializing the library, controlling the execution order of tasks, and using help

through Netscape.

System Control for Lua

Integrates the library with programs and dialogs written in Lua.

Dialog and Menu Control

Controls the visualization of dialogs and menus, and controls focus, mapping and destruction.

LED

Allows loading an interface-description file written in LED.

Element Composition

Helps adding and removing elements.

Element Manipulation

Allows creating and reading internal references of elements.

Attribute Manipulation

Creates, modifies and reads attributes of each element or global. Allows the attributes to return in different formats.

Callback Manipulation

Allows reading and creating references to functions.

Extra Lua Functions

Auxiliary functions in IupLua.

GKS

Auxiliary functions that help integrating GKS and IUP.

IupOpen

Initializes the IUP toolkit. Must be called before any other IUP function.

Parameters/Return

```
int IupOpen(void); [in C]
[There is no equivalent in Lua]
```

This function returns IUP_ERROR or IUP_NOERROR.

Notes

The IupOpen function in the Windows driver initializes OLE through the function OleInitialize; IupClose calls OleUninitialize.

The toolkit's initialization depends on several platform-dependent environment variables.

For a more detailed explanation on the system control, please refer to [Guide / System Control](#).

Lua Binding

Lua: This function must be called by the host program and before the Binding Lua initialization function, **iuplua_open**.

See Also

[iuplua_open](#), [IupClose](#), [Guide / System Control](#)

IupClose

Ends the IUP toolkit.

Parameters/Return

```
void IupClose(void); [in C]
[There is no Lua equivalent]
```

Notes

The IupOpen function in the Windows driver initializes OLE through the function OleInitialize; IupClose calls OleUninitialize.

Lua Binding

This function must be called by the host program.

See Also

[IupOpen](#)

IupMainLoop

Executes the user interaction until a callback returns IUP_CLOSE. Must be called before the IupClose function.

Parameters/Return

```
int IupMainLoop(void); [in C]
IupMainLoop() -> ret: number [in IupLua3]
iup.MainLoop() -> ret: number [in IupLua5]
```

Returns IUP_NOERROR or IUP_ERROR.

Notes

If this function is executed at any other moment, it will interrupt the execution until a callback returns `IUP_CLOSE`. A second execution of **IupMainLoop** will have a platform-dependent behavior.

Presently, the return value can be ignored, as in all platforms it currently returns `IUP_NOERROR`.

The message loop will go on only while there is a dialog. At the moment the last dialog is destroyed or hidden, the loop will be ended and **IupMainLoop** will return the control to the application, except if the `Idle` callback is defined - in this case, the `Idle` callback must return `IUP_CLOSE` for the application to receive the control back.

Motif Driver

Can be executed several times but a `IUP_CLOSE` must occur for each execution.

Win32 Driver

If the function is executed several times, only one `IUP_CLOSE` will end all executions.

See Also

[IupOpen](#), [IupClose](#), [IupLoopStep](#), [Guide / System Control](#), [IDLE_ACTION](#).

IupLoopStep

Runs an iteration of the message loop.

Parameters/Return

```
int IupLoopStep(void); [in C]
IupLoopStep() -> ret: number [in IupLua3]
iup.LoopStep() -> ret: number [in IupLua5]
```

This function returns `IUP_CLOSE` or `IUP_DEFAULT`.

Notes

This function is useful for allowing a second message loop to be managed by the application itself. This means that messages can be intercepted and callbacks can be processed inside an application loop.

An example of how to use this function is a counter that can be stopped by the user. For such, the user has to interact with the system, which is possible by calling the function periodically.

This way, this function also replaces some old mechanisms implemented using the `Idle` callback.

Note that this function does not replace **IupMainLoop**.

See Also

[IupOpen](#), [IupClose](#), [IupMainLoop](#), [IDLE_ACTION](#), [Guide / System Control](#)

IupFlush

Processes all pending messages in the message queue.

Parameters/Return

```
void IupFlush(void); [in C]
IupFlush() [in IupLua3]
iup.Flush() [in IupLua5]
```

Notes

When you change an attribute of a certain element, the change may not take place immediately. For this update to occur faster than usual, run `IupFlush` after the attribute is changed.

Important: A call to this function may cause other callbacks to be processed before its returns.

IupHelp

Opens the given URL. In UNIX executes Netscape passing the desired URL as a parameter. In Windows calls the default application that handle URLs.

In UNIX you can change the used browser setting the environment variable `IUP_HELPAPP`. If set it will replace "netscape".

Parameters/Return

```
void IupHelp(char* url); [in C]
IupHelp(url: string) [in IupLua3]
iup.Help(url: string) [in IupLua5]
```

url: may be any kind of address accepted by the Browser, that is, it can include 'http://', or be just a file name, etc.

IupSetLanguage

Defines the language used by IUP.

Parameters/Return

```
void IupSetLanguage(char *lng); [in C]
IupSetLanguage(language :string) [in IupLua3]
iup.SetLanguage(language :string) [in IupLua5]
```

lng: Language to be used. Can have one of the following values:

- "ENGLISH"
- "PORTUGUESE"

default: "PORTUGUESE".

Affects

All elements that have predefined texts.

Example in C

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include "iup.h"

void main(void)
{
    IupOpen();
    IupSetLanguage("ENGLISH");
    IupMessage("IUP Language", IupGetLanguage());
    IupClose();
    return;
}
```

IupGetLanguage

Verifies the language used by IUP.

Parameters/Return

```
char* IupGetLanguage(void); [in C]
IupGetLanguage() -> (language: string) [in IupLua3]
iup.GetLanguage() -> (language: string) [in IupLua5]
```

For a list of all possible return values, see [IupSetLanguage](#).

Affects

All elements with predefined texts.

Example in C

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include "iup.h"

void main(void)
{
    IupOpen();
    IupMessage("IUP Language", IupGetLanguage());
    IupClose();
    return;
}
```


IupMapFont

Retrieves the name of a native font, given the name of the IUP font.

Parameters/Return

```
char* IupMapFont(char *iupfont); [in C]
IupMapFont(iupfont : string) -> (nativefont : string) [in IupLua3]
iup.MapFont(iupfont : string) -> (nativefont : string) [in IupLua5]
```

This function returns the name of the native font.

See Also

[IupUnMapFont](#), [FONT](#) attribute

IupUnMapFont

Retrieves the name of the IUP font, given the native font.

Parameters/Return

```
char* IupUnMapFont(char *font); [in C]
IupUnMapFont(font :string) -> (iupfont : string) [in IupLua3]
iup.UnMapFont(font :string) -> (iupfont : string) [in IupLua5]
```

This function returns the name of the IUP font, given the native font. If such font does not exist, the function will return NULL.

See Also

[IupMapFont](#), [IUP_FONT](#)

iuplua_open

Initializes the Lua Binding. This function must be called by the host program before running any Lua functions, but it is important to call it after **IupOpen**.

Parameters/Return

```
void iuplua_open(void); [in C]
[There is no equivalent in Lua]
```

Note

For a more detailed explanation on the system control for the Lua Binding, please refer to [Lua Binding / System Control](#).

See Also

[IupOpen](#), [Guide / System Control](#)

iupkey_open

Allows IUP keyboard definitions to be used in IupLua. This function must be run by the host program after **iuplua_open**. Please refer to the [Keyboard Codes](#) table for a list of possible values.

Parameters/Return

```
void iupkey_open(void); [in C]
[There is no equivalent in Lua]
```

See Also

[K_ANY](#) callback, [KEY](#) attribute

IupDestroy

Destroys an interface element and all of its descendants.

Parameters/Return

```
void IupDestroy(Ihandle *element); [in C]
IupDestroy(element: iuplua_tag) [in IupLua3]
iup.Destroy(element: iuplua_tag) [in IupLua5]
or element:destroy() [in IupLua]
```

element: Identifier of the interface element to be destroyed.

Notes

This function deletes the names associated to the interface elements being destroyed. It does not free the memory of attribute values that was allocated by the application.

ATTENTION: The interface elements associated by means of attributes (such as menus in dialogs and images in buttons) are not destroyed. The application will be in charge of such task.

IupGetFocus

Verifies the interface element with keyboard focus, that is, the element that receives keyboard events.

Parameters/Return

```
Ihandle* IupGetFocus(void); [in C]
IupGetFocus() -> elem: iuplua_tag [in IupLua3]
iup.GetFocus() -> elem: iuplua_tag [in IupLua5]
```

This function returns the identifier of the interface element which at the moment is receiving keyboard events.

See Also

[IupSetFocus.](#)

IupSetFocus

Defines the interface element that will receive the keyboard focus, i.e., the element that will receive keyboard events.

Parameters/Return

```
Ihandle *IupSetFocus (Ihandle *element); [in C]
IupSetFocus(element: iuplua_tag) -> elem: iuplua_tag [in IupLua3]
iup.SetFocus(element: iuplua_tag) -> elem: iuplua_tag [in IupLua5]
```

element: identifier of the interface element that will receive the keyboard focus.

This function returns the identifier of the interface element that will receive the keyboard focus.

See Also

[IupGetFocus.](#)

IupHide

Hides an interface element. This function has the same effect as attributing value "NO" to the interface element's `VISIBLE` attribute.

Parameters/Return

```
int IupHide(Ihandle *element); [in C]
IupHide(element: iuplua_tag) -> (ret: number) [in IupLua3]
iup.Hide(element: iuplua_tag) -> (ret: number) [in IupLua5]
or element:hide() -> (ret: number) [in IupLua]
```

element: Identifier of the interface element.

This function returns `IUP_NOERROR` if the element was removed from the screen.

Note

Once a dialog is hidden, either by means of **IupHide** or by changing the `VISIBLE` attribute or by means of a callback returning `IUP_CLOSE`, the elements in this dialog are not destroyed, so that you can show them again. To destroy dialogs, the **IupDestroy** function must be called.

See Also

[IupShowXY](#), [IupShow](#), [IupPopup](#), [IupDestroy](#).

IupMap

Creates native interface objects corresponding to the given IUP interface elements.

Parameters/Return

```
int IupMap(Ihandle* element); [in C]
IupMap(element: iuplua-tag) -> ret: number [in IupLua3]
iup.Map(element: iuplua-tag) -> ret: number [in IupLua5]
```

element: Identifier of an interface element.

Notes

When **element** is of type *dialog*, this function creates the native interface element of a dialog and of each element it contains, but only if the **element** has not been mapped yet.

When **element** is not of type *dialog*, this function will only create the native interface element if the **element** is inside an already mapped dialog.

If **element** was already mapped, nothing happens.

If the WID attribute is NULL, it means the **element** was not already mapped.

This function is automatically called always before a dialog is made visible. This way, it only makes sense for the application to call it when the value of the WID attribute must be known before a dialog is made visible.

See Also

[IupShowXY](#), [IupShow](#), [IupPopup](#).

IupPopup

Shows a dialog or menu and restricts user interaction only to the specified element. This function will only return the control to the application after a callback returns **IUP_CLOSE** or when the popup dialog is hidden, for example using **IupHide**.

Parameters/Return

```
int IupPopup(Ihandle *element, int x, int y); [in C]
IupPopup(element: iuplua_tag, x, y: number) -> (ret: number) [in IupLua3]
iup.Popup(element: iuplua_tag, x, y: number) -> (ret: number) [in IupLua5]
or element:popup(x, y: number) -> (ret: number) [in IupLua]
```

element: Identifier of a dialog or a menu.

x: x coordinate of the left corner of the interface element. The following macros are valid:

- **IUP_LEFT**: Positions the element on the left corner of the screen

- `IUP_CENTER`: Centers the element on the screen
- `IUP_RIGHT`: Positions the element on the right corner of the screen
- `IUP_MOUSEPOS`: Positions the element on the mouse cursor

y: y coordinate of the upper part of the interface element. The following macros are valid:

- `IUP_TOP`: Positions the element on the top of the screen
- `IUP_CENTER`: Vertically centers the element on the screen
- `IUP_BOTTOM`: Positions the element on the base of the screen
- `IUP_MOUSEPOS`: Positions the element on the mouse cursor

This function returns `IUP_ERROR` if the element could not be created.

Notes

When a popup dialog is interacting with the user, another dialog can only be opened by means of the **IupPopup** function – never with **IupShow** or **IupShowXY**.

This function can be executed more than once for the same dialog. In fact, it works just like functions **IupShow** and **IupShowXY**, but it inhibits interaction with other dialogs. Therefore, it does not destroy the dialog's elements when it ends. To destroy the elements, function **IupDestroy** must be called.

See Also

[IupShowXY](#), [IupShow](#), [IupHide](#).

IupShow

Displays an interface element. This function has the same effect as setting value `IUP_YES` to the `IUP_VISIBLE` attribute of the interface element.

Parameters/Return

```
int IupShow(Ihandle *element); [in C]
IupShow(element: iuplua_tag) -> (ret: number) [in IupLua3]
iup.Show(element: iuplua_tag) -> (ret: number) [in IupLua5]
or element:show() -> (ret: number) [in IupLua]
```

element: identifier of the interface element.

This function returns `IUP_NOERROR` if the element was displayed.

Notes

An interface element is only visible if the dialog that contains it is also visible.

This function can be executed more than once for the same dialog. This will make the dialog be placed above all other dialogs in the application.

See Also

[IupShowXY](#), [IupHide](#), [IupPopup](#).

IupShowXY

Displays a dialog in a given position on the screen.

Parameters/Return

```
int IupShowXY(Ihandle *element, int x, int y); [in C]
IupShowXY(element: iuplua_tag, x, y: number) -> (ret: number) [in IupLua3]
iup.ShowXY(element: iuplua_tag, x, y: number) -> (ret: number) [in IupLua5]
or element:showxy(x, y: number) -> (ret: number) [in IupLua]
```

element: identifier of the dialog.

x: x coordinate of the dialog's left corner. The following macros are valid:

- IUP_LEFT: Positions the dialog on the left corner of the screen
- IUP_CENTER: Horizontally centralizes the dialog on the screen
- IUP_RIGHT: Positions the dialog on the right corner of the screen
- IUP_MOUSEPOS: Positions the dialog on the mouse position

y: y coordinate of the dialog's upper part. The following macros are valid:

- IUP_TOP: Positions the dialog on the top of the screen
- IUP_CENTER: Vertically centralizes the dialog on the screen
- IUP_BOTTOM: Positions the dialog on the base of the screen
- IUP_MOUSEPOS: Positions the dialog on the mouse position

This function returns IUP_NOERROR if the element was displayed.

Note

This function can be executed more than once for the same dialog. This will make the dialog be placed above all other dialogs in the application.

See Also

[IupShow](#), [IupHide](#), [IupPopup](#).

IupNextField

Shifts the focus to the next element in a dialog to which the specified element belongs. In does not depend on the element currently with the focus.

Parameters/Return

```
Ihandle* IupNextField(Ihandle* element); [in C]
IupNextField(element: iuplua_tag) -> (elem: iuplua_tag) [in IupLua3]
iup.NextField(element: iuplua_tag) -> (elem: iuplua_tag) [in IupLua5]
```

element: An element.

This function returns the element that received the focus.

See Also

[IupPreviousField](#).

IupPreviousField

Shifts the focus to the previous element in a dialog to which the specified element belongs. In does not depend on the element currently with the focus.

Parameters/Return

```
Ihandle* IupPreviousField(Ihandle* element); [in C]
IupPreviousField(element: iuplua_tag) -> (elem: iuplua_tag) [in IupLua3]
iup.PreviousField(element: iuplua_tag) -> (elem: iuplua_tag) [in IupLua5]
```

element: An element.

This function returns the element that received the focus.

See Also

[IupNextField](#).

IupDetach

Disassociates an interface element from the dialog that contains it.

Parameters/Return

```
void IupDetach(Ihandle *element); [in C]
IupDetach(element: iuplua_tag) [in IupLua3]
iup.Detach(element: iuplua_tag) [in IupLua5]
or element:detach() [in IupLua]
```

element: Identifier of the interface element to be detached.

Notes

This function does not destroy an interface element, but only removes the dialog that contains it so that it can be used in another dialog.

ATTENTION: At the moment, this function only works before the element is mapped by means of functions **IupMap**, **IupShow**, **IupShowXY** or **IupPopup**. The following example does not actually work!

Example

Inserts and removes a button box in/from a dialog.

In C

```

#include <stdlib.h> /* NULL */
#include "iup.h"

static Ihandle *pc; /* Identifier of the hbox that contains the button box */
static Ihandle *cb; /* Identifier of the button box */
static int visivel=1; /* =1 if the button box is on the screen, =0 otherwise */

/* Function that displays the button box */
static int exhibe (void)
{
    if (visivel == 0)
    {
        IupHide(pc);
        IupAppend(pc, cb);
        IupShow(pc);
        visivel = 1;
    }

    return IUP_DEFAULT;
}

/* Function that hides the button box */
static int retira (void)
{
    if (visivel == 1)
    {
        IupHide(pc);
        IupDetach(cb);
        IupShow(pc);
        visivel=0;
    }

    return IUP_DEFAULT;
}

void main (void)
{
    Ihandle *d; /* Identifier of the dialog */

    IupOpen();

    /* Creates button box */
    cb = IupFrame(IupVbox(
        IupButton(" Select ", "acao_select"),
        IupButton(" Line ", "acao_line"),
        IupButton("Poligono", "acao_poligono"),
        IupButton(" Circulo", "acao_circulo"),
        NULL));

    /* Creates a box containing a canvas and the buttons */
    pc = IupHbox(IupFrame(IupCanvas("acao_repaint")), cb, NULL);
    d = IupDialog (pc); /* criação do diálogo */
    IupSetAttribute (d, IUP_K_F1, "exibe_caixa"); /* Defines F1 to display the l
    IupSetAttribute (d, IUP_K_F2, "retira_caixa"); /* Defines F2 to hide the b

    /* Associates functions to the actions */
    IupSetFunction("exibe_caixa",(Icallback)exibe);
    IupSetFunction("retira_caixa",(Icallback)retira);

    IupShow(d); /* Shows the dialog */
    IupMainLoop(); /* Interacts with user */
    IupClose();
}

```


See Also

[IupAppend](#), [IupDestroy](#).

IupAppend

Inserts an interface element at the end of a list in hbox, vbox, zbox or menu.

Parameters/Return

```
Ihandle* IupAppend(Ihandle *box, Ihandle *element); [in C]
IupAppend(box, element: iuplua_tag) -> (box: iuplua_tag) [in IupLua3]
iup.Append(box, element: iuplua_tag) -> (box: iuplua_tag) [in IupLua5]
```

box: Identifier of an hbox, vbox, zbox or menu.

element: Identifier of the element to be inserted in the box.

This function returns **box** if the interface element was successfully inserted. Otherwise, NULL (nil in Lua) is returned.

Notes

This function must be used when the interface elements that will compose an hbox, vbox, zbox or menu are not known *a priori* (in the program's compilation stage).

If the box where the interface element is being inserted is visible, the IupAppend function does not update it automatically. For such, the box must be hidden (IupHide) and made visible (IupShow) again.

ATTENTION: Currently, this function only works before the element is mapped by means of functions **IupMap**, **IupShow**, **IupShowXY** or **IupPopup**.

See Also

[IupDetach](#), [IupHbox](#), [IupVbox](#), [IupZbox](#), [IupMenu](#).

IupGetNextChild

Returns the children of the given control based on his brother.

Parameters/Return

```
Ihandle *IupGetNextChild(Ihandle *parent, Ihandle *lastchild); [in C]
IupGetNextChild(parent, lastchild: iuplua_tag) -> ret: iuplua_tag [in IupLua3]
iup.GetNextChild(parent, lastchild: iuplua_tag) -> ret: iuplua_tag [in IupLua!]
```

parent: Identifier of an interface control.

lastchild: Identifier of the last interface control returned by the function.

Note

This function will return the children of the control in the exact same order in which they

were assigned.

Example

```
/* Lists all children of a IupVbox */

#include <stdio.h>
#include "iup.h"

int main()
{
    Ihandle *dialog, *bt, *lb, *vbox, *tmp = NULL;

    IupOpen();

    bt = IupButton("Button", "");
    lb = IupLabel("Label");

    vbox = IupVbox(bt, lb, NULL);

    dialog = IupDialog(vbox);
    IupShow(dialog);

    while(1)
    {
        tmp = IupGetNextChild(vbox, tmp);
        if(tmp)
            printf("vbox has a child of type %s\n", IupGetType(tmp));
        else
            break;
    }

    IupMainLoop();
    IupClose();

    return 0;
}
```

See Also

[IupGetBrother](#)

IupGetBrother

Returns the brother of a control or NULL if there is none.

Parameters/Return

```
Ihandle* IupGetBrother(Ihandle *control); [in C]
IupGetBrother(control: iuplua_tag) -> ret: iuplua_tag [in IupLua3]
iup.GetBrother(control: iuplua_tag) -> ret: iuplua_tag [in IupLua5]
```

control: Brother of interface control given.

See Also

[IupGetNextChild](#)

IupGetType

Verifies the name of the type of an interface element.

Parameters/Return

```
char* IupGetType(Ihandle* elem); [in C]
IupGetType(elem: iuplua_tag) -> (name: string) [in IupLua3]
iup.GetType(elem: iuplua_tag) -> (name: string) [in IupLua5]
```

elem: Identifier of the interface element.

This function returns the name of the type of an interface element.

Notes

The following names are predefined:

```
"unknown"
"color"
"image"
"button"
"canvas"
"dialog"
"fill"
"frame"
"hbox"
"item"
"separator"
"submenu"
"label"
"list"
"menu"
"radio"
"text"
"toggle"
"vbox"
"zbox"
"multiline"
"user"
```

IupSetHandle

Defines a name for an interface element.

Parameters/Return

```
Ihandle *IupSetHandle(char *name, Ihandle *element); [in C]
IupSetHandle(name: string, element: iuplua_tag) -> handle: iuplua_tag [in Iup]
iup.SetHandle(name: string, element: iuplua_tag) -> handle: iuplua_tag [in Iup]
```

name: name of the interface element.

element: identifier of the interface element.

This function returns the identifier of the interface element previously associated to the parameter **name**.

Note

Attention: To delete an element's name, use

```
IupSetHandle("my element name", NULL);
```

See Also

[IupGetHandle](#).

IupGetHandle

Retrieves the identifier of an interface element.

Parameters/Return

```
Ihandle *IupGetHandle(char *name); [in C]
IupGetHandle(name: string) -> handle: iuplua_tag [in IupLua3]
iup.GetHandle(name: string) -> handle: iuplua_tag [in IupLua5]
```

name: name of an interface element.

This function returns the identifier of the interface element.

Note

This function is used for integrating IUP and LED. To manipulate an interface element defined in LED, first capture its identifier using function **IupGetHandle**, passing the name of the interface element as parameter, then use this identifier on the calls to IUP functions – for example, a call to the function that verifies the value of an attribute, **IupGetAttribute**.

Attention: in Lua, IupGetHandle is not able to get the Ihandle of a IUP element created in C. To get an Ihandle created in C, use IupGetFromC{ "name" }.

See Also

[IupSetHandle](#).

IupGetName

Verifies the name of an interface element.

Parameters/Return

```
char* IupGetName(Ihandle* elem); [in C]
IupGetName(elem: iuplua_tag) -> (name: string) [in IupLua3]
iup.GetName(elem: iuplua_tag) -> (name: string) [in IupLua5]
```

elem: Identifier of the interface element.

This function returns the name of an interface element.

Lua Binding

This name is not associated with the Lua variable name; this was inherited from LED and is needed for some functions.

See Also

[IupSetHandle](#), [IupGetHandle](#), [IupGetAllNames](#).

IupGetAllNames

Verifies the names of all interface elements defined.

Parameters/Return

```
int IupGetAllNames(char *names[], int n); [in C]
IupGetAllNames(names: string table, n: number) -> (num: number) [in IupLua3]
iup.GetAllNames(names: string table, n: number) -> (num: number) [in IupLua5]
```

names: table receiving the names

n: maximum number of names the table can receive.

This function returns the number of names loaded to the table.

Lua Binding

This name is not associated to the name of the Lua variable – this was inherited from LED and is needed for some functions.

See Also

[IupSetHandle](#), [IupGetHandle](#), [IupGetName](#), [IupGetAllDialogs](#).

IupGetDialog

Verifies the identifier of a dialog to which an interface element belongs.

Parameters/Return

```
Ihandle* IupGetDialog(Ihandle *elem); [in C]
IupGetDialog(elem: iuplua_tag) -> (handle: iuplua_tag) [in IupLua3]
iup.GetDialog(elem: iuplua_tag) -> (handle: iuplua_tag) [in IupLua5]
```

elem: Identifier of an interface element.

This function returns the identifier of the dialog that contains that interface element.

IupGetAllDialogs

Verifies the names of all defined dialogs.

Parameters/Return

```
int IupGetAllDialogs(char *names[], int n); [in C]
IupGetAllDialogs(names: string table, n: number) -> (num: number) [in IupLua3]
iup.GetAllDialogs(names: string table, n: number) -> (num: number) [in IupLua5]
```

names: table receiving the names

n: maximum number of names the table can receive.

This function returns the number of names loaded to the table.

Lua Binding

This name is not associated to the name of the Lua variable – this was inherited from LED and is needed for some functions.

See Also

[IupSetHandle](#), [IupGetHandle](#), [IupGetName](#), [IupGetAllNames](#).

IupStoreAttribute

Defines an attribute for an interface element.

Parameters/Return

```
void IupStoreAttribute(Ihandle *element, char *a, char *v); [in C]
IupStoreAttribute(element: iulua_tag, attribute: string, value: string) [in IupLua3]
iup.StoreAttribute(element: iulua_tag, attribute: string, value: string) [in IupLua5]
```

element: identifier of the interface element.

a: name of the attribute.

v: value of the attribute. If it equals NULL (nil in IupLua), the attribute will be removed from the element.

Note

The value stored in the attribute is duplicated. Usually you will not use this function to store private attributes of the application.

See Also

[IupGetAttribute](#), [IupSetAttribute](#)

IupSetAttribute

Defines an attribute for an interface element.

Parameters/Return

```
void IupSetAttribute(Ihandle *element, char *a, char *v); [in C]
IupSetAttribute(element: iulua_tag, attribute: string, value: string) [in Iup]
iup.SetAttribute(element: iulua_tag, attribute: string, value: string) [in Iup]
```

element: Identifier of the interface element.

a: name of the attribute.

v: value of the attribute. If it equals NULL (nil in Lua), the attribute will be removed from the element.

Notes

The value stored in the attribute is not duplicated. Therefore, you can store your private attributes, such as a structure with data to be used in a callback.

When you want IUP to store an attribute by duplicating a string passed as a value, use function [IupStoreAttribute](#).

For further information on memory allocation by IupSetAttribute, see [IupGetAttribute's notes section](#).

Example 1

Defines a radio's initial value.

In C

```
Ihandle *portrait = IupToggle("Portrait" , "acao_portrait");
Ihandle *landscape = IupToggle("landscape" , "acao_landscape");
Ihandle *box = IupVbox(portrait, IupFill(),landscape, NULL);
Ihandle *modo = IupRadio(box);
IupSetHandle("landscape", landscape); /* associates a name to initialize the :
IupSetAttribute(modo, "VALUE", "landscape"); /* defines the radio's initial v
```

Example 2

Some usages:

In C

1. `IupSetAttribute(texto, "VALUE", "Olá!");`
2. `IupSetAttribute(indicador, "VALUE", "ON");`
3.

```
struct
{
    int x;
    int y;
} myData;

IupSetAttribute(texto, "myData", (char*)&myData);
```

See Also

[IupGetAttribute](#), [IupSetAttributes](#), [IupGetAttributes](#), [IupStoreAttribute](#)

IupSetfAttribute

Defines an attribute for an interface element.

Parameters/Return

```
void IupSetfAttribute(Ihandle *element, char *a, char *f, ...); [in C]
[There is no equivalent in Lua]
```

element: identifier of the interface element.

a: name of the attribute.

f: format that describes the attribute. It follows the same standard as the **printf** function in C .

. . .: values of the attribute.

Note

This function is very useful because we usually have integer values and want to pass them to IUP attributes, but this is done by means of a string. This way, we can commonly use **sprintf** to compose that string.

See Also

[IupGetAttribute](#), [IupSetAttribute](#), [IupSetAttributes](#),
[IupGetAttributes](#), [IupStoreAttribute](#)

IupGetAttribute

Verifies the name of an interface element attribute.

Parameters/Return

```
char *IupGetAttribute(Ihandle *element, char *a); [in C]
IupGetAttribute(element: iuplua_tag, a: string) -> value: string [in IupLua3]
iup.GetAttribute(element: iuplua_tag, a: string) -> value: string [in IupLua5]
```

element: Identifier of the interface element.

a: name of the attribute.

This function returns attribute's value. If the attribute does not exist, NULL (nil in IupLua) is returned.

Notes

This function's return value is not necessarily the same one used by the application to define the attribute's value. The subsequent call to the **IupGetAttribute** function may change the contents of the previously returned pointer, as this is an internal IUP buffer. The user is in charge of storing the value before calling any other IUP function.

The user has to understand that there is a difference between IUP attributes, such as VALUE or SIZE, and those stored for the user. The IUP attributes are often dynamically computed, stored in a temporary buffer and returned for the user to have access to the

values. In the case of attributes stored for the user, the pointer returned by **IupGetAttribute** will be the same as the stored pointer, allowing the contents to be changed.

The pointers of internal IUP attributes returned by **IupGetAttribute** must **never** be freed or changed.

In IupLua, only known internal pointer attributes are returned as user data, all other attributes are returned as strings. To access attribute data always as user data use **IupGetAttributeData** (Lua 3) and **iup.GetAttributeData** (Lua 5).

Example

See Also

[IupSetAttribute](#), [IupGetInt](#), [IupGetFloat](#), [IupSetAttributes](#), [IupGetHandle](#).

IupSetAttributes

Defines a set of attributes for an interface element. This function keeps a copy of the attributes' parameters.

Parameters/Return

```
Ihandle *IupSetAttributes(Ihandle *element, char *attributes); [in C]
IupSetAttributes(element: iulua_tag, attributes: string) -> elem: iulua_tag [:
iup.SetAttributes(element: iulua_tag, attributes: string) -> elem: iulua_tag
```

element: Identifier of the interface element.

attributes: in the form **v1=a1, v2=a2, ...** where **vi** is the name of an attribute and **ai** is its value.

This function returns **element** if all attributes were defined, or NULL (nil in Lua) otherwise.

Notes

It is worth noting that, in this function, the names of the attributes recognized by IUP cannot be defined with the prefix **IUP_**.

This function returns the same **Ihandle** it receives. This way, it is a lot easier to create dialogs in C. For example:

```
dialog = IupSetAttributes(
    IupDialog(
        IupVBox(
            IupSetAttributes(IupFill(), "SIZE = 5"),
            IupHBox(
                IupSetAttributes(IupFill(), "SIZE = 5"),
                canvas = IupSetAttributes(IupCanvas("repaind_cb"), "BORDER=NO, Ri
                IupSetAttributes(IupFill(), "SIZE = 5"),
                NULL),
            IupSetAttributes(IupFill(), "SIZE = 5"),
```

```

        NULL)),
        "TITLE = Teste")

```

Example

Creates a list with country names and defines Japan as the selected option.

In C

```

Ihandle *lista = IupList ("acao_lista");
IupSetAttributes(lista, "VALUE=3,1=Brazil,2=USA,3=Japan,4=France");

```

See Also

[IupGetAttribute](#), [IupSetAttribute](#), [IupGetAttributes](#),
[IupStoreAttribute](#)

IupGetAttributes

Verifies all attributes of a given element that are in the internal hash table. The known internal pointers are returned as integers.

Parameters/Return

```

char* IupGetAttributes (Ihandle *element); [in C]
IupGetAttributes(element: iulua_tag) -> (attributes: string) [in IupLua3]
iup.GetAttributes(element: iulua_tag) -> (attributes: string) [in IupLua5]

```

element: Identifier of the interface element.

attributes: in the form $v_1=a_1, v_2=a_2, \dots$ where v_i is the name of an attribute and a_i is its value.

This function returns all attributes defined for that element.

See Also

[IupGetAttribute](#), [IupSetAttribute](#), [IupSetAttributes](#),
[IupStoreAttribute](#)

IupGetFloat

Verifies the value of an interface element attribute and converts it to a float value.

Parameters/Return

```

float IupGetFloat(Ihandle *element, char *a) [in C]
[There is no equivalent in IupLua]

```

element: Identifier of the interface element.

a: name of the attribute.

This function returns a float corresponding to the attribute's value.

Note

The call to `IupGetFloat` cancels IUP's internal buffer. This means that after the call to `IupGetFloat`, the contents previously returned by function [IupGetAttribute](#) is no longer valid.

See Also

[IupGetAttribute](#), [IupGetInt](#).

IupGetInt

Verifies the value of an interface element attribute and converts it to int.

Parameters/Return

```
int IupGetInt(Ihandle *element, char *a); [in C]
[There is no equivalent in IupLua]
```

element: Identifier of the interface element.

a: name of the attribute.

This function returns the value of the interface element converted to int.

Notes

If the attribute value is "YES" / "NO" or "ON" / "OFF", the function returns 1 / 0, respectively.

The call to function `IupGetInt` invalidates IUP's internal buffer. This means that after a call to this function the contents previously returned by [IupGetAttribute](#) will no longer be valid.

See Also

[IupGetAttribute](#), [IupGetFloat](#).

IupStoreGlobal

Defines an attribute for the global environment.

Parameters/Return

```
void IupStoreGlobal(char *a, char *v); [in C]
IupStoreGlobal(attribute: string, value: string) [in IupLua3]
iup.StoreGlobal(attribute: string, value: string) [in IupLua5]
```

a: name of the attribute.

v: value of the attribute. If it equals NULL (nil in Lua), the attribute will be removed.

Note

The value stored in the attribute is duplicated.

See Also

[IupSetAttribute](#), [IupGetGlobal](#), [IupSetGlobal](#)

IupSetGlobal

Defines an attribute for the global environment.

Parameters/Return

```
void IupSetGlobal(char *a, char *v); [in C]
IupSetGlobal(attribute: string, value: string) [in IupLua3]
iup.SetGlobal(attribute: string, value: string) [in IupLua5]
```

a: name of the attribute.

v: value of the attribute. If it equals NULL (nil in IupLua), the attribute will be removed.

Notes

The value stored in the attribute is not duplicated. Therefore, you can store your private attributes, such as a structure of data to be used in a callback.

When you want IUP to store the attribute's value by duplicating the string, use function **IupStoreGlobal**.

See Also

[IupSetAttribute](#), [IupGetGlobal](#), [IupStoreGlobal](#)

IupGetGlobal

Verifies an attribute's value in the global environment.

Parameters/Return

```
char *IupGetGlobal(char *a); [in C]
IupGetGlobal(a: string) -> value: string [in IupLua3]
iup.GetGlobal(a: string) -> value: string [in IupLua5]
```

a: name of the attribute.

This function returns the attribute's value. If the attribute does not exist, NULL (nil in Lua) is returned.

Note

This function's return value is not necessarily the same one used by the application to define the attribute's value.

The subsequent call to the **IupGetGlobal** function may change the contents of the

previously returned pointer, as this is an internal IUP buffer. The user is in charge of storing the value before calling `IupGetGlobal` again. This pointer must not be freed either.

See Also

[`IupGetAttribute`](#), [`IupSetGlobal`](#)

IupGetActionName

Returns the name of the action being executed by the application.

Parameters/Return

```
char* IupGetActionName(void); [in C]
[There is no equivalent in IupLua]
```

Returns the name of the action.

Note

The programmer often defines an action with a given name, but when associating it to a function he/she might make a typo, or vice-versa. This kind of mistake is very common, but IUP cannot detect it automatically. The predefined **DEFAULT_ACTION** action combined with function **IupGetActionName** can help the programmer detect this problem. Simply define a default action and check which action name activated it.

See Also

[`DEFAULT_ACTION`](#)

IupGetFunction

Verifies the function associated to an action.

Parameters/Return

```
Icallback IupGetFunction (char *action); [in C]
[There is no equivalent in IupLua]
```

action: name of an action.

This function returns the path of the function associated to the action.

See Also

[`IupSetFunction`](#).

IupSetFunction

Associates a function to an action.

Parameters/Return

```
Icallback IupSetFunction (char *action, Icallback function); [in C]
[There is no equivalent in Lua]
```

action: name of an action.

function: path of a function.

This function returns the address of the previous function associated to the action.

See Also

[IupGetFunction](#), [DEFAULT_ACTION](#).

IupLoad

Compiles a LED specification.

Parameters/Return

```
char *IupLoad(char *name_file); [in C]
IupLoad(name_file: string) -> error: string [in IupLua3]
iup.Load(name_file: string) -> error: string [in IupLua5]
```

name_file: name of the file containing the LED specification.

This function returns NULL (nil in Lua) if the file was successfully compiled; otherwise it returns a pointer to a string containing the error message.

Note

Each time the function loads a LED file, the elements contained in it are created. Therefore, the same LED file cannot be loaded several times, otherwise the elements will also be created several times. The same applies for running Lua files several times.

Elements

Elements are basic interface components. They can have different forms:

Predefined Dialogs

Dialogs with a predefined functionality. They are used very frequently, and usually return useful values for the application.

Dialogs Predefined as Elements

The same idea, but now the dialog's attributes can be changed before they are shown on the screen. This provides the predefined dialogs a greater flexibility.

Composition Elements

Elements that do not have a visual representation, but they are essential for the functioning of the abstract-layout mechanism.

Elements

Basic elements with a visual representation. Together with composition elements, they constitute the dialog's layout.

Auxiliary Elements

Elements that complement the visual representation of the above elements.

Extra Lua Elements

Controls that help creating groups of elements in Lua.

CPI Controls

Extra controls. They can either be native controls or not. Most of them are not native, and they are implemented using the CD library to draw the elements.

Others

Extra controls which do not take part in IUP's distribution. They are distributed separately.

IupButton

Creates an interface element that is a button. When selected, this element activates a function in the application. Its visual presentation can contain a text or an image.

Parameters/Return

```
Ihandle* IupButton(char *title, char *action); [in C]
iupbutton{title = title: string} -> elem: iuplua_tag [in IupLua3]
iup.button{title = title: string} -> elem: iuplua_tag [in IupLua5]
button(title, action) [in LED]
```

title: Text to be shown to the user.

action: Name of the action generated when the button is selected.

This function returns the identifier of the created button, or NULL (`nil` in IupLua) if an error occurs.

Attributes

[BGCOLOR](#): Background color of the text.

[FGCOLOR](#): Text color.

[FONT](#): Character font of the text.

[IMAGE](#): Image of the non-pressed button. The button's title (attribute `TITLE`) is not shown when this attribute is defined.

[IMPRESS](#): Image of the pressed button.

[IMINACTIVE](#): Image of the button when the `ACTIVE` attribute equals "NO". If it is not defined but `IMAGE` is defined then for inactive buttons the non transparent colors will be replaced by a darker version of the background color creating the disabled effect.

[SIZE](#): Size of the button. Default: smallest size that allows viewing the text or image.

[TITLE](#): Text of the button.

Notes

Buttons with images or texts can not change its behavior after mapped. This is a creation attribute. But after creation the image can be changed for another image, and the text for another text.

Text and images are always centered.

Buttons are activated using Enter or Space keys.

When [IMPRESS](#) and [IMAGE](#) are defined together, IUP does not show the element's border to provide a 3D effect; the user has to define the border in the image itself.

Callbacks

[ACTION](#): Action generated when the button is selected.

[BUTTON_CB](#): Action generated when any mouse button is pressed or released.

Examples

See Also

[IupImage](#), [IupToggle](#).

IupCanvas

Creates an interface element that is a canvas - a working area for your application.

Parameters/Return

```
Ihandle* IupCanvas(char *action); [in C]
iupcanvas{} -> (elem: iuplua_tag) [in IupLua3]
iup.canvas{} -> (elem: iuplua_tag) [in IupLua5]
canvas(action) [in LED]
```

action: Name of the action generated when the canvas needs to be redrawn.

This function returns the identifier of the created canvas, or NULL if an error occurs.

Attributes

[BGCOLOR](#): Background color. In Windows the default value is "TRANSPARENT" which does not mean that the canvas is transparent, but it means that the application will draw its contents. This will avoid unnecessary redraws of the canvas. For better results use also the attribute CLIPCHILDREN=YES in the dialog.

[CURSOR](#): Canvas cursor.

[SIZE](#): Size of the canvas. Default: size of one character.

[SCROLLBAR](#): Associates a horizontal and/or vertical scrollbar to the canvas.

[DX](#): Size of the thumb in the horizontal scrollbar.

[DY](#): Size of the thumb in the vertical scrollbar.

[POSX](#): Position of the thumb in the horizontal scrollbar.

[POSY](#): Position of the thumb in the vertical scrollbar.

[XMAX](#): Maximum value of the horizontal scrollbar.

[XMIN](#): Minimum value of the horizontal scrollbar.

[YMIN](#): Minimum value of the vertical scrollbar.

[YMAX](#): Maximum value of the vertical scrollbar.

[BORDER](#): Shows a border around the canvas. It can only be changed before the element is mapped. This attribute does not work on Motif.

[EXPAND](#): The default value is YES, filling every possible space.

[CONID](#): Identifier of the canvas for GKS/puc.

DRAWSIZE: The size of the drawing area in pixels. In Motif this is identical to **RASTERSIZE**. In Windows it is obtained from the canvas client area since it may contain a border.

Callbacks

[ACTION](#): Action generated when the canvas needs to be redrawn. Also receives as parameters the scrollbar position:

```
int function(Ihandle *self, float x, float y); [in C]
elem:action(x, y: number) -> (ret: number) [in IupLua]
```

x: Thumb position in the horizontal scrollbar.

y: Thumb position in the vertical scrollbar.

This action is also generated right after the dialog is viewed by means of functions [IupShow](#), [IupShowXY](#) or [IupPopup](#).

[BUTTON_CB](#): Action generated when any mouse button is pressed or released.

[ENTERWINDOW_CB](#): Action generated when the mouse enters the canvas.

[LEAVEWINDOW_CB](#): Action generated when the mouse leaves the canvas.

[MOTION_CB](#): Action generated when the mouse is moved.

[KEYPRESS_CB](#): Action generated when a key is pressed or released.

[RESIZE_CB](#): Action generated when the canvas' size is changed.

[SCROLL_CB](#): Called when the scrollbar is manipulated.

[MAP_CB](#): Called right after the element is mapped.

[WOM_CB](#): Action generated when an audio device receives an event.

[WHEEL_CB](#): Action generated when the mouse wheel is rotated.

Note

Note that some keys might remove the focus from the canvas. To avoid this, return IGNORE in the [K_ANY](#) callback.

Examples

IupColor

Creates a color to be used in the color definition of interface elements.

Parameters/Return

```
Ihandle *IupColor(int r, int g, int b); [in C]
[There is no Lua equivalent]
color(r, g, b) [in LED]
```

r, **g**, **b**: intensity [0 . . . 255] of red, green and blue, respectively.

This function returns the identifier of the created color, or NULL (nil in IupLua) if an error occurs.

Attributes

BLUE: Blue intensity.

GREEN: Green intensity.

RED: Red intensity.

Integer values from 0 to 255 are accepted.

Notes

Though this function exists to help creating colors, the simplest way to modify a color is by directly defining values in a string, such as "0 0 255", as specified in attributes [FGCOLOR](#) and [BGCOLOR](#).

This function does not exist in IupLua, because the `r.. " " ..g.. " " ..b` concatenation can be used to obtain the same effect.

Examples

IupFrame

Creates a Frame interface element, which draws a frame with a title around an interface element.

Parameters/Return

```
Ihandle* IupFrame(Ihandle *element); [in C]
iupframe{element: iuplua_tag} -> (elem: iuplua_tag) [in IupLua3]
iup.frame{element: iuplua_tag} -> (elem: iuplua_tag) [in IupLua5]
frame(element) [in LED]
```

element: Identifier of an interface element which will receive the frame.

This function returns the identifier of the created frame, or NULL if an error occurs.

Attributes

[FGCOLOR](#): Text color.

[SIZE](#): Frame size.

[TITLE](#): Text the user will see at the top of the frame.

[MARGIN](#): Margin of the visible element.

Notes

Though this element has the attribute MARGIN, it does not have the attributes [ALIGNMENT](#) and [GAP](#), because it can contain only one element.

The [BGCOLOR](#) attribute has no effect.

Examples

IupImage

Creates an image to be shown on a label, button, toggle, or as a cursor.

Parameters/Return

```
Ihandle* IupImage(int width, int height, char *pixels); [in C]
iupimage{pixels: table of numbers, colors: table of colors} -> (elem: iuplua_
iup.image{pixels: table of numbers, colors: table of colors} -> (elem: iuplua_
image(width, height, b1, b2, ...) [in LED]
```

width: Image width in pixels.

height: Image height in pixels.

pixels: Vector containing the color of each pixel.

b1, **b2**, ...: Color index of the pixels.

This function returns the identifier of the created image, or NULL (`nil` in IupLua) if an error occurs.

Attributes

"0" Color in index 0.

"1" Color in index 1.

...

"i" Color in index i.

The indices can range from 0 to 255. The total number of colors is limited to 256 colors. Notice that in Lua the first index in the array is "1", the index "0" is ignored in IupLua. Be careful when setting colors, since they are attributes they follow the same storage rules for standard attributes.

The values are integer numbers from 0 to 255, one for each color in the RGB standard ("255 255 255"). If the value of a given index is "BGCOLOR", the color used will be the background color of the element on which the image will be inserted. The "BGCOLOR" must be defined with an index less than 16.

[HOTSPOT](#): The hotspot (x:y coordinates) used to define cursors.

[HEIGHT](#): Image height.

[WIDTH](#): Image width.

Notes

An image created with IupImage can be reused for different buttons and labels. But in Motif the BGCOLOR color index will be calculated only once when it is first used.

The images must be destroyed when they are no longer necessary, by means of the IupDestroy function. To destroy an image, it cannot be in use. Please observe the rules for creating cursor images: [CURSOR](#).

The pixels array is duplicated internally so you can discard it after calling IupImage.

If do not set a colors it is used a default color for the 16 first colors. The default color table is the same for Windows and Motif:

```

0 = 0, 0, 0 (black)
1 = 128, 0, 0 (dark red)
2 = 0, 128, 0 (dark green)
3 = 128, 128, 0 (dark yellow)
4 = 0, 0, 128 (dark blue)
5 = 128, 0, 128 (dark magenta)
6 = 0, 128, 128 (dark cyan)
7 = 192, 192, 192 (gray)
8 = 128, 128, 128 (dark gray)
9 = 255, 0, 0 (red)
10 = 0, 255, 0 (green)
11 = 255, 255, 0 (yellow)
12 = 0, 0, 255 (blue)
13 = 255, 0, 255 (magenta)
14 = 0, 255, 255 (cyan)
15 = 255, 255, 255 (white)

```

For images with more than 16 colors, all the color indices must be defined up to the maximum number of colors. For example, if the biggest image index is 100, then all the colors from $i=16$ up to $i=100$ must be defined even if some indices are not used. Note that to use more than 128 colors you must use an "unsigned char*" pointer and simply cast it to "char*" when calling the IupImage function.

The [EdPatt](#) and the [IMLAB](#) applications can load and save images in LED format. They allow operations such as importing GIF images and exporting them as IUP images.

EdPatt allows you to manually edit the images, and also have support for imagens in IupLua.

You can donwload several IUP images in LED format from [iup_images.zip](#). To view the images you can use the LED viewer application, see **IupView** in the applications included in the distribution.

Application icons are usually 32x32. Toolbar bitmaps are 24x24. Menu bitmaps and small icons are 16x16.

Examples

See Also

[IupLabel](#), [IupButton](#), [IupToggle](#).

IupLabel

Creates a label interface element, which displays a text or an image.

Parameters/Return

```

Ihandle* IupLabel(char *title); [in C]
iuplabel{title = title: string} -> (elem: iuplua_tag) [in IupLua3]
iup.label{title = title: string} -> (elem: iuplua_tag) [in IupLua5]
label(title) [in LED]

```

title: Text to be shown on the label.

This function returns the identifier of the created label, or NULL (nil in IupLua) if an error occurs.

Attributes

[BGCOLOR](#): Background color of the text.

[FGCOLOR](#): Text color.

[FONT](#): Character font of the text.

[IMAGE](#): Label image. When this attribute is defined, the text is not shown.

[SIZE](#): Label size.

[TITLE](#): Label's text.

ACTIVE: Activates or deactivates the label. The only difference between an active label and an inactive one is its visual feedback. Possible values:

"YES", "NO".

Default: "YES".

[ALIGNMENT](#): Label's alignment. Possible values:

"ALEFT", "ARIGHT", "ACENTER".

Default: "ALEFT".

SEPARATOR: Turns the label into a line separator. The EXPAND attribute is updated accordingly. Possible values:

"HORIZONTAL", "VERTICAL".

Notes

Labels with images, texts or line separator can not change its behavior after mapped. This is a creation attribute. But after creation the image can be changed for another image, and the text for another text.

Though this element can have the [IMAGE](#) attribute, it does not have attributes [IMINACTIVE](#) and [IMPRESS](#), because it does not interact with the user through the mouse or keyboard.

The '\n' character is accepted for line change, but the initial size of the element is computed for one line only. In this case, the EXPAND attribute must be "YES" so that the text can be properly visualized.

Examples

See Also

[IupImage](#), [IupButton](#).

IupList

Creates a `list` interface element, which is a list of two-state (on or off) items. An action is generated when an event changes the state of an item.

Parameters/Return

```
Ihandle* IupList(char *action); [in C]
iuplist{} -> (elem: iuplua_tag) [in IupLua3]
iup.list{} -> (elem: iuplua_tag) [in IupLua5]
list(action) [in LED]
```

action: String with the name of the action generated when the state of an item is changed.

This function returns the identifier of the created list, or NULL (`nil` in IupLua) if an error occurs.

Attributes

"1": First item in the list.

"2": Second item in the list.

"3": Third item in the list.

...

"n": nth item in the list.

The values can be any text. Default: NULL. The first element with a NULL is considered the end of the list. The string containing the item's number does not need to be static, because IUP duplicates it internally, but the contents of each element in the list needs to be either static or stored in IUP by means of the [IupStoreAttribute](#) function.

[DROPDOWN](#): Changes the appearance of the list for the user: only the selected item is shown beside a button with the image of an arrow pointing down. Creation-only attribute. Can be "YES" or "NO". Default "NO".

EDITBOX: Adds an edit box to the list. Creation-only attribute. Can be "YES" or "NO". Default "NO".

[VISIBLE_ITEMS](#): Number of items that appear when a DROPDOWN list is activated.

[MULTIPLE](#): Allows selecting several items simultaneously (multiple list).

[SIZE](#): Size of the list. Default: smallest size that allows viewing the list.

[VALUE](#):

List with edit box: Text entered by the user.

Simple list: Integer number representing the selected element in the list (begins at 1). It can be zero if there is no selected item.

Multiple list: Sequence of '+' and '-' symbols indicating the state of each item. When setting this value, the user must provide the same amount of '+'

and '-' symbols as the amount of items in the list, otherwise the specified items will be deselected.

[APPEND](#): Inserts a text at the end of the current text. Valid only when EDITBOX=YES.

[INSERT](#): Inserts a text in the caret's position. Valid only when EDITBOX=YES.

[NC](#): Maximum number of characters allowed. Valid only when EDITBOX=YES.

[CARET](#): Position of the insertion point. Valid only when EDITBOX=YES.

[READONLY](#): Allows the user only to read the contents, without changing it. Possible values: YES, NO (default). Valid only when EDITBOX=YES.

[SELECTION](#): Selection interval. Valid only when EDITBOX=YES.

[SELECTEDTEXT](#): Selection text. Valid only when EDITBOX=YES.

SHOWDROPDOWN: Opens a dropdown list. Windows Only.

Callbacks

[ACTION](#): Action generated when the state of an item in the list is changed. Also provides information on the changed item:

```
int function (Ihandle *self, char *t, int i, int v); [in C]
elem:action(t: string, i, v: string) -> (ret: number) [in IupLua]
```

t: Text of the changed item.

i: Number of the changed item.

v: Equal to 1 if the option was selected or to 0 if the option was deselected.

EDIT_CB: Action generated when the text in the text box is manually changed by the user. Valid only when EDITBOX=YES.

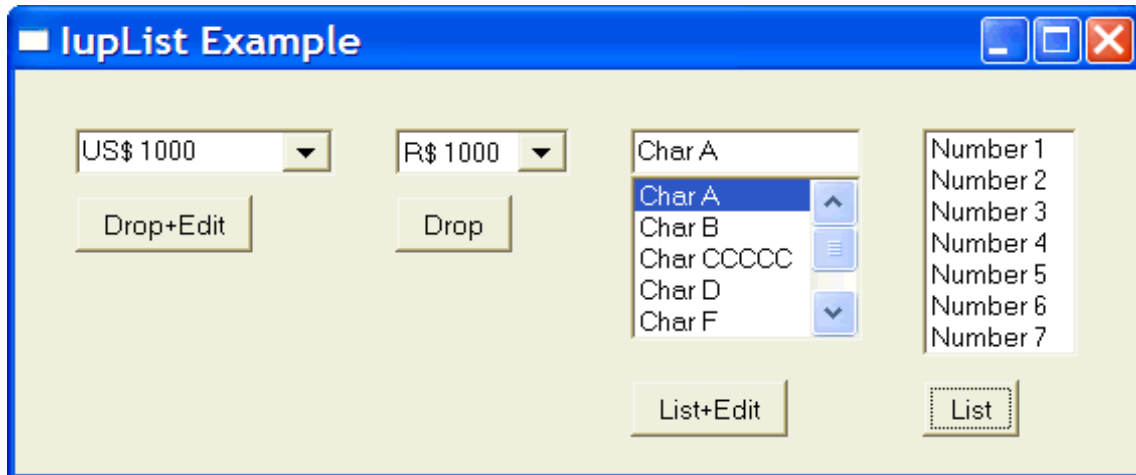
```
int function(Ihandle *self, int c, char *after); [in C]
elem:action(c: number, after: string) -> (ret: number) [in IupLua]
```

text: Represents the new text value. This is the same callback definition as for the [IupText](#).

Notes

Text is always left aligned.

Examples



See Also

[IupListDialog](#), [IupText](#)

IupMultiLine

Creates an editable field with one or more lines.

Parameters/Return

```
Ihandle* IupMultiLine(char *action); [in C]
iupmultiline{} -> (elem: iuplua_tag) [in IupLua3]
iup.multiline{} -> (elem: iuplua_tag) [in IupLua5]
multiline(action) [in LED]
```

action: name of the action generated when the user types something.

This function returns the identifier of the created multiline, or NULL if an error occurs.

Attributes

[APPEND](#): Inserts a text at the end of the multiline.

[INSERT](#): Inserts a text in the caret's position.

[BORDER](#): Shows a frame around the multiline.

[CARET](#): Position of the insertion point in the multiline.

[READONLY](#): Allows the user only to read the contents, without changing it. Possible values: "YES", "NO" (default).

[SELECTION](#): Selection interval.

[SELECTEDTEXT](#): Selection's text.

[NC](#): Maximum number of characters.

[SIZE](#): Multiline size. Default: 5 characters width and 1 character height.

[VALUE](#): Text typed by the user. The '\n' character indicates line change.

Default: NULL.

TABSIZE (Windows Only)

Controls the number of characters for a tab stop.

Callbacks

[ACTION](#): Action generated when a keyboard event occurs. The callback also receives the typed key.

```
int function(Ihandle *self, int c, char* after); [in C]
elem:action(c: number, after: string) -> (ret: number) [in IupLua]
```

c: Identifier of the typed key. Please refer to the [Keyboard Codes](#) table for a list of possible values.

after: Represents the new text value if the key is validated (i.e. the callback returns IUP_DEFAULT).

If the function returns IUP_IGNORE, the system will ignore the typed character. If the function returns the code of any other key, IUP will treat this new key instead of the one typed by the user.

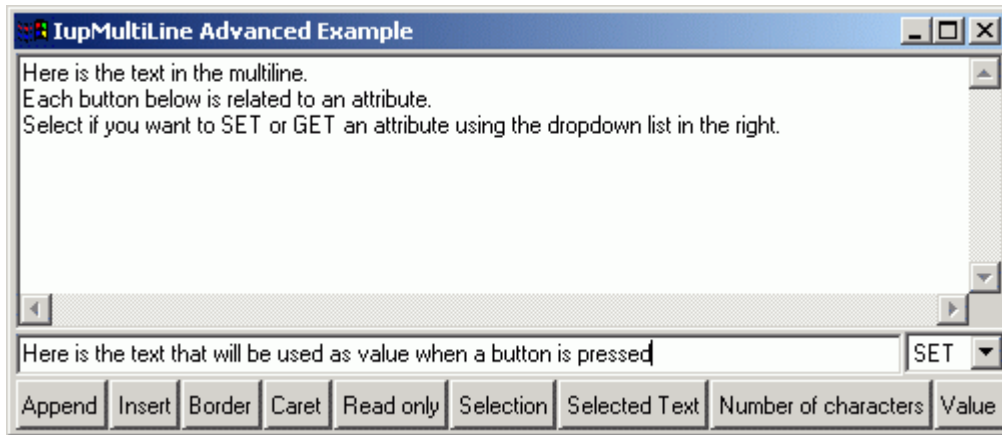
Notes

Text is always left aligned.

iupmultiline has a limitation of about 64,000 characters.

Since all the keys are processed to change focus to the next element press <Ctrl>+<Tab>. The "DEFAULTENTER" button will not be processed, but the "DEFAULTESC" will.

Examples



IupText

Creates an editable field with one line.

Parameters/Return

```
Ihandle* IupText(char *action); [in C]
iupText{} -> (elem: iuplua_tag) [in IupLua3]
iup.text{} -> (elem: iuplua_tag) [in IupLua5]
text(action) [in LED]
```

action: name of the action generated when the user types something.

This function returns the identifier of the created text, or NULL if an error occurs.

Attributes

[APPEND](#): Inserts a text at the end of the current text.

[INSERT](#): Inserts a text in the caret's position.

[BORDER](#): Shows a border around the text.

[NC](#): Maximum number of characters allowed.

[CARET](#): Position of the insertion point.

[READONLY](#): Allows the user only to read the contents, without changing it. Possible values: "YES", "NO" (default).

[SELECTION](#): Selection interval.

[SELECTEDTEXT](#): Selection text.

[SIZE](#): Text size. Default: 5 characters width and 1 character height.

[VALUE](#): Text entered by the user. If the element is already mapped, the string is directly copied to the native control (see [IupMap](#)).

The value can be any text, including '\n' characters indicating line change.
Default: NULL when the element is not yet mapped; " " if it is.

Callbacks

ACTION: Action generated when a keyboard event occurs. The callback also receives the typed key.

```
int function(Ihandle *self, int c, char *after); [in C]
elem:action(c: number, after: string) -> (ret: number) [in IupLua]
```

c: Identifier of the typed key. Please refer to the [Keyboard Codes](#) table for a list of possible values.

after: Represents the new text value in case the key is validated (i.e. the callback returns IUP_DEFAULT).

If the function returns IUP_IGNORE, the system will ignore the typed character. If the function returns the code of any other key, IUP will treat this new key instead of the one typed by the user.

Notes

Text is always left aligned.

On the Windows driver, the `action` callback is not called for the function keys (K_F???).

The [IupMask](#) control can be used to create a mask and filter the text entered by the user.

Examples

See Also

[IupMultiLine](#)

IupTimer

Creates a timer which periodically invokes a callback when the time is up.

Parameters/Return

```
Ihandle* IupTimer(); [in C]
iuptimer() -> (elem: iuplua_tag) [in IupLua3]
iup.timer() -> (elem: iuplua_tag) [in IupLua5]
timer() [in LED]
```

The function returns the identifier of the created handle, or NULL if an error occurs.

Attributes

TIME: The time interval in milliseconds.

RUN: Starts the timer. Possible values: "YES" or "NO".

Callbacks

ACTION_CB: Called when the time is up.

```
int function(Ihandle *self); [in C]
elem:action() -> (ret: number) [in IupLua]
```

self: Timer handle.

Examples

IupToggle

Creates the `toggle` interface element. It is a two-state (on/off) button that, when selected, generates an action that activates a function in the associated application. Its visual representation can contain a text or an image.

Parameters/Return

```
Ihandle* IupToggle(char *title, char *action); [in C]
iuptoggle{title = title: string} -> (elem: iuplua_tag) [in IupLua3]
iup.toggle{title = title: string} -> (elem: iuplua_tag) [in IupLua5]
toggle(title, action) [in LED]
```

title: Text to be shown on the toggle.

action: name of the action generated when the toggle is selected.

This function returns the identifier of the created toggle, or `NULL` if an error occurs.

Attributes

[BGCOLOR](#): Background color of the text shown on the toggle.

[FGCOLOR](#): Color of the text shown on the toggle.

[FONT](#): Character font of the text shown on the toggle.

[IMAGE](#): Toggle image. When the `IMAGE` attribute is defined, the `TITLE` is not shown. This makes the toggle look just like a button with an image, but its behavior remains the same.

[IMPRESS](#): Image of the pressed toggle.

[IMINACTIVE](#): Image of the inactive toggle. If it is not defined but `IMAGE` is defined then for inactive toggles the non transparent colors will be replaced by a darker version of the background color creating the disabled effect.

[VALUE](#): Toggle's state. Values can be "ON" or "OFF". Default: "OFF".

[SIZE](#): Toggle size.

[TITLE](#): Toggle's text.

SELECTCOLOR: (Motif Only) Color of a selected toggle.

Callbacks

[ACTION](#): Action generated when the toggle's state (on/off) changes. The callback also receives the toggle's state.

```
int function(Ihandle *self, int v); [in C]
elem:action(v: number) -> (ret: number) [in IupLua]
```

v: 1 if the toggle's state was shifted to on; 0 if it was shifted to off.

Notes

Toggles with images or texts can not change its behavior after mapped. This is a creation attribute. But after creation the image can be changed for another image, and the text for another text.

Text is left aligned and image is centered.

Toggles are activated using the Space key.

In Windows, the BGCOLOR attribute is ignored when an IMAGE is specified.

In Windows, for toggles inside a radio the ACTION callback may also be called when a not selected toggle receive the focus.

Examples

See Also

[IupImage](#), [IupButton](#), [IupLabel](#).

IupUser

Creates a user element in IUP, which is not associated to any interface element. It is used to map an external element to a IUP element. Its use is restricted and is usually done by CPI elements.

Parameters/Return

```
Ihandle* IupUser(void); [in C]
[There is no equivalent in IupLua]
[There is no equivalent in LED]
```

This function returns the identifier of the created element, or NULL if an error occurs.

IupDialog

Creates a dialog element. It manages user interaction with the interface elements. For any

interface element to be shown, it must be encapsulated in a dialog.

Parameters/Return

```
Ihandle* IupDialog(Ihandle *element); [in C]
iupdialog{element: iuplua_tag} -> (elem: iuplua_tag) [in IupLua3]
iup.dialog{element: iuplua_tag} -> (elem: iuplua_tag) [in IupLua5]
dialog(element) [in LED]
```

element: Identifier of an interface element.

This function returns the identifier of the created dialog, or NULL if an error occurs.

Attributes

[CURSOR](#): Defines a cursor for the dialog.

[ICON](#): Dialog's icon.

[MAXBOX](#): Requires a maximize button from the window manager. Creation-only attribute.

[MENU](#): Associates a menu to the dialog.

[MENUBOX](#): Requires a menu box from the window manager. Creation-only attribute.

[MINBOX](#): Requires a minimize button from the window manager. Creation-only attribute.

[RESIZE](#): Allows interactively changing the dialog's size. Creation-only attribute.

[SIZE](#): Dialog's size. Differently from other interface elements, the following values can be defined for width and height:

- "FULL": Defines the dialog's width (or height) equal to the screen's width (or height)
- "HALF": Defines the dialog's width (or height) equal to half the screen's width (or height)
- "THIRD": Defines the dialog's width (or height) equal to 1/3 the screen's width (or height)
- "QUARTER": Defines the dialog's width (or height) equal to 1/4 of the screen's width (or height)
- "EIGHTH": Defines the dialog's width (or height) equal to 1/8 of the screen's width (or height)

Default: the smallest size that allows viewing the dialog.

The dialog's size has precedence over the smallest size required by its children (either if it was specified in its creation or in run-time). Attributing a NULL value to SIZE or RASTERSIZE (in C) in a dialog will recompute its size according to its children.

[TITLE](#): Dialog's title. On Motif, if it is not defined, the dialog will not be properly displayed.

STARTFOCUS: Name of the dialog element that must receive the focus right after the dialog is opened.

DEFAULTENTER: Name of the button activated when Enter is hit.

DEFAULTESC: Name of the button activated when Esc is hit.

X: Dialog's horizontal position on the screen, in pixels.

Y: Dialog's vertical position on the screen, in pixels.

SHRINK: Allows changing the elements' distribution when the dialog is smaller than the minimum size.

PARENTDIALOG: Makes the dialog be treated as a child of the specified dialog.

FULLSCREEN

Makes the dialog occupy the whole screen. All dialog details, such as border, maximize button, etc, are removed. Possible values: YES, NO. Must be set before mapping to the native system. In Motif you may have to click in the dialog to set its focus. Use IupPopup for better results.

WIN_SAVEBITS (Windows Only)

This attribute is only consulted when the dialog is mapped. When this attribute is true (YES), the dialog stores the original image of the desktop region it occupies (if Windows has enough memory to store the image). In this case, when the dialog is closed or moved, a redrawing event is not generated for the windows that were shadowed by it. Its default value is YES.

TOPMOST (Windows Only)

This attribute puts the dialog always in front of all other dialogs in all applications. Default: NO.

TOOLBOX (Windows Only)

This attribute makes the dialog look like a toolbar. It is only valid if the PARENTDIALOG attribute is also defined. Default: NO.

CLIPCHILDREN (Windows Only)

Modifies the way the dialog and its children are redrawn.

When option YES is selected, the area occupied by the children in the dialog is not redrawn, thus preventing the matrix and the canvas from blinking when a resize is made. Usually this brings better performance, but in some cases it may bring a performance reduction, as every time the dialog needs to be redrawn all children are redrawn as well – including IupFrame. For the attribute to work efficiently, the canvas cannot be inside a IupFrame. Default: NO.

BRINGFRONT (Windows Only)

This attribute makes the dialog the foreground window. Use "YES" to activate it. Useful for multithreaded applications.

NATIVEPARENT (Windows Only)

Makes any window created in the system (even from outside IUP) able to be parent of a IUP dialog. The value provided should be a valid window handle (HWND.)

PLACEMENT (Windows Only)

Changes how the dialog will be show. Values: "MAXIMIZED", "MINIMIZED" and "NORMAL". After IupShow the attribute is set to "NORMAL" if it was different. "NORMAL" is equivalent of not defining the attribute.

HELPBUTTON (Windows Only)

Inserts a help button in the same place of the maximize button. It can only be used for dialogs without the minimize and maximize buttons, and with the menu box. For the next interaction of the user with a control in the dialog, the callback [HELP_CB](#) will be called instead of the control defined ACTION callback. Possible values: YES, NO. Default: NO.

[TRAY](#) (Windows Only): When set to "YES", displays an icon on the system tray.

[TRAYICON](#) (Windows Only): System tray icon

[TRAYTIP](#) (Windows Only): Tray icon's tooltip text

HIDETASKBAR (Windows Only)

When set to "YES", hides the dialog from the task bar. Must be used with TRAYICON attribute.

Callbacks

[SHOW_CB](#): Called right after the dialog is opened, minimized or restored from a minimization.

[MAP_CB](#): Called right after the element is mapped.

[CLOSE_CB](#): Called right before the dialog is closed.

TRAYCLICK_CB: Called right after the mouse button is pressed or released over the tray icon.

```
int function(Ihandle *n, int but, int pressed, int dclick); [in C]
elem:trayclick(but, pressed, dclick: number) -> (ret: number) [in IupLua]
elem:trayclick_cb(but, pressed, dclick: number) -> (ret: number) [in IupC]
```

but: identifies the activated mouse button.
pressed: indicates the state of the button.
click: indicates a double click.

Returning CLOSE closes the dialog.

Notes

Except for the menu, all other elements must be inside a dialog to interact with the user. Therefore, an interface element will only be visible when its [VISIBLE](#) attribute and that of the dialog are "YES".

A menu that is not associated to a dialog can interact with the user by means of the IupPopup function.

Values attributed to the SIZE attribute of a dialog are **always** accepted, regardless of the minimum size required by its children. For a dialog to have the minimum necessary size to fit all elements contained in it, simply define NULL (in C) to SIZE. In the case of partial dimensions, a specified dimension is **always** used, while a non-defined dimension uses the smallest necessary size for the elements in the corresponding direction.

In Motif the decorations MENUBOX, MINBOX, MAXBOX, RESIZE and BORDER will work only if the running Window Manager supports the Motif WM hints.

Examples

IupFileDlg

Creates the File Dialog element. It is a predefined dialog for selecting files or a directory.

Parameters/Return

```
Ihandle* IupFileDlg (void); [in C]
iupfiledlg() -> (elem: iuplua_tag) [in IupLua3]
iup.filedlg() -> (elem: iuplua_tag) [in IupLua5]
filedlg() [in LED]
```

This function returns the identifier of the created dialog, or NULL if an error occurs.

Attributes

DIALOGTYPE: Type of dialog (Open, Save or GetDirectory)

Can have values "OPEN", "SAVE" or "DIR". Default: "OPEN".

[TITLE](#): Dialog's title.

[FILE](#): Name of the file initially shown in the "File Name" field in the dialog.

[FILTER](#): File filter.

FILTERINFO: Filter's description.

EXTFILTER: (Windows Only) Defines several file filters. It has priority over **FILTER** and **FILTERINFO**. Must be a text with the format "Description1|filter1|Description2|filter2;filter3". The amount of descriptions and of filters is unlimited.

Example: "Text files|*.txt;*.doc|Image files|*.gif;*.jpg;*.bmp".

DIRECTORY: Initial directory.

PARENTDIALOG: Makes the dialog be treated as a child of the specified dialog.

ALLOWNEW: Indicates if non-existent file names are accepted. If equals "NO" and the user specifies a non-existing file, an alert dialog is shown.

NOCHANGEDIR: Indicates if the initial working directory must be restored after the user navigation.

FILEEXIST: Indicates if the file defined by the **FILE** attribute exists or not.

STATUS: Indicates the status of the selection made:

"1": New file.

"0": Normal, existing file.

"-1": Operation cancelled.

VALUE: Name of the selected file, or NULL if no file was selected.

NOOVERWRITEPROMPT do not prompt to overwrite an existant file when in "SAVE" dialog. Default is "NO", i.e. prompt before overwrite.

MULTIPLEFILES (Windows Only)

When "YES", this attribute allows the user of `IupFileDlg` in `fileopen` mode to select multiple files.

The value returned by **VALUE** is to be changed the following way: the directory and the files are passed separately, in this order. The character used for separating the directory and the files is '|'. The file list ends with character '|' followed by NULL.

When the user selects just one file, the directory and the file are not separated by '|'.

Ex.:

"C:\users\sab|a.txt|b.txt|" or

"C:\users\sab\a.txt" (only one file is selected)

The maximum size allowed by `IupFileDlg` for file return is 2000 characters. If the size exceeds 2000 characters, **VALUE** will return NULL.

FILTERUSED (Windows Only)

In a `IupFileDlg`, this attribute allows the user to select which `EXTFILTER` to use. It is also possible to retrieve the selection made by the user. Value: a string containing the number of the filter.

SHOWPREVIEW (Windows Only)

A preview area is show inside the File Dialog. Can have values "YES" or "NO". Default: "NO". When this attribute is set you must use the "**FILE_CB**" callback to retrieve the file name and the necessary attributes to paint the preview area. You must link with the "iup.rc" resource file so the preview area can be enabled.

PREVIEWDC, PREVIEWWIDTH and PREVIEWHEIGHT (Windows Only)

Read only attributes that are updated during the "PAINT" status of the "FILE_CB" callback. Return the Device Context (HDC), the width and the height of the client rectangle for the preview area.

Callbacks

FILE_CB: (Windows Only) Action generated when a file is selected.

```
int function(Ihandle *self, const char* file_name, const char* status);
elem:file(file_name, status: string) -> (ret: number) [in IupLua3]
elem:file_cb(file_name, status: string) -> (ret: number) [in IupLua5]
```

self: identifier of the element that activated the function.

file_name: name of the file selected.

status: describes the current action. Can be:

```
"INIT" - when the dialog has started. file_name is NULL.
"FINISH" - when the dialog is closed. file_name is NULL.
"SELECT" - a file has been selected.
"OK" - the user pressed the OK button. If the callback returns IGNORE
"PAINT" - the preview area must be repainted. This is used only when
```

Notes

In the Windows driver, the `FileDialog` is not altered by `IupSetLanguage`.

To show the dialog, use function **IupPopup**. In Lua, use the **popup** function.

Example in C

```
filedlg = IupFileDlg();
IupPopup(filedlg, IUP_ANYWHERE, IUP_ANYWHERE);
```

Example in IupLua 3

```
filedlg = iupfiledlg{}
filedlg:popup(IUP_ANYWHERE, IUP_ANYWHERE)
```

[Examples](#)

See Also

[IupMessage](#), [IupScanf](#), [IupListDialog](#), [IupAlarm](#), [IupGetFile](#), [IupPopup](#)

IupAlarm

Shows a dialog containing a message with up to three buttons, and waits for the user to press one.

Parameters/Return

```
int IupAlarm(char *t, char *m, char *b1, char *b2, char *b3); [in C]
IupAlarm(t, m, b1, b2, b3: string) -> (button: number) [in IupLua3]
iup.Alarm(t, m, b1, b2, b3: string) -> (button: number) [in IupLua5]
```

t: Dialog's title

m: Message

b1: Text of the first button

b2: Text of the second button (optional)

b3: Text of the third button (optional)

This function returns the number (1, 2, 3) of the button selected by the user, or 0 (nil in IupLua) if the dialog could not be opened.

Notes

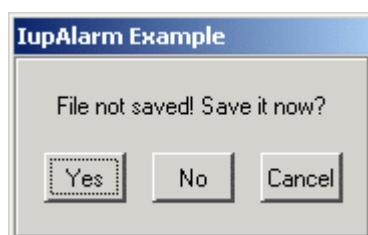
This function shows a dialog centralized on the screen, with the message and the buttons. The '\n' character can be added to the message to indicate line change.

A button is not shown if its parameter is NULL. This is valid only for **b2** and **b3**.

Button 1 is set as the "DEFAULTENTER" and "DEFAULTESC". If Button 2 exists it is set as the "DEFAULTESC". If Button 3 exists it is set as the "DEFAULTESC".

The dialog uses a global attribute called "PARENTDIALOG" as the parent dialog if it is defined.

[Examples](#)



See Also

[IupMessage](#), [IupScanf](#), [IupListDialog](#), [IupGetFile](#).

IupGetFile

Shows the dialog of the native interface system, which captures a filename.

Parameters/Return

```
int IupGetFile(char *file); [in C]
IupGetFile(file: string) -> (file: string, error: number) [in IupLua3]
iup.GetFile(file: string) -> (file: string, error: number) [in IupLua5]
```

file: This parameter is used as an input value to define the default filter and directory. Example: `"../docs/*.txt"`. As an output value, it is used to contain the filename entered by the user.

error: The function returns an error code, whose values can be:

- 1: The name defined by the user is that of a new file
- 0: The name defined by the user is that of an already existent file
- 1: The operation was cancelled by the user

Note

The **IupGetFile** function does not allocate memory space to store the complete filename entered by the user. Therefore, the file parameter must be large enough to contain the directory and file names.

The dialog uses a global attribute called "PARENTDIALOG" as the parent dialog if it is defined.

Examples

See Also

[IupMessage](#), [IupScanf](#), [IupListDialog](#), [IupAlarm](#),
[IupSetLanguage](#).

IupGetText

Shows the dialog to edit a multiline text.

Parameters/Return

```
int IupGetText(char* title, char *text); [in C]
IupGetText(title, text: string) -> (text: string) [in IupLua3]
iup.GetText(title, text: string) -> (text: string) [in IupLua5]
```

text: It contains the initial value of the text and the returned text. It must have room for the edited string.

The function returns a non zero value if successfull. In Lua if an error occurred returns nil.

Notes

The dialog uses a global attribute called "PARENTDIALOG" as the parent dialog if it is defined.

See Also

[IupMessage](#), [IupScanf](#), [IupListDialog](#), [IupAlarm](#),
[IupSetLanguage](#).

IupListDialog

This dialog shows a simple or multiple list, and waits for user feedback.

Parameters/Return

```
int IupListDialog(int type, char *title, int size, char *list[], int option, ...
IupListDialog(type: number, title: string, size: number, list: table of strings, ...
iup.ListDialog(type: number, title: string, size: number, list: table of strings, ...)
```

type: =1 simple selection; =2 multiple selection

title: Text for the dialog's title

size: Number of options

list: List of options

option: Initial option, starting at 1 (note that this index is different from the return value, kept for compability reasons)

max_col: Maximum number of columns in the list

max_lin: Maximum number of lines in the list

mark: Flag vector, used only when type=2

When type=1, the function returns the number of the selected option (the first option is 0), or -1 if the user cancels the operation.

When type=2, the function returns -1 when the user cancels the operation. If the user does not cancel the operation the function returns a non zero value and the mark parameter will have value 1 for the options selected by the user and value 0 for non-selected options.

Comments

In IupLua, the return value depends on used option. In case type is 1 (simple selection), the return value is a 0-based number of the selected option. If the type is 2 (multiple selection), the return type is a table with the marked options.

The dialog uses a global attribute called "PARENTDIALOG" as the parent dialog if it is defined.

Examples

See Also

[IupMessage](#), [IupScanf](#), [IupGetFile](#), [IupAlarm](#)

IupMessage

Shows a dialog containing a message and a button, and waits for the user to click the button.

Parameters/Return

```
void IupMessage(char *t, char *m); [in C]
IupMessage(t: string, m: string) [in IupLua3]
iup.Message(t: string, m: string) [in IupLua5]
```

t: Dialog's title

m: Message

Note

The **IupMessage** function shows a dialog centralized on the screen, showing the message and the “OK” button. The ‘\n’ character can be added to the message to indicate line change.

In C there is an utility function to help build the message string, it accepts the same format as the C **sprintf**:

```
void IupMessagef(char *t, char *f, ...); [in C]
```

The dialog uses a global attribute called "PARENTDIALOG" as the parent dialog if it is defined.

Examples

See Also

[IupGetFile](#), [IupScanf](#), [IupListDialog](#), [IupAlarm](#)

IupScanf

Shows a dialog for capturing values with a format similar to the `scanf` function in the C stdio library.

Parameters/Return

```
int IupScanf(char *fmt, ...); [in C]
IupScanf(fmt: string, ...) -> (n: number, ...) [in IupLua3]
iup.Scanf(fmt: string, ...) -> (n: number, ...) [in IupLua5]
```

fmt: Reading format

...: List of variables

This function returns the number of successfully read fields, or -1 when the user has canceled the operation.

In Lua, the values are returned by the function in the same order they were passed.

Notes

The **fmt** format must include a title and the descriptions of the variable fields to be read, using the following syntax:

- **First line:** Window title followed by '\n'
- **Following lines:** Must be specified for each variable to be read, in the following format:

"**text**%**t**.**v**%**f**\n", where:

text is a descriptive text, to be placed to the left of the entry field in a label.
t is the maximum number of characters allowed
v is the maximum number of visible characters in the entry field
f is the type (char, float, etc.), in the C format for I/O services

All the fields use a text box for input. If you need better control of what characters the user enters, you should use [IupGetParam](#). This other dialog also has many other resources not available in **IupScanf**.

The dialog uses a global attribute called "PARENTDIALOG" as the parent dialog if it is defined.

Examples

Captures an integer number, a floating-point value and a character string.

See Also

[IupGetFile](#), [IupMessage](#), [IupListDialog](#), [IupAlarm](#), [IupGetParam](#)

IupFill

Creates a **Fill** interface element, which dynamically occupies empty spaces.

Parameters/Return

```
Ihandle* IupFill(void); [in C]
iupfill{} -> elem: iuplua_tag [in IupLua3]
iup.fill{} -> elem: iuplua_tag [in IupLua5]
fill() [in LED]
```

This function returns the identifier of the created **Fill**, or NULL if an error occurs.

Attributes

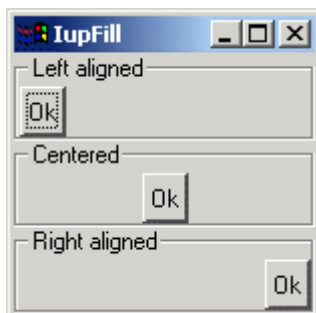
[SIZE](#): Defines the width, if the **Fill** is inside a horizontal box, or the height, if it is inside a vertical box. Default: "0".

[EXPAND](#): The default value is "YES", which fills every possible space.

Note

This element is used to maintain the dialog's layout untouched after the user made size changes, and to align the interface elements.

Examples



See Also

[IupHbox](#), [IupVbox](#).

IupHbox

Creates an hbox interface element. It is a box that shows the elements it contains, horizontally and from left to right.

Parameters/Return

```
Ihandle* IupHbox(Ihandle *elem1, Ihandle *elem2, ..., NULL); [in C]
iuphbox{elem1, elem2, ...: iuplua_tag} -> (elem: iuplua_tag) [in IupLua3]
iup.hbox{elem1, elem2, ...: iuplua_tag} -> (elem: iuplua_tag) [in IupLua5]
hbox(elem1, elem2, ...) [in LED]
```

elem1, **elem2**, ...: List of identifiers that will be placed in the box. NULL defines the end of the list.

This function returns the identifier of the created hbox.

Attributes

[ALIGNMENT](#): Aligns the elements vertically. Possible values:

"ATOP", "ACENTER", "ABOTTOM".

Default: "ATOP".

[GAP](#): Defines a space in pixels between the interface elements.

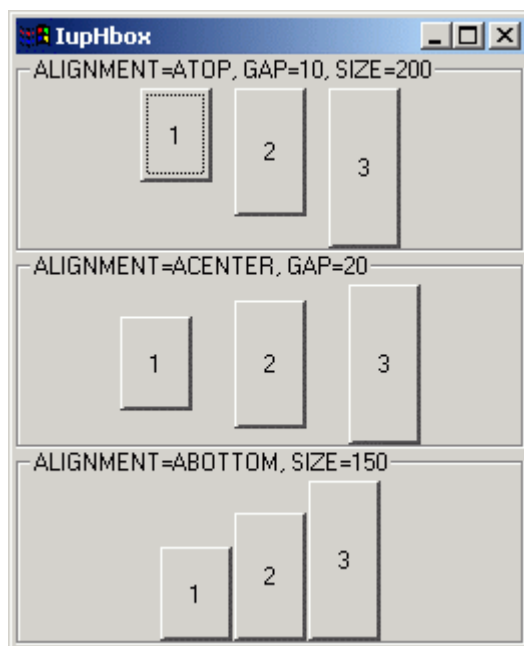
[MARGIN](#): Defines a margin in pixels.

[SIZE](#): Width of the hbox. Default: the smallest size that contains the children elements.

Note

The box can be created with no elements and be dynamic filled using [IupAppen](#).

Examples



See Also

[IupZbox](#), [IupVBox](#)

IupRadio

Creates the `radio` interface element. Only one of its descendant toggles to be activated at a time.

Parameters/Return

```
Ihandle* IupRadio(Ihandle *element); [in C]
iupradio{element: iuplua_tag} -> (elem: iuplua_tag) [in IupLua3]
iup.radio{element: iuplua_tag} -> (elem: iuplua_tag) [in IupLua5]
radio(element) [in LED]
```

element: Identifier of an interface element. Usually it is a vbox or an hbox containing the toggles associated to the radio.

This function returns the identifier of the created radio, or `NULL` (`nil` in IupLua) if an error occurs.

Attributes

[VALUE](#): Identifier of the activated toggle. The identifier is set by means of `IupSetHandle`.

Examples



IupVbox

Creates a vbox interface element. It is a box that shows the elements it contains, vertically and from the top down.

Parameters/Return

```
Ihandle* IupVbox(Ihandle *elem1, Ihandle *elem2, ..., NULL); [in C]
iupvbox{elem1, elem2, ...: iuplua_tag} -> (elem: iuplua_tag) [in IupLua3]
iup.vbox{elem1, elem2, ...: iuplua_tag} -> (elem: iuplua_tag) [in IupLua5]
vbox(elem1, elem2, ...) [in LED]
```

elem1, elem2, ...: List of the identifiers that will be placed in the box. NULL defines the end of the list.

This function returns the identifier of the created vbox, or NULL (nil in Lua) if an error occurs.

Attributes

[ALIGNMENT](#): Horizontally aligns the elements. Possible values:

"ALEFT", "ACENTER", "ARIGHT".

Default: "ALEFT".

[GAP](#): Defines a space, in pixels, between the interface elements.

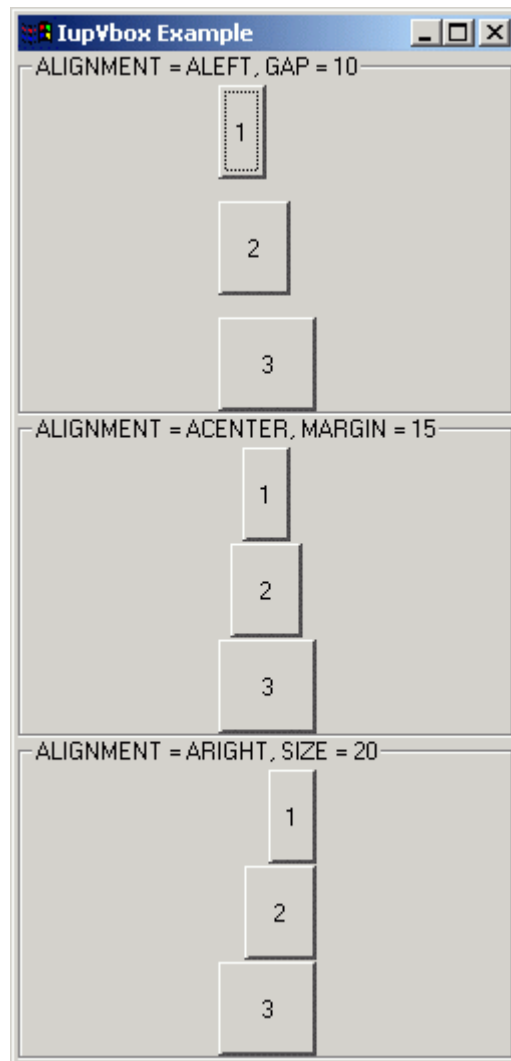
[MARGIN](#): Defines a margin in pixels.

[SIZE](#): Height of the vbox. Default: smallest size that contains the children elements.

Note

The box can be created with no elements and be dynamic filled using [IupAppen](#).

Examples



See Also

[IupZbox](#), [IupHbox](#)

IupZbox

Creates a zbox interface element. It is a box that piles up the elements it contains.

Parameters/Return

```
Ihandle* IupZbox (Ihandle *elem1, Ihandle *elem2,..., NULL); [in C]
iupzbox{elem1, elem2, ... : iuplua_tag} -> (elem: iuplua_tag) [in IupLua3]
iup.zbox{elem1, elem2, ... : iuplua_tag} -> (elem: iuplua_tag) [in IupLua5]
zbox(elem1, elem2,...) [in LED]
```

elem1, elem2, ...: List of the identifiers that will be placed in the box.

Note that, in C, NULL must be added as the last element, defining the end of the list.

This function returns the identifier of the created zbox, or NULL (nil in Lua) if an error occurs.

Attributes

[ALIGNMENT](#): Defines the alignment of the visible element. Possible values:

```
"NORTH" , "SOUTH" , "WEST" , "EAST" ,  
"NE" , "SE" , "NW" , "SW" ,  
"ACENTER" .
```

Default: "NE" .

[MARGIN](#): Defines the margin of the visible element.

[VALUE](#): Defines the visible element. The value passed must be the identifier of one of the elements contained in the `zbox`. Default: the first element.

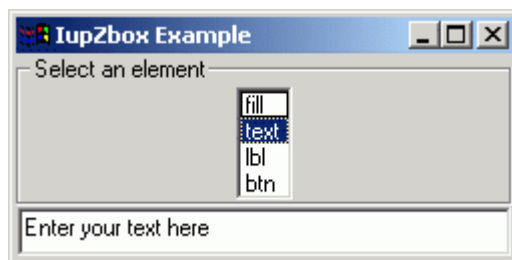
[SIZE](#): Defines the `zbox` size. Default: the smallest size that fits its largest element.

Note

The box can be created with no elements and be dynamic filled using [IupAppen](#).

Though this element can have attributes `ALIGNMENT` and `MARGIN`, it does not have attribute [GAP](#).

Examples



See Also

[IupHbox](#) , [IupVBox](#)

IupItem

Creates an item of the menu interface element. When selected, it generates an action.

Parameters/Return

```
Ihandle* IupItem(char *title, char *action); [in C]  
iupitem(title = title: string) -> elem: iuplua_tag [in IupLua3]  
iup.item(title = title: string) -> elem: iuplua_tag [in IupLua5]  
item(title, action) [in LED]
```

title: Text to be shown on the item.

action: Name of the action generated when the item is selected.

This function returns the identifier of the created item.

Attributes

[KEY](#): Associates a key to the item.

[VALUE](#): Indicates the item's state. When the value is ON, a mark will be displayed to the left of the item. Default: OFF.

[TITLE](#): Text shown to the user. It is possible to change its value on-the-fly.

[IMAGE](#): (Windows Only) Image of the non-checked menu item.

[IMPRESS](#): (Windows Only) Image of the checked menu item.

Callbacks

[ACTION](#): Action generated when the item is selected.

[HIGHLIGHT_CB](#): Action generated when the item is highlighted.

Notes

The text of the menu item accepts the control character '\t' to force text alignment to the right after this character. This is used to add shortcut keys to the menu, aligned to the right. Ex.: "Save\tCtrl+S".

Menu items are activated using the Enter key.

Attention: Never use the same menu item in different menus.

Examples

See Also

[IupSeparator](#), [IupSubmenu](#), [IupMenu](#).

IupMenu

Creates a menu element, which groups 3 types of interface elements: item, submenu and separator. Any other interface element defined inside a menu will be ignored.

Parameters/Return

```
Ihandle* IupMenu(Ihandle *elem1, Ihandle *elem2,..., NULL); [in C]
iupmenu{elem1, elem2, ...: iuplua_tag} -> (elem: iuplua_tag) [in IupLua3]
iup.menu{elem1, elem2, ...: iuplua_tag} -> (elem: iuplua_tag) [in IupLua5]
menu(elem1, elem2, ...) [in LED]
```

elem1, elem2, ...: List of identifiers that will be grouped by the menu. NULL defines the end of the list in C.

This function returns the identifier of the created menu, or NULL if an error occurs.

Note

A menu can be that of a dialog bar, defined by the dialog's MENU attribute, or a popup menu. A popup menu is displayed for the user (usually on the mouse position) and disappears when an item is selected. Its implementation is done by means of a call to the `IupPopup` function. `IupDestroy` should be called only for popup menus.

Lua Binding

Offers a "cleaner" syntax than LED for defining menu, submenu and separator items. The list of elements in the menu is described as a string, with one element after the other, separated by commas.

Each element can be:

- 1) { "<item_name>" , "<action>" } - menu item
- 2) { "<submenu_name>" , "<menu>" } - submenu
- 3) {} - separator
- 4) <interface element> - submenu item

Callbacks

[OPEN_CB](#): Called just before a submenu is opened.

[MENUCLOSE_CB](#): Called right before the submenu is closed.

Examples

See Also

[IupDialog](#), [IupPopup](#), [IupItem](#), [IupSeparator](#), [IupSubmenu](#)

IupSeparator

Creates the separator interface element. It shows a line between two menu items.

Parameters/Return

```
Ihandle* IupSeparator(void); [in C]
iupseparator{} -> (elem: iuplua_tag) [in IupLua3]
iup.separator{} -> (elem: iuplua_tag) [in IupLua5]
separator() [in LED]
```

This function returns the identifier of the created separator, or NULL if an error occurs.

Note

The separator is ignored when it is part of the definition of the items in a bar menu.

Examples

See Also

[IupItem](#), [IupSubMenu](#), [IupMenu](#).

IupSubMenu

Creates a menu item that, when selected, opens another menu.

Parameters/Return

```
Ihandle* IupSubMenu(char *title, Ihandle *menu); [in C]
iupsubmenu{menu: iuplua_tag; title = title: string} -> (elem: iuplua_tag) [in
iup.submenu{menu: iuplua_tag; title = title: string} -> (elem: iuplua_tag) [in
submenu(title, menu) [in LED]
```

title: String containing the text to be shown on the item. It is a creation-only attribute and cannot be changed later.

menu: menu identifier.

This function returns the identifier of the created submenu, or NULL if an error occurs.

Attributes

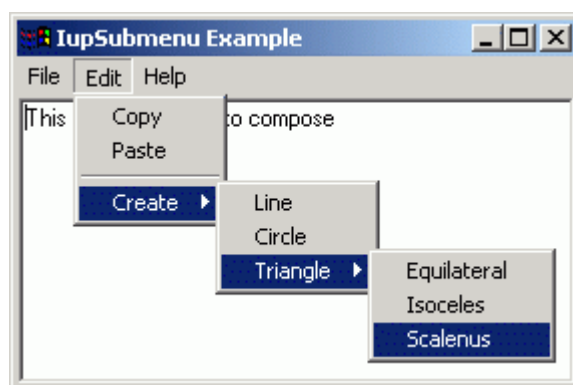
KEY: Associates a key to the submenu. In Windows, when used will also set an underscore on the respective letter of the submenu title.

Callbacks

OPEN_CB: Called just before the submenu is opened.

MENUCLOSE_CB: Called right before the submenu is closed.

Examples



See Also

[IupItem](#), [IupSeparator](#), [IupMenu](#).

IupColorBrowser

Creates an element for selecting colors from the HLS (Hue Saturation Brightness) model, which allows the user to interactively choose a color.

For a dialog that simply returns the selected color, you can use function [IupGetColor](#).

Parameters/Return

```
Ihandle* IupColorBrowser(void); [in C]
iupcb{} (elem: iuplua_tag) [in IupLua3]
iup.colorbarbrowser{} (elem: iuplua_tag) [in IupLua5]
colorbrowser() [in LED]
```

The function returns the identifier of the created colorbrowser, or NULL if an error occurs.

Attributes

RGB: Contains the color selected in the control, in the “rrr ggg bbb” format; r, g and b are integers ranging from 0 to 255. This value can both be consulted and modified.

Callbacks

DRAG_CB: Called several times while the color is being changed by dragging the mouse over the control.

```
int drag(Ihandle *self, unsigned char r, unsigned char g, unsigned char b)
elem:drag(r: number, g: number, b: number) -> (ret: number) [in IupLua3]
elem:drag_cb(r: number, g: number, b: number) -> (ret: number) [in IupLua5]
```

CHANGE_CB: Called when the user releases the left mouse button over the control, defining the selected color.

```
int change(Ihandle *self, unsigned char r, unsigned char g, unsigned char b)
elem:change(r: number, g: number, b: number) -> (ret: number) [in IupLua3]
elem:change_cb(r: number, g: number, b: number) -> (ret: number) [in IupLua5]
```

Examples

IupDial

Creates a dial for regulating a given angular variable. It inherits from [IupCanvas](#).

Parameters/Return

```
Ihandle* IupDial(char *type); [in C]
iupdial{type: string} -> (elem: iuplua_tag) [in IupLua3]
iup.dial{type: string} -> (elem: iuplua_tag) [in IupLua5]
dial(type) [in LED]
```

tipo: dial type. Can be "HORIZONTAL", "VERTICAL" or "CIRCULAR".

The function returns the identifier of the created dial, or NULL if an error occurs.

Attributes

FGCOLOR: Controls the foreground color. The default value is "64 64 64". The foreground color is not used for the circular dial.

BGCOLOR: Controls the background color. The default value is the parent or the dialog background color.

DENSITY: Contains average value of the number of lines per pixel in the dial. The purpose of this attribute is to maintain the control's appearance when its size changes. Default is "0.2".

UNIT: Contains the unit of the angle. Can be "DEGREES" or "RADIANS". Default is "RADIANS".

VALUE: Contains the dial value in a given moment. The value is an angle starting at zero when the interaction started.

TYPE: Informs whether the dial is "VERTICAL", "HORIZONTAL" or "CIRCULAR".

EXPAND: The default is "NO".

SIZE: the default is "16x80", "80x16" or "40x35" according to the dial type.

Callbacks

MOUSEMOVE_CB: Called each time the user moves the dial with the mouse button pressed. The angle the dial rotated since it was initialized is passed as a parameter.

```
int function(Ihandle *self, double angle); [in C]
elem:mousemove(angle: number) -> (ret: number) [in IupLua3]
elem:mousemove_cb(angle: number) -> (ret: number) [in IupLua5]
```

BUTTON_PRESS_CB: Called when the user presses the left mouse button over the dial. The angle here is always zero, except for the circular dial.

```
int function(Ihandle *self, double angle)
elem:buttonpress(angle: number) -> (ret: number) [in IupLua3]
elem:buttonpress_cb(angle: number) -> (ret: number) [in IupLua5]
```

BUTTON_RELEASE_CB: Called when the user releases the left mouse button after pressing it over the dial.

```
int function(Ihandle *self, double angle)
elem:buttonrelease(angle: number) -> (ret: number) [in IupLua3]
elem:buttonrelease_cb(angle: number) -> (ret: number) [in IupLua5]
```

Notes

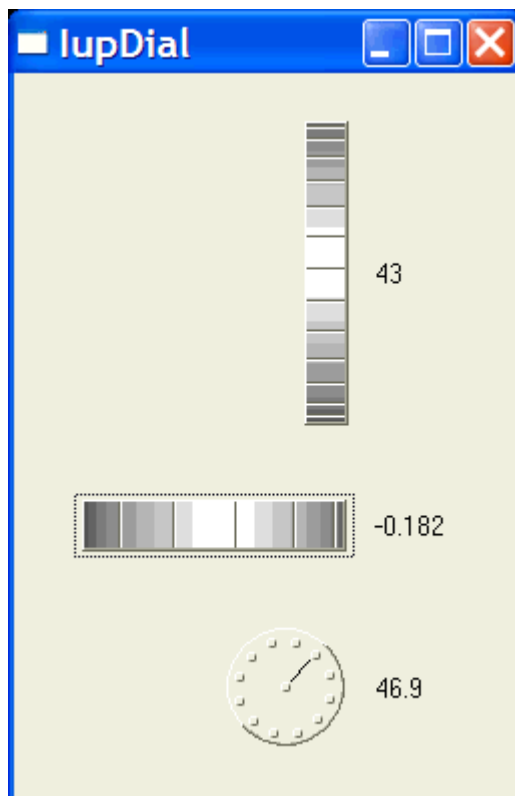
When the keyboard arrows are pressed and released the mouse press and the mouse release callbacks are called in this order. If you hold the key down a mouse move callback is also called.

When the wheel is rotated only the mouse move callback is called, and it increments the last angle the dial was rotated.

In these cases the value is incremented by $\pi/10$ (18 degrees).

Examples

Creates several `Dials` and shows each dial's value in a `Label`.



See Also

[IupCanvas](#)

IupGauge

Creates a Gauge control. Shows a percent value that can be updated to simulate a progression. It inherits from [IupCanvas](#).

Parameters/Return

```
Ihandle* IupGauge(void); [in C]
iupgauge{} -> (elem: iuplua_tag) [in IupLua3]
iup.gauge{} -> (elem: iuplua_tag) [in IupLua5]
gauge() [in LED]
```

The function returns the identifier of the created Gauge, or NULL if an error occurs.

Attributes

MIN: Contains the minimum valuator value. Default is "0".

MAX: Contains the maximum valuator value. Default is "1".

VALUE: Contains a number between "MIN" and "MAX", indicating the gauge position.

DASHED: Changes the style of the gauge for a dashed pattern. Default is "NO".

MARGIN: Changes the distance from the Gauge's border to its inside. It is only one number that works in both directions (x and y). Default: 1.

Ex.: `IupSetAttribute(mygauge, "MARGIN", "5")`;

TEXT: Contains a text to be shown inside the Gauge. If it is NULL, the percentage value given by `VALUE` will be shown. If the gauge is dashed the text is never shown.

SHOW_TEXT: Indicates if the text inside the Gauge is to be shown or not. Possible values:

"YES" or "NO". Default: "YES".

FGCOLOR: Controls the gauge and text color. The default is "64 96 192".

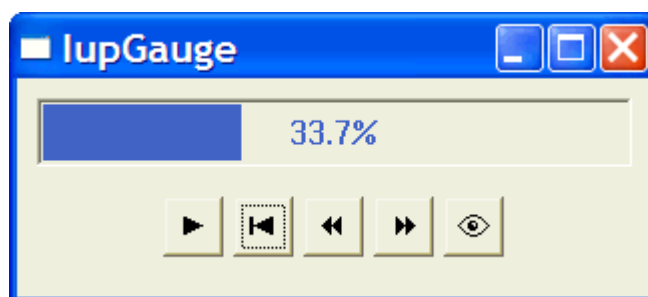
[FONT](#): Character font of the text.

[SIZE](#): The default is "170x17".

EXPAND: The default is "NO".

Examples

Creates a Gauge with a control bar.



See Also

[IupCanvas](#)

IupGetColor

Shows the default IUP dialog which allows the user to select a color.

Parameters/Return

```
int IupGetColor(int x, int y, unsigned char *r, unsigned char *g, unsigned char *b);
IupGetColor(x, y, r, g, b: number) -> (r, g, b: number) [in IupLua3]
iup.GetColor(x, y, r, g, b: number) -> (r, g, b: number) [in IupLua5]
```

x, y: x, y values of the `IupPopup` function.

r, g, b: Pointers to variables that will receive the color selected by the user if the OK button is pressed. The value in the variables at the moment the function is called defines the color being selected when the dialog is shown. If the OK button is not pressed, the r, g and b values are not

changed. These values cannot be NULL.

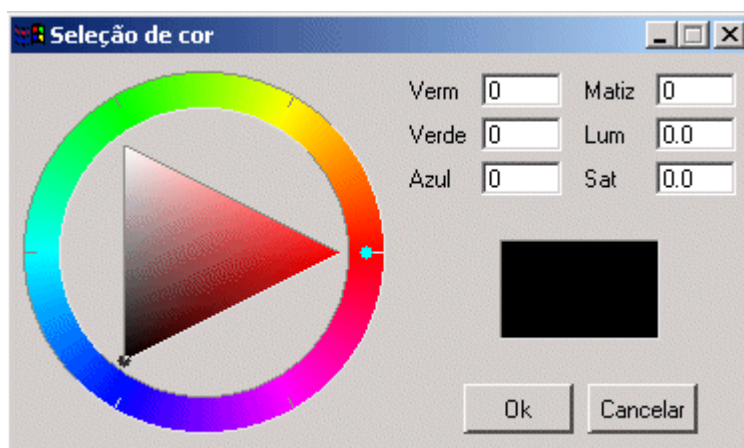
The function returns 1 if the OK button is pressed, or 0 otherwise.

Notes

In systems with few colors available (256), this function will show the colors by automatically performing dithering, providing good results. However, if only a few colors are available at the system's palette, strange artifacts may appear.

The dialog uses a global attribute called "PARENTDIALOG" as the parent dialog if it is defined.

Examples



See Also

[IupMessage](#), [IupScanf](#), [IupListDialog](#), [IupAlarm](#), [IupGetFile](#).

IupGetParam

Shows a popup dialog for capturing parameter values using several types of controls.

Parameters/Return

```
int IupGetParam(const char* title, Iparamcb action, void* user_data, const char* format, ...);
```

[Not available in IupLua]

title: dialog title.

action: user callback to be called whenever a parameter value was changed, and when the user pressed the OK button. It can be NULL.

user_data: user pointer repassed to the user callback.

format: string describing the parameter

...: list of variables address with initial values for the parameters. The last variable must be NULL.

The function returns 1 if the OK button is pressed, 0 if the user canceled or if an error occurred. The function will abort if there are errors in the format string as in the number of the expected parameters.

Callback

```
typedef int (*Iparamcb)(Ihandle* dialog, int param_index, void* user_data);
```

dialog: dialog handle

param_index: current parameter being changed. It is -1 if the user pressed the **OK** button. It is -2 when the dialog is **mapped**, just before shown. It is -3 if the user pressed the **Cancel** button.

user_data: a user pointer that is passed in the function call.

You can reject the change or the OK action by returning "0" in the callback, otherwise you must return "1".

You should not programmatically change the current parameter value during the callback. On the other hand you can freely change the value of other parameters.

Use the dialog attribute "PARAMn" to get the parameter "Ihandle*", but not that this is not the actual control. Where "n" is the parameter index in the order they are specified starting at 0, but separators are not counted. Use the parameter attribute "CONTROL" to get the actual control. For example:

```
Ihandle* param2 = (Ihandle*)IupGetAttribute(dialog, "PARAM2");
int value2 = IupGetInt(param2, IUP_VALUE);

Ihandle* param5 = (Ihandle*)IupGetAttribute(dialog, "PARAM5");
Ihandle* ctrl5 = (Ihandle*)IupGetAttribute(param5, "CONTROL");

if (value2 == 0)
{
    IupSetAttribute(param5, IUP_VALUE, "New Value");
    IupSetAttribute(ctrl5, IUP_VALUE, "New Value");
}
```

Since parameters are user controls and not real controls, you must update the control value and the parameter value.

Be aware that programmatically changes are not filtered. The valuator, when available, can be retrieved using the parameter attribute "AUXCONTROL". The valuator is not automatically updated when the text box is changed programmatically. The parameter label is also available using the parameter attribute "LABEL".

Attributes (inside the callback)

For the dialog:

"PARAMn" - returns an IUP Ihandle* representing the nth parameter, indexed by the declaration order not counting separators.
 "OK" - returns an IUP Ihandle*, the main button.
 "CANCEL" - returns an IUP Ihandle*, the close button.

For a parameter:

"LABEL" - returns an IUP Ihandle*, the label associated with the parameter.
 "CONTROL" - returns an IUP Ihandle*, the real control associated with the

parameter.

"AUXCONTROL" - returns an IUP `Ihandle*`, the auxiliary control associated with the parameter (only for Valuator).

"INDEX" - returns an integer value associated with the parameter index.

`IupGetInt` can also be used.

"VALUE" - returns the parameter value as a string, but `IupGetFloat` and `IupGetInt` can also be used.

Notes

The format string must have the following format, notice the "\n" at the end

"**text**%**t**[**extra**]\n", where:

text is a descriptive text, to be placed to the left of the entry field in a label.

t is the type of the parameter. The valid options are:

b = boolean (shows a True/False toggle, use "int" in C)

i = integer (shows a integer filtered text box, use "int" in C)

r = real (shows a real filtered text box, use "float" in C)

a = angle in degrees (shows a real filtered text box and a dial, use "float" in C)

s = string (shows a text box, use "char*" in C, it must have room enough for your string)

m = multiline string (shows a multiline text box, use "char*" in C, it must have room enough for your string)

l = list (shows a dropdown list box, use "int" in C for the zero based item index selected)

t = separator (shows a horizontal line separator label, in this case text can be an empty string)

extra is one or more additional options for type **t**

[**min,max**] are optional limits for integer and real types. The maximum value can be omitted. When both are specified a valuator will also be added to change the value.

[**false,true**] are optional strings for boolean types. The strings can not have commas ',', nor brackets '[' or ']'.
 [**item0** | **item1** | **item2**,...] are the items of the list. At least one item must exist. Again the brackets are not used to increase the possibilities for

mask is an optional mask for the string and multiline types. The dialog uses the [IupMask](#) internally. In this case we do not use the brackets '[' and ']' to avoid confusion with the specified mask.

[**item0** | **item1** | **item2**,...] are the items of the list. At least one item must exist. Again the brackets are not used to increase the possibilities for

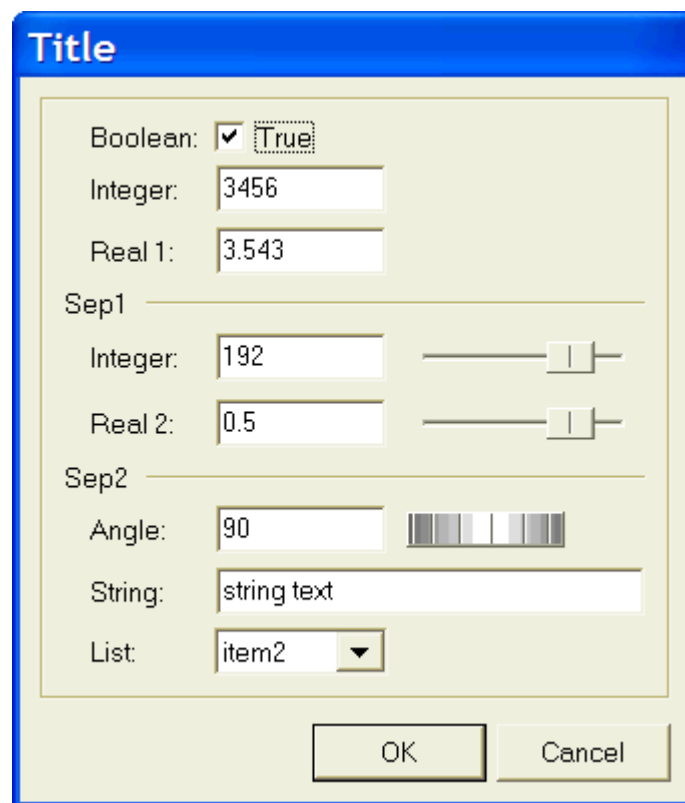
the strings, instead you must use '|'. Items index are zero based start.

The dialog is resizable if it contains a string, a multiline string or a number with a valuator. All the multiline strings will increase size equally in both directions.

The dialog uses a global attribute called IUP_PARENTDIALOG as the parent dialog if it is defined.

Examples

Here is an example showing many the possible parameters. We show only one for each type, but you can have as many parameters of the same type you want.



See Also

[IupScanf](#), [IupGetColor](#), [IupMask](#), [IupValuator](#), [IupDial](#), [IupList](#).

IupMask

Allows associating a mask to a IupText or a IupMatrix element.

See the [Pattern Specification](#) for information on patterns.

Functions

```
int iupMaskSet(Ihandle *h, char *mask, int autofill, int casei) [in C or in Iup]
int iupMaskMatSet(Ihandle *h, char *mask, int autofill, int casei, int lin, int col)
```

These functions are responsible for setting the mask to be used.

h: Ihandle of IupText or IupMatrix

mask: Mask to be used

autofill: When “1”, turns the auto-fill mode on. In auto-fill mode, whenever possible, literal characters will be automatically added to the field

casei: When “1”, uppercase or lowercase characters will be treated indistinctly

lin, col: Line and column numbers in the matrix

They return 1 if the mask is set, or 0 if there is an error (e.g., invalid mask).

```
int iupMaskSetInt(Ihandle *h, int autofill, int min, int max); [in C or in Iup]
int iupMaskSetFloat(Ihandle *h, int autofill, float min, float max); [in C or
int iupMaskMatSetInt(Ihandle *h, int autofill, int min, int max, int lin, int col);
int iupMaskMatSetFloat(Ihandle *h, int autofill, float min, float max, int lin, int col);
```

These functions set a mask that defines a limit to the typed number. Works only for integers and floats. Limitations: since the validation process is performed key by key, the user cannot type intermediate numbers (even inside the limit) if they are not following predetermined rules

h: Ihandle of IupText or IupMatrix

autofill: When “1”, turns the auto-fill mode on. In auto-fill mode, whenever possible, literal characters will be automatically added to the field

min: Minimum value accepted in the field

max: Maximum value accepted in the field

lin, col: Line and column numbers in the matrix

They always return 1.

```
int iupMaskCheck (Ihandle *h); [in C or in IupLua]
int iupMaskMatCheck(Ihandle *h, int lin, int col); [in C or in IupLua]
```

These functions verify if what was typed by the user is valid for the defined mask.

h: Ihandle of IupText or IupMatrix

lin, col: Line and column numbers in the matrix

They return 1 if the text satisfies the mask, or 0 otherwise.

```
int iupMaskGet(Ihandle *h, char **val); [in C]
int iupMaskGetFloat(Ihandle *h, float *fval); [in C]
int iupMaskGetInt(Ihandle *h, int *ival); [in C]
int iupMaskMatGet(Ihandle *h, char **val, int lin, int col); [in C]
int iupMaskMatGetFloat(Ihandle *h, float *fval, int lin, int col); [in C]
int iupMaskMatGetDouble(Ihandle *h, double *dval, int lin, int col); [in C]
int iupMaskMatGetInt(Ihandle *h, int *ival, int lin, int col);
```

These functions check if the mask is complete, and they retrieve the field's value in only one call.

h: Ihandle of IupText or IupMatrix

val, fval, ival: Pointers used to complete the return value

lin, col: Line and column numbers in the matrix.

They return 1 if the text satisfies the mask, or 0 otherwise.

Notes

User callbacks previously associated to the text-editing field or to the Matrix field (that is, before the `iupMaskSet` function is called) will be called by the library if the pressed key satisfies the mask. Attention: for the callback to be actually called, the user must call not only `IupSetAttribute`, but also `IupSetFunction`.

To make the use of masks simpler, the following predefined masks were created:

IUPMASK_FLOAT - Float number
 IUPMASK_UFLOAT - Float number with no sign
 IUPMASK_EFLOAT - Float number with exponential notation
 IUPMASK_INT - Integer number
 IUPMASK_UINT - Integer number with no sign

Examples

IupSbox

Creates a `split` panel control. Allows the provided control to be enclosed in a box that allows resizing.

Parameters/Return

```
Ihandle* IupSbox(Ihandle* elem); [in C]
iupsbox{elem: iuplua_tag} -> (elem: iuplua_tag) [in IupLua3]
iup.sbox{elem: iuplua_tag} -> (elem: iuplua_tag) [in IupLua5]
sbox(elem) [in LED]
```

elem: This function receives as parameter the element that will be enclosed in a Sbox.

This function returns the created Sbox's identifier, or `NULL` if an error occurs.

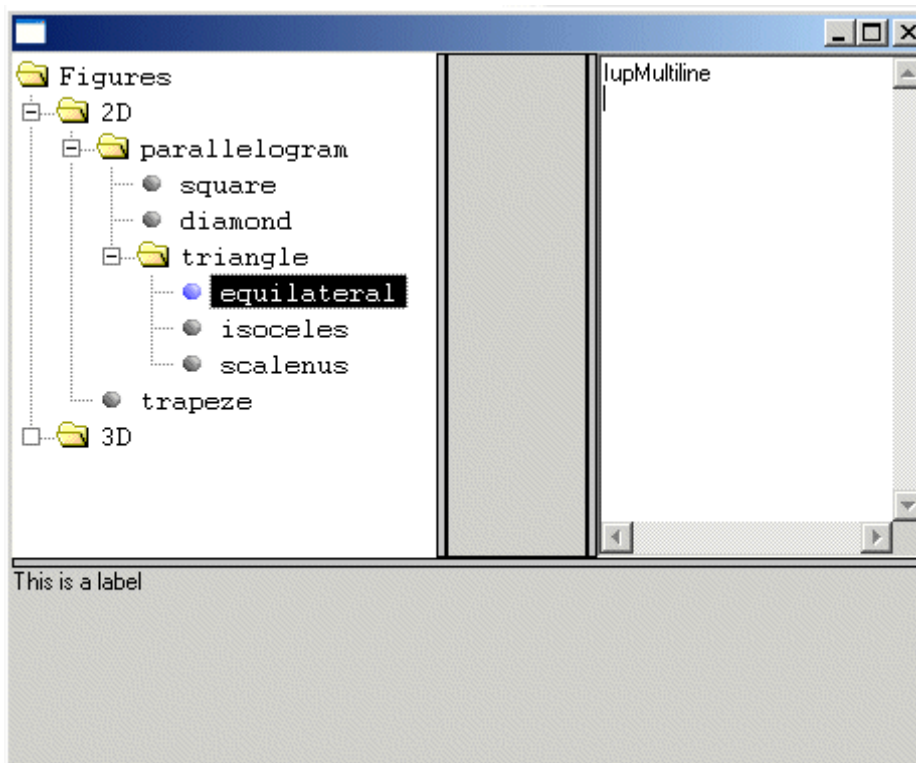
Attributes

DIRECTION: Indicates the direction of the resize. Possible values are:

"NORTH", "SOUTH", "EAST", "WEST".

COLOR: Changes the color of the Sbox's thumb. The value should be given in "R G B" color style.

Examples



Example 2 image

IupTabs

Creates a Tabs element. Allows a single dialog to have several screens, grouping options. The grouping is done in a single line of tabs arranged according to the tab type. It inherits from [IupCanvas](#). It contains a [IupZbox](#) to control the groups of controls.

Parameters/Return

```
Ihandle* IupTabs(Ihandle* elem1, Ihandle* elem2, ...); [in C]
Ihandle* IupTabsv(Ihandle** elems); [in C]
iuptabs{elem1, elem2, ...: iuplua_tag} -> (elem: iuplua_tag) [in IupLua3]
iup.tabs{elem1, elem2, ...: iuplua_tag} -> (elem: iuplua_tag) [in IupLua5]
tabs(elem1, elem2, ...) [in LED]
```

elem1, elem2, ...: This function receives as parameters the elements that will be transformed into Tabs. Each of such elements must have a "**TABTITLE**" attribute, specifying the text to be shown in its tab's title. **If this attribute is omitted, the Tabs behavior is undetermined.**

This function returns the created Tabs's identifier, or NULL if an error occurs. The second form in C must end the array with a NULL.

Attributes

ALIGNMENT: Changes the respective attribute in the internal zbox.

TABTITLE: Contains the text to be shown in the tab's title. If this value is NULL, it will remain empty. This attribute is used only in the elements contained in the tab.

TABTYPE: Indicates the type of tab, which can be one of the following:

"TOP" , "BOTTOM" , "LEFT" or "RIGHT". Default is "TOP".

FONT: Indicates the font to be used in the internal tab text. [Font Table](#)

FONT_ACTIVE: Indicates the font to be used when the tab is selected. [Font Table](#)

FONT_INACTIVE: Indicates the font to be used when the tab is inactive. [Font Table](#)

TABSIZE: Contains the size of a tab. If this value is NULL, the tab will be shown with the smallest possible value that fits its title. This size can refer to the whole IupTabs, thus affecting all tabs, or to a specific tab. If both are defined, the size of a specific tab will have priority over the global IupTabs size.

VALUE: Contains the *name* of the currently selected tab. Changing this attribute, adding the name of a different tab, makes the latter be the active tab. If the provided name does not correspond to any tab, nothing occurs. To set a name to a tab, use the [IupSetHandle](#) function on the element to be inserted in the tab.

ACTIVE: Allows or inhibits user interaction with a given tab. When the attribute is "NO", the corresponding tab modifies the text color to show that interaction is inhibited. Be careful, because a "REPAINT" may be needed to generate a Tabs repaint.

REPAINT: This attribute immediately generates a Tabs repaint.

Callbacks

TABCHANGE_CB: Callback called when the **user** shifts the active tab. The parameters passed are:

```
int function(Ihandle* self, Ihandle* new_tab, Ihandle* old_tab); [in C]
elem:tabchange(new_tab, old_tab: iuplua_tag) -> (ret: number) [in IupLua]
elem:tabchange_cb(new_tab, old_tab: iuplua_tag) -> (ret: number) [in IupLua]
```

self: Ihandle* of the control

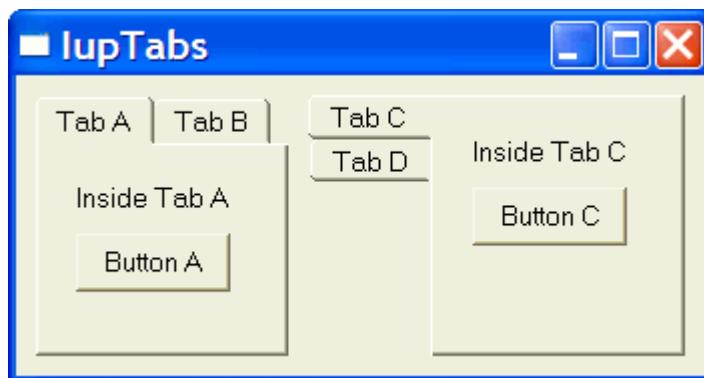
new_tab: Ihandle* of the tab selected by the user

old_tab: Ihandle* of the previously selected tab

Note

The Tabs elements, differently from a ZBOX, **does not** need to have associated *names*. Those without a name will receive an automatically generated one.

[Examples](#)



See Also

[IupCanvas](#)

IupVal

Creates the Valuator control. It allows creating a regulator similar to `IupDial`, but with well-defined limits. It inherits from [IupCanvas](#).

Parameters/Return

```
Ihandle* IupVal(char *type); [in C]
iupval{type: string} -> (elem: iuplua_tag) [in IupLua3]
iup.val{type: string} -> (elem: iuplua_tag) [in IupLua5]
val(type) [in LED]
```

type: Type of valuator. Can be "VERTICAL" or "HORIZONTAL" .

The function returns the identifier of the created val, or NULL if an error occurs.

Attributes

MIN: Contains the minimum valuator value. Default is "0".

MAX: Contains the maximum valuator value. Default is "1".

VALUE: Contains a number between MIN and MAX, indicating the valuator position.

TYPE: Informs whether the valuator is "VERTICAL" or "HORIZONTAL". Vertical valutors are bottom to up, and horizontal valutors are left to right variations of min to max.

SHOWTICKS: Display tick mark along the valuator trail. The attribute controls the number of ticks. Minimum value is "3". Default is "0", in this case the ticks are not shown. The precision of the ticks are affected by the raster size of the control.

BGCOLOR: Controls the background color. The default value is the parent or the dialog background color.

[RASTERSIZE](#): The default is "124x28" or "28x124". We recomend to leave this as the minimum size.

EXPAND: The default is "NO". The thumb will not expand if the valuator is expanded.

Callbacks

MOUSEMOVE_CB: Called each time the user moves the valuator's thumb keeping the mouse button pressed. The value of `VALUE` is passed as parameter.

```
int function(Ihandle *self, double val); [in C]
elem:mousemove(val: number) -> (ret: number) [in IupLua3]
elem:mousemove_cb(val: number) -> (ret: number) [in IupLua5]
```

BUTTON_PRESS_CB: Called when the user presses the left mouse button over the valuator. The value of `VALUE` is passed as parameter. The thumb is always repositioned to the current mouse position.

```
int function(Ihandle *self, double val); [in C]
elem:buttonpress(val: number) -> (ret: number) [in IupLua3]
elem:buttonpress_cb(val: number) -> (ret: number) [in IupLua5]
```

BUTTON_RELEASE_CB: Called when the user releases the mouse button, after having pressed it over the valuator. The value of `VALUE` is passed as parameter.

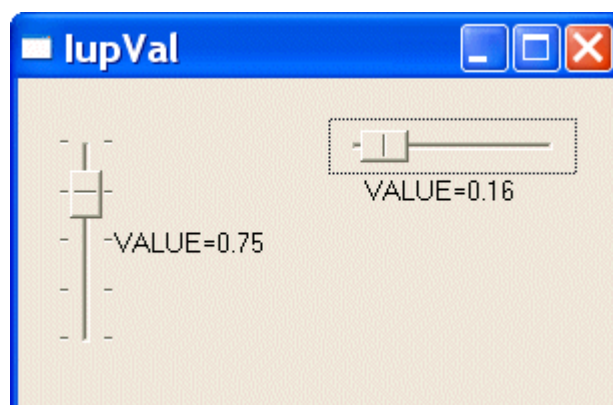
```
int function(Ihandle *self, double val); [in C]
elem:buttonrelease(val: number) -> (ret: number) [in IupLua3]
elem:buttonrelease_cb(val: number) -> (ret: number) [in IupLua5]
```

Notes

When the keyboard arrows are pressed and released, or the mouse wheel is rotated, the mouse press and the mouse release callbacks are called, in this order. If you hold the key down a mouse move callback is also called.

In these cases the value is incremented by 10% of the interval max-min.

Examples



See Also

[IupCanvas](#)

IupMatrix

Creates a matrix of alphanumeric fields. Therefore, all values of the matrix's fields are strings. The matrix is not a grid container like many systems have. It inherits from [IupCanvas](#).

It has two modes of operation: normal and callback mode. In normal mode string values are stored in attributes for each cell. In callback mode these attributes are ignored and the cells are filled with strings returned by the "VALUE_CB" callback. So the existence of this callback defines the mode the matrix will operate.

Parameters/Return

```
Ihandle* IupMatrix(char *action); [in C]
iupmatrix{} -> (elem: iuplua_tag) [in IupLua3]
iup.matrix{} -> (elem: iuplua_tag) [in IupLua5]
matrix(action) [in LED]
```

action: Name of the action generated when the user types something.

Returns the identifier of the created matrix, or NULL if an error occurs.

Attributes

Cell Attributes

[L:C](#)
[ALIGNMENTn](#)
[BGCOLOR](#)
[FGCOLOR](#)
[FONT](#)
[FOCUS_CELL](#)
[VALUE](#)

Line and Column Attributes

[NUMCOL](#)
[NUMCOL_VISIBLE](#)
[NUMLIN](#)
[NUMLIN_VISIBLE](#)
[ORIGIN](#)
[WIDTHDEF](#)
[WIDTHn](#)
[HEIGHTn](#)
[RESIZEMATRIX](#)

Mark Attributes

[AREA](#)
[MARK_MODE](#)
[MARKED](#)
[MULTIPLE](#)

Action Attributes

[ADDCOL](#)

[ADDLIN](#)
[DELCOL](#)
[DELLIN](#)
[EDIT_MODE](#)
[REDRAW](#)

General Attributes

[CURSOR](#)
[FRAMECOLOR](#)
[SCROLLBAR](#)
[SIZE](#)
[CARET](#)
[SELECTION](#)
[HIDEFOCUS](#)

Callbacks

[ACTION](#) - Action generated when a keyboard event occurs.

[BGCOLOR_CB](#) - Action generated to retrieve the background color of a cell when it needs to be redrawn.

[CLICK_CB](#) - Action generated when any mouse button is pressed over a cell.

[DROP_CB](#) - Action generated to determine if a text field or a dropdown will be shown.

[DROPCHECK_CB](#) - Action generated to determine if a dropdown feedback should be shown.

[DROPSELECT_CB](#) - Action generated when an element in the dropdown list is selected.

[EDITION_CB](#) - Action generated when the current cell enters or leaves the edition mode.

[ENTERITEM_CB](#) - Action generated when a matrix cell is selected, becoming the current cell.

[FGCOLOR_CB](#) - Action generated to retrieve the foreground color of a cell when it needs to be redrawn.

[LEAVEITEM_CB](#) - Action generated when a cell is no longer the current cell.

[MOUSEMOVE_CB](#) - Action generated to notify the application that the mouse has moved over the matrix.

[SCROLL_CB](#) - Action generated when the matrix' visualization surface is changed.

[VALUE_CB](#) - Action generated to verify the value of a cell in the matrix when it needs to be redrawn.

[VALUE_EDIT_CB](#) - Action generated to notify the application that the value of a cell was changed.

Additional Functions in IupLua

```
elem:setcell(lin, col: number, value: string)
```

Modifies the text of a cell.

```
elem:getcell(lin, col: number) -> (cell: string)
```

Returns the text of a cell.

Notes

The [IupMask](#) control can be used to create a mask and filter the text entered by the user in each cell.

In Motif, when start editing a cell using a double click, the user must click again to the edit control get the focus.

Titles

A matrix might have titles for lines and columns. This must be defined before the matrix is mapped, and cannot be changed afterwards. A matrix will have line titles if, before it is mapped, an attribute of the “L : 0” type is defined. It will have column titles if, before being mapped, an attribute of the “0 : C” type is defined. The size of a line title is given by attribute “WIDTH0”, if it is defined. Otherwise, it is given by the size of the largest title defined when the matrix is mapped.

Titles (for lines or columns) can be generated with more that one text line. For such, the title value must contain a “\n”. The matrix does not by itself change the line’s height to fit titles with multiple lines. The user must change the line’s size manually, by using attribute HEIGHTn. In IUP’s size definition, a line with height 8 will fit one text line, a line with height 16 will fit two text lines, and so on.

Callback Mode

Very large matrices must use the callback mode to set the values, and not the regular value attributes of the cells. The idea is the following:

- 1 - Register the VALUE_CB callback
- 2 - No longer set the value of the cells. They will be set one by one by the callback. Note that the values of the cells must now be stored by the user.
- 3 - If the matrix is editable, set the VALUE_EDIT_CB callback.
- 4 - When the matrix must be invalidated, use the REDRAW attribute to force a matrix redraw.

A negative aspect is that, when VALUE_CB is defined, the matrix never verifies attributes of type “%d : %d”. Therefore, it also does not verify line and column titles (which must be given by the callback). The result is that, at the moment the matrix is created, it resorts solely to the existence of attributes WIDTH0 and HEIGHT0 to find out if it will have line or column titles. That is, for such matrices to have titles, the WIDTH0 and HEIGHT0 attributes must be defined. This problem is not serious, because with IUP’s definition of SIZE, HEIGHT0=8 will always produce a column title in the size desired.

Another important reminder: if VALUE_CB is defined and VALUE_EDIT_CB is not, it is absolutely necessary that the application does not allow the user to edit any cell. This must be done by returning IUP_IGNORE in the IUP_EDITION_CB callback. (In the future, this will be done inside the matrix.)

Utility Functions

These functions can be used to help set and get attributes from the matrix:

```
void IupMatSetAttribute (Ihandle *n, char* a, int l, int c, char* v);
void IupMatStoreAttribute(Ihandle *n, char* a, int l, int c, char* v);
char* IupMatGetAttribute (Ihandle *n, char* a, int l, int c);
int IupMatGetInt (Ihandle *n, char* a, int l, int c);
float IupMatGetFloat (Ihandle *n, char* a, int l, int c);
void IupMatSetfAttribute (Ihandle *n, char* a, int l, int c, char* f, .
```

They work just like the respective traditional set and get functions. But the attribute string is complemented with the L and C values. For ex:

```
IupMatSetAttribute (n, "" , 30, 10, v) = IupSetAttribute(n, "30:10", v)
IupMatSetAttribute (n, "BGCOLOR" , 30, 10, v) = IupSetAttribute(n, "BGCOI
IupMatSetAttribute (n, "ALIGNMENT" , 10, 0, v) = IupSetAttribute(n, "ALI
```

(*) noticed that in this case the second value will be ignored.

Navigation

Navigating through the matrix cells outside the edition mode is done by using the following keys:

- **Arrows:** Shift from the current cell to the next one, according to the arrow's direction.
- **Page Up** and **Page Down:** Scroll a spreadsheet up or down one page.
- **Home:** Shifts from the current cell to the first column in the line.
- **Home Home:** Shifts from the current cell to the upper left corner of the visible page.
- **Home Home Home:** Shifts from the current cell to the upper left corner of the first page of the matrix.
- **End:** Shifts from the current cell to the last column in the line.
- **End End:** Shifts from the current cell to the lower right corner of the visible page.
- **End End End:** Shifts from the current cell to the lower right corner of the last page in the matrix.
- **Del:** Deletes the contents of all selected cells.

Inside the edition mode, the following keys are used for a text field:

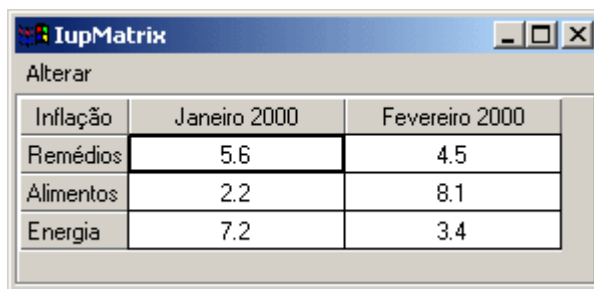
- **Up and down arrows:** Shift the current key, leaving the edition mode.
- **Left and right arrows:** If they are on the extremes of the text being edited, they shift the current key, leaving the edition mode.
- **Ctrl + Arrows:** Shift the current cell, leaving the edition mode.

When the matrix is outside the edition mode, pressing any character key makes the current key to enter in the edition mode, the old text is replaced by the new one being typed. If **Enter** or **Space** is pressed, the current cell enters the edition mode. If **Del** is pressed, the whole contents of the cell will be deleted. Double-clicking a cell also enters the edition mode.

When the matrix is in the edition mode, to confirm the entered value, press **Enter**. By pressing **Esc**, the previous value is restored. If the cell leaves the edition mode for any other reason (for instance, the user shifted the current cell, or the user clicked in another control), the value will be confirmed. After pressing **Enter** to confirm the value the focus goes to the cell below the current cell, if at the last line then the focus goes to the cell on the left.

Examples

Creates a simple matrix with the values and layout shown in the image below. There is also a menu that allows making some changes to the matrix.



Inflação	Janeiro 2000	Fevereiro 2000
Remédios	5.6	4.5
Alimentos	2.2	8.1
Energia	7.2	3.4

See Also

[IupCanvas](#)

IupMatrix Attributes

Cell Attributes

L:C: Text of the cell located in line L and column C, where L and C are integer numbers. These are valid only in normal mode.

L:0: Title of line L.

0:C: Title of column C.

0:0: Title of the area between the line and column titles.

ALIGNMENTn: Alignment of the cells in column n, where n must be replaced by the wished column number ($n \geq 0$). Possible values:

"ALEFT" , "ACENTER" or "ARIGHT". Default: "ALEFT".

BGCOLOR: Background color of the matrix.

BGCOLORL:C: Background color of the cell in line L and column C.

BGCOLORL:*: Background color of the cells in line L.

BGCOLOR*:C: Background color of the cells in column C.

When more than one of the four types of attributes that define the background color are defined, or if two of them are in conflict, the color of a cell will be selected following this priority: **BGCOLORL:C** , **BGCOLORL:*** , **BGCOLOR*:C**, and last **BGCOLOR**. (L or C ≥ 0 , but only the second forms is valid for titles.)

FGCOLOR: Text color.

FGCOLORL:C: Text color of the cell in line L and column C.

FGCOLORL:*: Text color of the cells in line L.

FGCOLOR*:C: Text color of the cells in column C.

When more than one of the four types of attributes that define the text color are defined, or if two of them are in conflict, the text color of a cell will be selected following this priority: **FGCOLORL:C** , **FGCOLORL:*** , **FGCOLOR*:C**, and last **FGCOLOR**. (L or C ≥ 0 , but only the second forms is valid for titles.)

FONT: Character font of the text. This attribute must be set before the control is showed. It affects the calculation of the size of all the matrix cells.

FONTL:C: Text font of the cell in line L and column C.

FONT*: Text font of the cells in line L.

FONT*:C: Text font of the cells in column C.

The cell size is always calculated from the base FONT attribute.

FOCUS_CELL: Defines the currently focused cell.

Two numbers in the "L : C" format, (L and C>=1). Default: "1 : 1".

VALUE: Allows setting or verifying the value of the current cell. Is the same as using the "L : C" attribute, L and C corresponding to the current cell's line and column. (L and C>=0)

Line and Column Attributes

NUMCOL: Defines the number of columns in the matrix.

Must be an integer number. Default: "0".

NUMCOL_VISIBLE: Defines the minimum number of visible columns in the matrix. The remaining columns will be accessible only by using the scrollbar.

Must be an integer number. Default: "4".

NUMLIN: Defines the number of lines in the matrix.

Must be an integer number. Default: "0".

NUMLIN_VISIBLE: Defines the minimum number of visible lines in the matrix. The remaining lines will be accessible only by using the scrollbar.

Must be an integer number. Default: "3".

ORIGIN: Allows setting which cell will be placed in the upper left corner of the matrix by automatically scrolling the visible area. To keep one of the coordinates of the cell in the upper left corner from being modified, use a value such as "L : *" or "* : C" (L and C>=1). Possible values:

Two numbers in the "L : C" format.

WIDTHDEF: Default column width.

Must be an integer number. Default: Width corresponding to 20 characters.

WIDTHn: Width of column n, where n is the number of the wished column (n>=0). If the width value is 0, the column will not be shown on the screen.

Must be an integer number. Default: Width defined in the WIDTHDEF attribute.

HEIGHTn: Height of column n, where n is the number of the wished column (n>=0). If

the height value is 0, the column will not be shown on the screen.

RESIZEMATRIX: Defines if the width of a column can be interactively changed. When this is possible, the user can change the size of a column by dragging the column title's upper corner. Possible values:

"YES" or "NO". Default: "NO" (does not allow interactive width change).

Mark Attributes

AREA: Defines if the area to be interactively marked by the user will be continuous or not. Possible values:

"CONTINUOUS" or "NOT_CONTINUOUS". Default: "CONTINUOUS".

MARK_MODE: Defines the entity that can be marked: none, lines, columns, lines and/or columns, and cells. Possible values:

"NO", "LIN", "COL", "LINCOL" or "CELL". Default: "NO" (no mark).

MARKED: Vector of "0" or "1" characters, informing which cells are marked (indicated by value "1"). The NULL value indicates there is no marked cell. The format of this character vector depends on the value of the MARK_MODE attribute: if its value is CELL, the vector will have NUMLIN × NUMCOL positions, corresponding to all the cells in the matrix. If its value is LIN, the vector will begin with letter "L" and will have further NUMLIN positions, each one corresponding to a line in the matrix. If its value is COL, the vector will begin with letter "C" and will have further NUMCOL positions, each one corresponding to a column in the matrix. If its value is LINCOL, the first letter, which can be either "L" or "C", will indicate which of the above formats is being used.

The values must be numbers in a vector of characters "0" and "1". Default: NULL.

MULTIPLE: Defines if more than one entity defined by MARK_MODE can be marked. Possible values:

"YES" or "NO". Default: "NO".

Action Attributes

ADDCOL: Adds a new column to the matrix after the number of the specified column. To insert a column at the top of the spreadsheet, value 0 must be used. To add more than one column, use format "C-C", where the first number corresponds to the base column and the second number corresponds to the number of columns to be added. It is valid only in normal mode.

The value must be a column number.

ADDLIN: Adds a new line to the matrix after the number of the specified line. To insert a line at the top of the spreadsheet, value 0 must be used. To add more than one line, use format "L-L", where the first number corresponds to the base line and the second number corresponds to the number of lines to be added. It is valid only in normal mode.

The value must be a line number.

DELCOL: Removes the given column from the matrix. To remove more than one column, use format "**C-C**", where the first number corresponds to the base column and the second number corresponds to the number of columns to be removed. It is valid only in normal mode.

The value must be a column number.

DELLIN: Removes the given line from the matrix. To remove more than one line, use format "**L-L**", where the first number corresponds to the base line and the second number corresponds to the number of lines to be removed. It is valid only in normal mode.

The value must be a line number.

EDIT_MODE: When set to YES, programatically puts the current cell in edition mode, allowing the user to modify its value. When consulted informs if the the current cell is being edited. Possible values:

"YES" or "NO".

REDRAW: The user can inform the matrix that the data has changed, and it must be redrawn. Values:

"ALL": Redraws the whole matrix.

"L%d": Redraws the given line (e. g.: "L3" redraws line 3)

"L%d : %d": Redraws the lines in the given region (e.g.: "L2 : 4" redraws lines 2, 3 and 4)

"C%d": Redraws the given column (e.g.: "C3" redraws column 3)

"C%d : %d": Redraws the columns in the given region (e.g: "C2 : 4" redraws columns 2, 3 and 4)

General Attributes

CURSOR: Default cursor used by the matrix. The default cursor is a symbol that looks like a cross. If you need to refer to this default cursor, use name "matrx_img_cur_excel".

FRAMECOLOR: Sets the color to be used in the matrix's frame lines.

[SCROLLBAR](#): Associates a horizontal and/or vertical scrollbar to the matrix. Is only effective if defined before the matrix is mapped. Default is YES.

[SIZE](#): Size of the matrix. Default: Minimum size which allows viewing the whole matrix.

[CARET](#): Allows specifying and verifying the caret's position when the matrix is in edition mode.

[SELECTION](#): Allows specifying and verifying selection interval when the matrix is in edition mode.

HIDEFOCUS: do not show the focus mark when drawing the matrix.

IupMatrix Callbacks

ACTION: Action generated when a keyboard event occurs.

```
int function(Ihandle *self, int c, int lin, int col, int active, char* a:
elem:action(c, lin, col, active, after: number) -> (ret: number) [in Iup]
```

self: Identifier of the matrix where the user typed something.

c: Identifier of the typed key. Please refer to the [Keyboard Codes](#) table for a list of possible values.

lin, col: Coordinates of the selected cell.

active: 1 if the cell is in edition mode, and 0 if it is not.

after: The new value of the text in case the key is validated (see return values).

Possible return values are: **IUP_DEFAULT** validates the key, **IUP_IGNORE** ignores the key, **IUP_CONTINUE** forwards the key to IUP's conventional processing. This function can also return the identifier of the key to be treated by the matrix.

BGCOLOR_CB - Action generated to retrieve the background color of a cell when it needs to be redrawn.

```
int function(Ihandle *self, int lin, int col, unsigned int *red, unsigned
elem:bgcolor_cb(lin, col: number) -> (ret, red, green, blue: number) [in
```

self: Identifier of the matrix where the user typed something.

lin, col: Coordinates of the cell.

red, green, blue: the cell background color.

If the function return **IUP_IGNORE**, the return values are ignored and the attribute defined background color will be used. If returns **IUP_DEFAULT** the returned values will be used as the background color.

CLICK_CB: Action generated when any mouse button is pressed over a cell. This callback is always called after other callbacks.

```
int function(Ihandle *self, int lin, int col, char *r); [in C]
elem:click(lin, col: number, r:string) -> (ret: number) [in IupLua3]
elem:click_cb(lin, col: number, r:string) -> (ret: number) [in IupLua5]
```

self: Identifier of the matrix interacting with the user.

lin, col: Coordinates of the cell where the mouse button was pressed.

They can be -1 if the user click outside the matrix but inside the canvas that contains it.

r: Status of the mouse buttons and some keyboard keys at the moment the event is generated. The following macros must be used for verification: `isshift(r)`, `iscontrol(r)`, `isbutton1(r)`, `isbutton2(r)`, `isbutton3(r)`, `isdoube(r)`. They return 1 if the respective key or button is pressed, or 0 otherwise.

DROPCHECK_CB: Action generated before the current cell is redrawn to determine if a dropdown feedback should be shown. If this action is not registered, no feedback will be shown.

```
int function(Ihandle *self, int lin, int col); [in C]
elem:dropcheck(lin, col: number) -> (ret: number) [in IupLua3]
elem:dropcheck_cb(lin, col: number) -> (ret: number) [in IupLua5]
```

self: Identifier of the matrix interacting with the user.
lin, col: Coordinates of the cell.

This function must return IUP_DEFAULT to show a dropdown field feedback, or IUP_IGNORE to ignore the dropdown feedback.

DROP_CB: Action generated before the current cell enters edition mode to determine if a text field or a dropdown will be shown. If this action is not registered, a text field will be shown. Its return determines what type of element will be used in the edition mode. If the selected type is a dropdown, the values appearing in the dropdown must be fulfilled in this callback, just like elements are added to any list (the drop parameter is the handle of the dropdown list to be shown). You should also set the list's current value ("VALUE"), the default is always "1". The previously cell value can be verified from the given drop Ihandle via the "PREVIOUSVALUE" attribute.

```
int function(Ihandle *self, Ihandle *drop, int lin, int col); [in C]
elem:drop(drop: iuplua_tag, lin, col: number) -> (ret: number) [in IupLua]
elem:drop_cb(drop: iuplua_tag, lin, col: number) -> (ret: number) [in IupLua5]
```

self: Identifier of the matrix interacting with the user.
drop: Identifier of the dropdown list which will be shown to the user.
lin, col: Coordinates of the current cell.

This function must return IUP_IGNORE to show a text-edition field, or IUP_DEFAULT to show a dropdown field.

DROPSELECT_CB: Action generated when an element in the dropdown list is selected.

```
int function(Ihandle *self, int lin, int col, Ihandle *drop, char *t, int i); [in C]
elem:dropselect(lin, col: number, drop: iuplua_tag, t: string, i, v: number) [in IupLua]
elem:dropselect_cb(lin, col: number, drop: iuplua_tag, t: string, i, v: number) [in IupLua5]
```

self: Identifier of the matrix interacting with the user.
lin, col: Coordinates of the current cell.
drop: Identifier of the dropdown list shown to the user.
t: Text of the item whose state was changed.
i: Number of the item whose state was changed.
v: Indicates if item was selected or unselected (0 or 1).

EDITION_CB: Action generated when the current cell enters or leaves the edition mode.

```
int function(Ihandle *self, int lin, int col, int modo); [in C]
elem:edition(lin, col, mode: number) -> (ret: number) [in IupLua]
```

self: Identifier of the matrix interacting with the user.
lin, col: Coordinates of the current cell.

mode: 1 if the cell has entered the edition mode, or 0 if the cell has left the edition mode.

The user must return IUP_IGNORE if he/she wants to prevent the field from being editable, or IUP_DEFAULT otherwise.

ENTERITEM_CB: Action generated when a matrix cell is selected, becoming the current cell.

```
int function(Ihandle *self, int lin, int col); [in C]
elem:enteritem(lin, col: number) -> (ret: number) [in IupLua3]
elem:enteritem_cb(lin, col: number) -> (ret: number) [in IupLua5]
```

self: Identifier of the matrix interacting with the user.

lin, col: Coordinates of the selected cell.

The user must return IUP_DEFAULT.

FGCOLOR_CB - Action generated to retrieve the foreground color of a cell when it needs to be redrawn.

```
int function(Ihandle *self, int lin, int col, unsigned int *red, unsigned int *green, unsigned int *blue); [in C]
elem:bgcolor_cb(lin, col: number) -> (ret, red, green, blue: number) [in IupLua5]
```

self: Identifier of the matrix where the user typed something.

lin, col: Coordinates of the cell.

red, green, blue: the cell foreground color.

If the function return IUP_IGNORE, the return values are ignored and the attribute defined foreground color will be used. If returns IUP_DEFAULT the returned values will be used as the foreground color.

LEAVEITEM_CB: Action generated when a cell is no longer the current cell.

```
int function(Ihandle *self, int lin, int col); [in C]
elem:leaveitem(lin, col: number) -> (ret: number) [in IupLua3]
elem:leaveitem_cb(lin, col: number) -> (ret: number) [in IupLua5]
```

self: Identifier of the matrix interacting with the user.

lin, col: Coordinates of the cell which is no longer the current cell.

The user must return either IUP_DEFAULT or IUP_IGNORE. Returning IUP_IGNORE prevents the current cell from changing (this will not work when the focus is leaving the matrix.)

MOUSEMOVE_CB: Action generated to notify the application that the mouse has moved over the matrix.

```
int function(Ihandle *self, int lin, int col); [in C]
elem:mousemove(lin, col: number) -> (ret: number) [in IupLua3]
elem:mousemove_cb(lin, col: number) -> (ret: number) [in IupLua5]
```

self: Identifier of the matrix interacting with the user.

lin, col: Coordinates of the cell that the mouse cursor is currently on.

SCROLL_CB: Action generated when the matrix' visualization surface is changed. Can be used together with the "ORIGIN" attribute to synchronize the movement of two or more matrices.

```
int function(Ihandle *self, int lin, int col); [in C]
elem:scroll(lin, col: number) -> (ret: number) [in IupLua3]
elem:scroll_cb(lin, col: number) -> (ret: number) [in IupLua5]
```

self: Identifier of the matrix interacting with the user.

lin, col: Coordinates of the cell currently in the upper left corner of the matrix.

The user must return IUP_DEFAULT.

VALUE_CB: Action generated to verify the value of a cell in the matrix when it needs to be redrawn. Called both for common cells and for line and column titles.

```
char* function(Ihandle* self, int lin, int col); [in C]
elem:valuecb(lin, col: number) -> (ret: string) [in IupLua3]
elem:value_cb(lin, col: number) -> (ret: string) [in IupLua5]
```

self: Identifier of the matrix interacting with the user.

lin, col: Coordinates of the cell currently in the upper left corner of the matrix.

Must return the string to be redrawn. The existence of this callback defines the callback operation mode of the matrix.

VALUE_EDIT_CB: Action generated to notify the application that the value of a cell was changed. Since it is a notification, it cannot refuse the value modification (which can be done by the "EDITION_CB" callback).

```
int function(Ihandle *self, int lin, int col, char* newval); [in C]
elem:value_edit(lin, col, newval: string) -> (ret: number) [in IupLua3]
elem:value_edit_cb(lin, col, newval: string) -> (ret: number) [in IupLua5]
```

self: Identifier of the matrix interacting with the user.

lin, col: Coordinates of the cell currently in the upper left corner of the matrix.

newval: String containing the new cell value

The user must return IUP_DEFAULT.

The canvas callbacks [ACTION](#), [SCROLL_CB](#), [KEYPRESS_CB](#), [GETFOCUS_CB](#), [KILLFOCUS_CB](#) if set will be called before the internal callbacks. The IupGetAttribute always returns the internal callbacks.

The canvas callbacks [MOTION_CB](#), [MAP_CB](#), [RESIZE_CB](#) and [BUTTON_CB](#) can not be changed. The other callbacks can be freely changed.

The GETFOCUS/KILLFOCUS callbacks are always called before other callbacks.

See [IupCanvas](#).

IupTree

Creates a tree containing nodes of branches or leaves. It inherits from [IupCanvas](#).

The branches can be expanded or collapsed. When a branch is expanded, its immediate children are visible, and when it is collapsed they are hidden. The leaves can generate an executed or renamed action, branches can only generate renamed actions. Both branches and leaves can have an associated text. The selected node is the node with the focus rectangle, marked nodes have their background inverted.

Parameters/Return

```
Ihandle* IupTree(void); [in C]
iuptree{} -> (elem: iuplua_tag) [in IupLua3]
iup.tree{} -> (elem: iuplua_tag) [in IupLua5]
tree() [in LED]
```

This function returns the identifier of the created IupTree, or NULL if an error occurs.

Attributes

[General](#)

SCROLLBAR
FONT
ADDEXPANDED

[Marks](#)

CTRL
SHIFT
STARTING
VALUE
MARKED

[Images](#)

IMAGELEAF
IMAGEBRANCHCOLLAPSED
IMAGEBRANCHEXPANDED
IMAGEid
IMAGEEXPANDEDid

[Nodes](#)

NAME
STATE
DEPTH
KIND
PARENT
COLOR

Action

ADDLEAF
ADDBRANCH
DELNODE
REDRAW

Callbacks

SELECTION_CB: Action generated when an node is selected or deselected.
BRANCHOPEN_CB: Action generated when a branch is expanded.
BRANCHCLOSE_CB: Action generated when a branch is collapsed.
EXECUTELEAF_CB: Action generated when a leaf is to be executed.
RENAMENODE_CB: Action generated when a node is to be renamed.
RIGHTCLICK_CB: Action generated when the right mouse button is pressed over a node.

Notes

Branches may be added in IupLua using a Lua Table (see Example 2).

Hierarchy

Branches can contain other branches or leaves. The tree always has at least one branch, the **root**, which will be the parent of all the first level branches and leaves.

Structure

The IupTree is stored as a list, so that each node or branch has an associated identification number (`id`), starting by the root, with `id=0`. However, this number does not always correspond to the same node as the tree is modified. For example, a node with `id 2` will always refer to the third node in the tree. For that reason, there is also `userid`, which allows identifying a specific node. The `userid` always refers to the same node (just as the associated text). The `userid` may contain a user-created structure allowing the identification of a node.

Each node also contains its depth level, starting by the root, which has depth 0. To allow inserting nodes in any position, sometimes the depth of a node must be explicitly changed. For instance, if you create a leaf in a child branch of the root, it will be created with depth 2. To make it become a child of the root, its depth must be set to 1.

Images

IupTree has three types of images: one associated to the leaf, one to the collapsed branch and the other to the expanded branch. Each image can be changed, both globally and individually.

The predefined images used in IupTree can be obtained by means of function `IupGetHandle`. The names of the predefined images are: `IMGLEAF`, `IMGCOLLAPSED`, `IMGEXPANDED`, `IMGBLANK` (blank sheet of paper) and `IMGPAPER` (written sheet of paper).

Scrollbar

IupTree's scrollbar is activated by default and works automatically. When a node leaves the visible area, the scrollbar automatically scrolls so as to make it visible. We recommend not changing the SCROLLBAR attribute.

Fonts

The fonts used by IupTree are like the ones defined by IUP (see attribute FONT). We recommend using only IUP-defined fonts.

Manipulation

Node insertion or removal is done by means of attributes. It is allowed to remove nodes and branches inside callbacks associated to opening or closing branches.

This means that the user may insert nodes and branches only when necessary when the parent branch is opened, allowing the use of a larger IupTree without too much overhead. Then when the parent branch is closed the subtree can be removed. A side-effect of this use is that the expanded or collapsed state of the children branches must be managed by the user.

When a node is added, removed or renamed the tree is not automatically redrawn. You must set REDRAW=YES when you finish changing the tree.

Simple Marking

Is the IupTree's default operation mode. In this mode only one node is marked, and it matches the selected node.

Multiple Marking

IupTree allows marking several nodes simultaneously using the Shift and Control keys. To use multiple marking, the user must use attributes SHIFT and CTRL.

When a user keeps the Control key pressed, the individual marking mode is used. This way, the selected node can be modified without changing the marked node. To reverse a node marking, the user simply has to press the space bar.

When the user keeps the Shift key pressed, the block marking mode is used. This way, all nodes between the selected node and the initial node are marked, and all others are unmarked. The initial node is changed every time a node is marked without the Shift key being pressed. This happens when any movement is done without Shift or Control being pressed, or when the space bar is pressed together with Control.

Removing a Node with "Del"

You can simply implement a K_ANY or KEYPRESS_CB and do:

```
int k_any(Ihandle* self, int c)
{
    if (c == K_DEL)
    {
        IupSetAttribute(self, "DELNODE", "MARKED");
        IupSetAttribute(self, "REDRAW", "");
    }
    return IUP_DEFAULT;
}
```

}

Navigation

Using the keyboard:

- **Arrow Up/Down:** Shifts the selected node to the neighbor node, according to the arrow direction.
- **Arrow Left/Right:** Makes the branch collapse/expand
- **Home/End:** Selects the root/last node.
- **Page Up/Page Down:** Selects the node one page above/below the selected node.
- **Enter:** If the selected node is an expanded branch, it is collapsed; if it is a collapsed branch, it is expanded; if it is a leaf, it is executed.
- **Space:** If the Control key is pressed marks or unmarks a node, if not calls the rename callback.

Using the mouse:

- **Clicking a node:** Selects the clicked node.
- **Clicking a (-/+) box:** Makes the branch to the right of the (-/+) box collapse/expand.
- **Clicking an empty region:** Unmarks all nodes (including the selected one).
- **Double-clicking a node image:** If the selected node is an expanded branch, it is collapsed; if it is a collapsed branch, it is expanded; if it is a leaf, it is executed.
- **Double-clicking a node text:** Calls the rename callback.

Extra Functions

IupTree has functions that allow associating a pointer (or a user defined id) to a node. In order to do that, you provide the id of the node and the pointer (userid); even if the node's id changes later on, the userid will still be associated with the given node. In IupLua, instead of a pointer the same functions are defined for tables.

```
int IupTreeSetUserId(Ihandle *self, int id, void *userid); [in C]
IupTreeSetUserId(self: handle, id: number, userid: userdata); [in IupLua]
iup.TreeSetUserId(self: handle, id: number, userid: userdata); [in IupLua]
```

self: Identifier of the IupTree interacting with the user.

id: Node identifier.

userid: User pointer associated to the node. Use NULL value to free reference.

Note: This function needs to be called again freeing the node from the userdata or it will never be garbage collected.

```
void* IupTreeGetUserId(Ihandle *self, int id); [in C]
IupTreeGetUserId(self: handle, id: number) -> (ret: userdata) [in IupLua]
iup.TreeGetUserId(self: handle, id: number) -> (ret: userdata) [in IupLua]
```

self: Identifier of the IupTree interacting with the user.

id: Node identifier.

Returns the pointer associated to the node.

```
int IupTreeGetId(Ihandle *self, void *userid); [in C]
IupTreeGetId(self: handle, userid: userdata) -> (ret: number) [in IupLua]
```

```
iup.TreeGetId(self: handle, userid: userdata) -> (ret: number) [in IupLua3]
```

self: Identifier of the IupTree interacting with the user.

userid: Pointer associated to the node.

Returns the id of the node on success and -1 on failure.

```
IupTreeSetTableId(self: handle, id: number, table: table) [in IupLua3]
```

```
iup.TreeSetTableId(self: handle, id: number, table: table) [in IupLua5]
```

self: Identifier of the IupTree interacting with the user.

id: Node identifier.

table: Table that should be associated to the node or leaf. Use nil value to free reference.

Notes: This function needs to be called again freeing the node from the table or the table will never be garbage collected. Also, the user should not use the same table to reference different nodes (neither in the same nor across different trees.)

```
iup.TreeGetTableId(self: handle, table: table) -> (ret: number) [in IupLua3]
```

```
iup.TreeGetTableId(self: handle, table: table) -> (ret: number) [in IupLua5]
```

self: Identifier of the IupTree interacting with the user.

table: Table that should be associated to the node or leaf.

Returns the **id** of the node on success and nil otherwise.

```
iup.TreeGetTable(self: handle, id: number) -> (ret: table) [in IupLua3]
```

```
iup.TreeGetTable(self: handle, id: number) -> (ret: table) [in IupLua5]
```

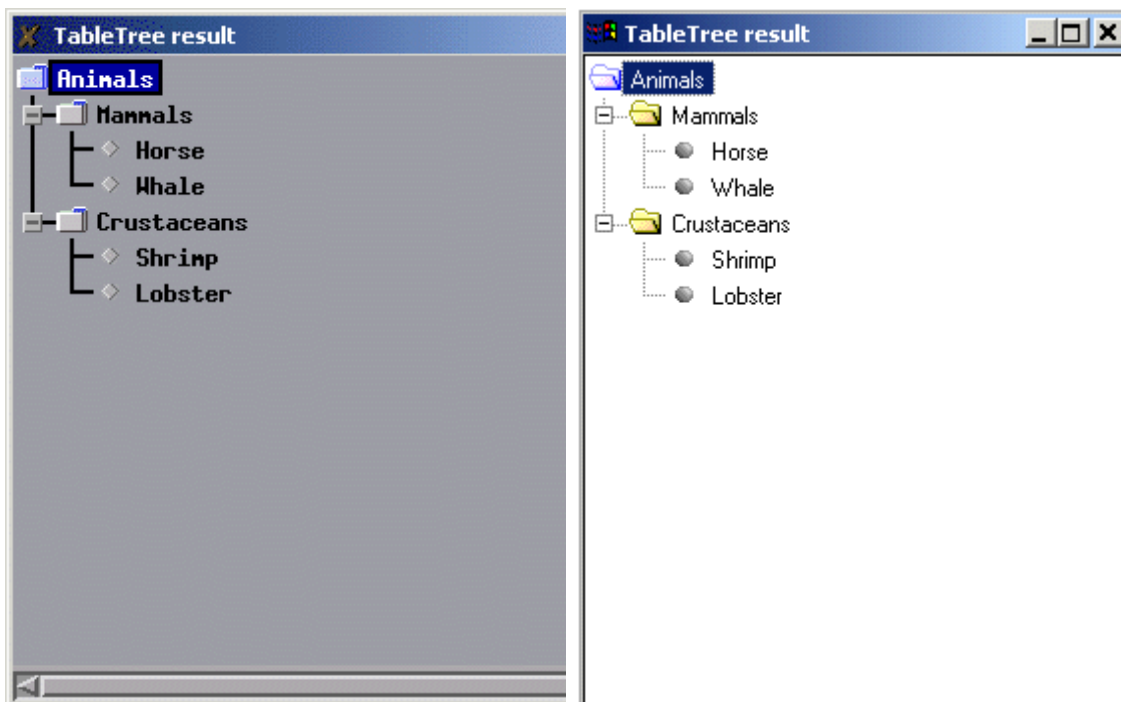
self: Identifier of the IupTree interacting with the user.

id: Node identifier.

Returns the **table** of the node on success and nil otherwise.

Examples

Creates a IupTree with the values shown on the images below, and allows the user to change them dynamically.



See Also

[IupCanvas](#)

IupTree Attributes

General

SCROLLBAR: Associates a horizontal and/or vertical scrollbar to the canvas. Default: "YES" .

FONT: Character font of the text displayed on the element.

ADDEXPANDED: Defines if branches will be expanded when created. The root node is always expanded when created. Possible values:

"YES": The branches will be created expanded

"NO": The branches will be created collapsed

Default: "NO" .

Marks

VALUE: The selected node. When changed also marks the node, but only if the Control and Shift keys are not used. Possible values:

The node identifier to be selected.

When changed also accepts the values:

"ROOT": the root node

"LAST": the last node

"PGUP": the node one page below
 "PGDN": the node one page above
 "NEXT": the node following the selected node. If the selected node is the last one, the last one will be used instead
 "PREVIOUS": the previous node of the selected node. If the selected node is the root, the root will be used instead

The following values are also accepted but they are independent from the state of the Control and Shift keys, and from the CTRL and SHIFT attributes. And the selected node is not changed. These values are kept here for backward compatibility, but they would fit better in the MARKED attribute.

"INVERT": Inverts the node's marking. Using the "INVERTid" form, where id is the node identifier, it is possible to invert the marking of any node.
 "BLOCK": Marks all nodes between the selected node and the initial block-marking node (see Navigation / Multiple Marking)
 "CLEARALL": Unmarks all nodes
 "MARKALL": Marks all nodes
 "INVERTALL": Inverts the marking of all nodes

MARKED: The marking state of the selected node. Using the **MARKEDid** form, where id is the node identifier, it is possible to retrieve or change the marking state of any node. Possible values:

"YES": The node is marked
 "NO": The node is not marked

Returns NULL if the node's id is invalid.

CTRL: Activates or deactivates the Control key function. Possible values:

"YES": Control key activated; allows marking individual nodes
 "NO": Control key deactivated; does not allow marking individual nodes

Default: "NO" .

SHIFT: Activates or deactivates the Shift key function. Possible values:

"YES": Shift key activated; allows marking adjacent nodes
 "NO": Shift key deactivated; does not allow marking adjacent nodes

Default: "NO" .

STARTING: Defines the initial node for the block marking.

The value must be the node identifier.

Default: root node.

Images

IMAGELEAF: Defines the image that will be shown for all leaves . Must be a 16x16 image.

IUP name of the image (see IupImage)

Default: "IMGLEAF" .

IMAGEBRANCHCOLLAPSED: Defines the image that will be shown for all collapsed branches. Must be a 16x16 image.

IUP name of the image (see IupImage)

Default: "IMGCOLLAPSED" .

IMAGEBRANCHEXPANDED: Defines the image that will be shown for all expanded branches. Must be a 16x16 image.

IUP name of the image (see IupImage)

Default: "IMGEXPANDED" .

IMAGEid: Defines the image that will be shown on a specific node. Valid for leaves and for collapsed branches. This attribute must always be used with the id number. This attribute can only be set. Ex. "IMAGE2".

IUP name of the image (see IupImage)

Default: NULL .

IMAGEEXPANDEDid: Defines the image that will be shown on a specific node. It has no effect over leaves and is valid for expanded branches. This attribute must always be used with the id number. This attribute can only be set. Ex. "IMAGEEXPANDED3".

IUP name of the image (see IupImage)

Default: NULL .

Nodes

NAME: Changes or retrieves the name of the selected node. Using the "**NAMEid**" form, where **id** is the node identifier, it is possible to change the name of any node.

The value must be a node name.

STATE: Changes or retrieves the state of the selected branch. Using the "**STATEid**" form, where **id** is the node identifier, it is possible to change the state of any branch. This attribute only works on branches. If it is used on a leaf, nothing will happen. Possible values:

"EXPANDED": Expanded branch state (shows its children)

"COLLAPSED": Collapsed branch state (hides its children)

DEPTH: If set, it defines the node's depth. Does not verify if the resulting tree is valid. If retrieved, it returns the node's depth. Using the "**DEPTHid**" form, where *id* is the node identifier, it is possible to refer to any node.

The value must be the node's depth

KIND: Returns the kind of the selected node. Using the "**KINDid**" form, where *id* is the node identifier, it is possible to retrieve the kind of any node. This attribute can only be retrieved. Possible values:

"LEAF": The node is a leaf

"BRANCH": The node is a branch

PARENT: Returns the identifier of the selected node's parent. Using the "**PARENTid**" form, where *id* is the node identifier, it is possible to retrieve the identifier of any node. This attribute can only be retrieved.

COLOR: Color of the provided node. Using the form "**COLORid**", where *id* is the node identifier, it is possible to set or retrieve the color of any node. The value should be a string in the format "R G B" where R, G, B are numbers from 0 to 255.

Action

ADDLEAF: Adds a new leaf after the selected node. The id of the new leaf will be the id of the selected node + 1. The selected node is marked and all others unmarked. The selected node position remains the same. Using the "**ADDLEAFid**" form, where *id* is the node identifier, it is possible to insert a leaf after any node. In this case, the id of the inserted node will be *id* + 1. If the specified node does not exist, nothing happens. This attribute can only be set.

The value must be a leaf name.

ADDBRANCH: Adds a new branch after the selected node. The id of the new branch will be the id of the selected node + 1. The selected node is marked and all others unmarked. The selected node position remains the same. Using the "**ADDBRANCHid**" form, where *id* is the node identifier, it is possible to insert a branch after any node. In this case, the id of the inserted node will be *id* + 1. By default, all branches created are expanded. If the specified node does not exist, nothing happens. This attribute can only be set.

The value must be a branch name.

DELNODE: Removes the marked node (or its children). Using the "**DELNODEid**" form, where *id* is the node identifier, it is possible to remove any node. The root cannot be removed. If the specified node does not exist, nothing happens. This attribute can only be set. Possible values:

"MARKED": Deletes all marked nodes (and all their children)

"SELECTED": Deletes only the selected node (and its children)

"CHILDREN": Deletes only the children of the selected node

Returns the identifier of the marked node's parent.

REDRAW: Forces an immediate redraw. It is necessary to force a redraw whenever the

user adds or removes a node or a branch. The value is ignored.

IupTree Callbacks

SELECTION_CB: Action generated when an element is selected or deselected. This action occurs when the user clicks with the mouse or uses the keyboard with the appropriate combination of keys.

```
int function(Ihandle *self, int id, int status) [in C]
elem:selection(id, status: number) -> (ret: number) [in IupLua3]
elem:selection_cb(id, status: number) -> (ret: number) [in IupLua5]
```

self: Identifier of the IupTree interacting with the user.

id: Node identifier.

status: 1 - node was selected, 0 - node was unselected.

This function must return IUP_IGNORE for the selected node not to be changed, or IUP_DEFAULT to change it.

BRANCHOPEN_CB: Action generated when a branch is expanded. This action occurs when the user clicks the "+" sign on the left of the branch, or when double clicks the branch image, or hits Enter on a collapsed branch.

```
int function(Ihandle *self, int id) [in C]
elem:branchopen(id: number) -> (ret: number) [in IupLua3]
elem:branchopen_cb(id: number) -> (ret: number) [in IupLua5]
```

self: Identifier of the IupTree interacting with the user.

id: Node identifier.

This function must return IUP_IGNORE for the branch not to be opened, or IUP_DEFAULT for the branch to be opened.

BRANCHCLOSE_CB: Action generated when a branch is collapsed. This action occurs when the user clicks the "-" sign on the left of the branch, or when double clicks the branch image, or hits Enter on an expanded branch.

```
int function(Ihandle *self, int id); [in C]
elem:branchclose(id: number) -> (ret: number) [in IupLua3]
elem:branchclose_cb(id: number) -> (ret: number) [in IupLua5]
```

self: Identifier of the IupTree interacting with the user.

id: Identifier of the clicked node.

This function must return IUP_IGNORE for the branch not to be closed, or IUP_DEFAULT for the branch to be closed.

EXECUTELEAF_CB: Action generated when a leaf is to be executed. This action occurs when the user double clicks the leaf image, or hits Enter on a leaf.

```
int function(Ihandle *self, int id); [in C]
elem:executeleaf(id: number) -> (ret: number) [in IupLua3]
elem:executeleaf_cb(id: number) -> (ret: number) [in IupLua5]
```

self: Identifier of the IupTree interacting with the user.

id: Identifier of the clicked node.

RENAMENODE_CB: Action generated when a node is to be renamed. It occurs only when the user double clicks the **text** associated to a node (leaf or branch).

```
int function(Ihandle *self, int id, char *name); [in C]
elem:renamenode(id: number, name: string) -> (ret: number) [in IupLua3]
elem:renamenode_cb(id: number, name: string) -> (ret: number) [in IupLua!]
```

self: Identifier of the IupTree interacting with the user.

id: Identifier of the clicked node.

name: Name of the clicked node.

RIGHTCLICK_CB: Action generated when the right mouse button is pressed over the IupTree.

```
int function(Ihandle *self, int id); [in C]
elem:rightclick(id: number) -> (ret: number) [in IupLua3]
elem:rightclick_cb(id: number) -> (ret: number) [in IupLua5]
```

self: Identifier of the IupTree interacting with the user.

id: Identifier of the clicked node.

The canvas callback [K_ANY](#) if set will be called before the internal callback. The IupGetAttribute always returns the internal callback.

The canvas callbacks [ACTION](#), [SCROLL_CB](#), [MAP_CB](#), [RESIZE_CB](#) and [BUTTON_CB](#) can not be changed. The other callbacks can be freely changed.

See [IupCanvas](#).

Others

The following controls and utilities are offered in separate libraries:.

- [IupGLCanvas](#)

Creates an OpenGL canvas.

- [IupLoadImage](#)

Creates an image from file using the IM library.

IupGLCanvas

Creates an OpenGL canvas (drawing area for OpenGL). It inherits from [IupCanvas](#).

Initialization and Usage

The **IupGLCanvasOpen** function must be called after a **IupOpen**, so that the control can be used. The `iupgl.h` file must also be included in the source code. The program must be linked to the control's library (`iupgl.lib` on Windows and `libiupgl.a` on Unix), and with the OpenGL library.

To make the control available in Lua, use the initialization function in C, **gllua_open**, after calling **iuplua_open**. The `luagl.h` file must also be included in the source code. The program must be linked to the control's libraries (`luagl.lib` on Windows and `libluagl.a` on Unix).

You will need also to link with the OpenGL libraries. In Windows add: `opengl32.lib` and optionally `glu32.lib` or `glaux.lib`. In Motif add before the Motif libraries: `-LGLw -LGLU -LGL` (in Linux the GLw library is not available in the system so we include it inside the `iupgl` library).

Parameters/Return

```
Ihandle* IupGLCanvas(char* action); [in C]
iupglcanvas{} -> (elem: iuplua_tag) [in IupLua3]
iup.glcanvas{} -> (elem: iuplua_tag) [in IupLua5]
glcanvas(action) [in LED]
```

action: Name of the action generated when the canvas needs to be redrawn.

This function returns the identifier of the created canvas, or NULL if an error occurs.

Attributes

The `IupGLCanvas` element understands all attributes defined for a conventional canvas, see [IupCanvas](#).

Apart from these attributes, `IupGLCanvas` understands specific attributes used to define the kind of buffer to be instanced. Such attributes must be set before the element is mapped on the screen. After the mapping, specifying these special attributes has no effect.

The specific `IupGLCanvas` attributes are:

BUFFER: Indicates if the buffer will be single "SINGLE" or double "DOUBLE". Default is "SINGLE".

COLOR: Indicates the color model to be adopted: "INDEX" or "RGBA". Default is "RGBA".

BUFFER_SIZE: Indicates the number of bits for representing the color indices (valid only for INDEX). Default is "8" (256-color palette).

RED_SIZE, **GREEN_SIZE** and **BLUE_SIZE**: Indicate the number of bits for representing each color component (valid only for RGBA). Default is "8" for each component (True Color support).

ALPHA_SIZE: Indicates the number of bits for representing each color's alpha component (valid only for RGBA and for hardware that store the alpha component).

Default is "0".

DEPTH_SIZE: Indicates the number of bits for representing the z coordinate in the z -buffer. Value "0" means the z -buffer is not necessary.

STENCIL_SIZE: Indicates the number of bits in the stencil buffer. Value "0" means the stencil buffer is not necessary. Default is "0".

ACCUM_RED_SIZE, **ACCUM_GREEN_SIZE**, **ACCUM_BLUE_SIZE** and **ACCUM_ALPHA_SIZE**: Indicate the number of bits for representing the color components in the accumulation buffer. Value "0" means the accumulation buffer is not necessary. Default is "0".

STEREO: Creates a stereo GL canvas (special glasses are required to visualize it correctly). Possible values: "YES" or "NO". Default: "NO".

"ERROR": If an error is found, returns a string containing a description of it.

"CONTEXT", **"VISUAL"** and **"COLORMAP"**: [Motif Only] Returns "GLXContext", "XVisualInfo*" and "Colormap".

"PRINTINFO": [Motif Only] If "1" during the canvas initialization some information will be printed on `stderr`.

Callbacks

The `IupGLCanvas` element understands all callbacks defined for a conventional canvas, see [IupCanvas](#).

Additionally:

[RESIZE_CB](#): By default the resize callback sets:

```
glViewport(0,0,width,height);
```

Auxiliary Functions

```
void IupGLMakeCurrent(Ihandle* self);
```

Activates the "self" canvas. All subsequent OpenGL commands are directed to such canvas.

```
void IupGLSwapBuffers(Ihandle* self);
```

Makes the "BACK" buffer visible. This function is necessary when a double buffer is used.

```
void IupGLPalette(Ihandle* self, int index, float r, float g, float b);
```

Defines a color in the color palette. This function is necessary when INDEX color is used.

Comments

An OpenGL canvas when put inside an IupFrame will not work.

[Examples](#)

See Also

[IupCanvas](#)

IupLoadImage

A function that creates an IupImage from file using the IM library. The function can load the formats: BMP, JPEG, GIF, TIFF, PNG, PNM, PCX, ICO and others. For more information about the IM library see <http://www.tecgraf.puc-rio.br/im>.

Initialization and Usage

To generate an application that uses this function, the program must be linked to the function's library (`iupim.lib` on Windows and `libiupim.a` on Unix). The `iupim.h` file must also be included in the source code.

To make the function available in Lua, use the initialization function in C, **iupluaaim_open**, after calling **iuplua_open**. The `iupluaaim.h` file must also be included in the source code. The program must be linked to the functions's libraries (`iupluaaim.lib` on Windows and `libiupluaaim.a` on Unix).

Parameters/Return

```
Ihandle* IupLoadImage(const char* file_name); [in C]
IupLoadImage{file_name: string} -> (elem: iuplua_tag) [in IupLua3]
iup.LoadImage{file_name: string} -> (elem: iuplua_tag) [in IupLua5]
```

file_name: Name of the file to be loaded.

This function returns the identifier of the created image, or NULL if an error occurs.

See Also

[IupImage](#)

Third-party

- [IupSpeech](#)

Creates a speech engine that allows speech recognition and speech. Uses Microsoft Speech SDK 5.1.

- [Color Bar](#) (Portuguese)

The extended control **Colorbar** was developed with the purpose of aiding [IUP](#) / [CD](#) applications which need to work with a color palette, allowing a selection of up to two

colors.

- [Joystick](#) (Portuguese) - OLD

Allows the use use of joystick (Windows only).

- [Play Video](#) (Portuguese) - OLD

Allows the user to play videos using IUP (Windows only).

- [Capture Video](#) (Portuguese) - OLD

Allows the user to capture frames from cameras, VCRs and TVs (Windows only).

- [IupDB](#) (Portuguese) - DISCONTINUED

Associates controls to database fields.

Attributes

Attributes are used to add or remove characteristics to/from elements. Each element has a set of attributes that affect it, and each attribute can work differently for each element. Depending on the element, its attribute's value can be computed or simply verified; it can be internally stored or not.

If an element does not have a given attribute defined, this attribute will be inherited from its parent. Only a few attributes are not inherited: `IUP_TITLE`, `IUP_VALUE`, `IUP_ALIGNMENT`, `IUP_X`, `IUP_Y`, `IUP_RASTERSIZE` and `IUP_SIZE`.

For further information on attributes, see [Guide / Attributes](#).

ATTENTION: Not all attributes are listed here. Some are described only in the documentation of each control.

All

Attributes that affect all elements, or all of the elements with user interaction, or most of them.

Image

Attributes that affect element `IupImage`.

Item/SubItem

Affects menu-composition elements `Item` and `SubItem`.

List

Attributes that change the functioning of element `IupList`.

Vbox/Hbox/Zbox/Frame

Attributes that affect dialog-composition elements.

Button/Toggle/Label

Attributes that affect buttons, toggles and labels, which are basic elements for constructing the dialog.

Text

Attributes that affect element `IupText`, which allows the user to enter data to the interface.

Multiline

Attributes that affect the `IupMultiline` element, which is actually an extension of `IupText`, differing only by the number of lines.

Text/Multiline

Attributes that affect both the `IupText` and `IupMultiline` elements.

Dialog/Text/Multi./Canvas**Dialog****Canvas/Dialog****Canvas****Canvas/Scrollbar**

Attributes for each of such elements.

Global

Element-independent attributes.

Tables

Value tables accepted by certain special attributes.

ACTIVE

Activates or inhibits user interaction.

Value

"YES" (active), "NO" (inactive).

Default: "YES".

Affects

All.

VISIBLE

Shows or hides the element.

Value

"YES" (visible), "NO" (hidden).

Default: "YES"

Note

Returns NULL if the element has not yet been mapped.

Affects

All except [IupItem](#) and [IupSeparator](#).

BGCOLOR

Element's background color.

Value

The RGB components. Values should be between 0 and 255, separated by a blank space.

Default: Depends on the native interface system.

Affects

All.

See Also

[FGCOLOR](#)

FGCOLOR

Element's foreground color. Usually it is the color of the associated text.

Value

The RGB components. Values should be between 0 and 255, separated by a blank space.

Default: Depends on the native interface system.

Affects

All.

See Also

[BGCOLOR](#)

FONT

Character font of the text shown in the element.

Value

Font name. Please refer to the [Character Fonts](#) table for a list of the fonts existing in IUP drivers.

Default: Depends on the native interface system.

Affects

All elements with an associated text.

Note

To set a font, the user can use one of the font options provided in the [Character Fonts](#) table, or directly use the name of a native font in the driver. Attention: when consulting this attribute, the user will always be returned the name of the driver font being used, not the name of the IUP font. To get the name of the IUP font, the user must use the [IupUnMapFont](#) function.

See Also

[TITLE](#), [IupMapFont](#), [IupUnMapFont](#).

EXPAND

Makes the size of an element dynamic. It expands or retracts, fulfilling empty spaces inside a dialog.

Value

"YES" (both directions), "HORIZONTAL", "VERTICAL" or "NO".

Default: Depends on the element. When not specified otherwise, the default value is "NO".

Affects

All that have a visual representation. Does not apply to `radio`, `zbox`, `vbox`, `hbox`.

X

Control's absolute horizontal position on the screen in pixels (relative to the upper left corner.) This attribute can only be consulted.

Value

Integer number.

Affects

All controls that have visual representation.

Y

Control's absolute vertical position on the screen in pixels (relative to the upper left corner.) This attribute can only be consulted.

Value

Integer number.

Affects

All controls that have visual representation.

SIZE

Size of the element in units proportional to the size of a character.

Value

"*widthxheight*", where *width* and *height* are integer values corresponding to the horizontal and vertical size, respectively, in characters. The element may have only one dimension which is applicable to be modified - for instance, [IupText](#), which has only width. In this case, the second parameter is ignored and does not need to be passed. You can also change only one of the parameters by removing the other one and maintaining "x". For example: "x40" (height only) or "40x" (width only). The other size will be chosen by IUP depending on the composition elements and on the EXPAND attribute.

Default: Depends on the element and on the element's EXPAND attribute.

Notes

The size observes the following heuristics:

- Width in 1/4's of the average width of a character.
- Height in 1/8's of the average height of a character.

When this attribute is changed, the [RASTERSIZE](#) attribute is automatically updated.

When this attribute is changed by means of a call to function `IupSetAttribute` or `IupStoreAttribute`, the size will be the minimum size for the element. If you wish to use this size only as an initial size, change this attribute to `NULL` after viewing the dialog.

Affects

All.

See Also

[EXPAND](#), [RASTERSIZE](#)

WID

Element identifier in the native interface system.

Value

Depends on the platform.

Note

Verification-only attribute.

Affects

All.

TIP

Summarized text, usually just a word, identifying the element's functionality. The text will be shown when the mouse lies over the element.

Value

Text.

Default: `NULL`.

Note

Background and foreground colors, and the font used, are predetermined and depend on the native system.

Affects

All except `label`, `menu item` and `submenu item`.

RASTERSIZE

Specifies the element's size in pixels.

Value

"*width*x*height*", where *width* and *height* are integer values corresponding to the horizontal and vertical size, respectively, in pixels.

Default: Depends on the system and on the EXPAND attribute.

Affects

All.

Note

When this attribute is changed, the [SIZE](#) attribute is automatically updated. Please refer to the notes on the [SIZE](#) attribute for further detail.

See Also

[SIZE](#)

TITLE

Element's title. It is often used to modify some static text of the element (which cannot be changed by the user).

Value

Text.

Default: ""

Affects

All elements with an associated text.

See Also

[FONT](#)

VALUE

Affects several elements differently - that is, its behavior is element dependent. It is often used to change the control's main value, such as the text of a [IupText](#).

For the [IupRadio](#) and [IupZbox](#), elements, which are categorized as composition elements, this attribute represents the element "selected" among the others in the designed composition. To change this attribute in such cases, different mechanisms are necessary according to the programming environment used. When the elements taking part in a composition were created in C, this attribute's contents is a name that must be defined by

the [IupSetHandle](#) function. When the elements were created in Lua, this attribute's contents is the name of a variable - more precisely, the one receiving the return from the function that created the element you wish to select. In LED it is not possible to dynamically change the value of any attribute, so the elements created in this environment must be identified and manipulated in C by means of their identifying name.

HOTSPOT

Hotspot is the position inside a cursor image indicating the mouse-click spot. Its value is given by the x and y coordinates inside a cursor image.

Value

"x:y", where x and y are integers defining the coordinates in pixels.

Default: NULL (no hotspot)

Affects

[IupImage](#)

See Also

[CURSOR](#)

HEIGHT

Image height in pixels. Verification-only attribute.

Value

Integer number.

Affects

[IupImage](#)

WIDTH

Image width in pixels. This attribute can only be consulted.

Value

Integer number.

Affects

[IupImage](#)

KEY

Associates a key to a menu or submenu item. Such key works as a shortcut when the menu is open.

Value

String containing a key description. Please refer to the [Keyboard Codes](#) table for a list of the possible values.

Default: NULL

Notes

IUP automatically underlines the first appearance of the chosen menu letter. For such, the chosen letter must necessarily be a part of the menu text.

In the menu bar, some systems automatically associate the ALT+<letter> combination for the chosen letter. This is valid for the Windows driver, but not for the Motif driver.

Be careful not to misuse this attribute in relation to [K_ANY](#) callback.

Affects

[IupItem](#), [IupSubMenu](#).

DROPDOWN

Changes the appearance of the list for the user: only the selected item is shown beside a button with an arrow pointing down. To select another option, the user must press this button, which displays all items in the list.

Value

"YES" or "NO".

Default: "NO"

Notes

This attribute is ignored for multiple lists (attribute [MULTIPLE](#) = "YES").

This attribute is only consulted when the dialog is first mapped ([IupMap](#), [IupShow](#), [IupShowXY](#) or [IupPopup](#)). After such, it cannot be changed.

Affects

[IupList](#)

MULTIPLE

Allows the simultaneous selection of several items. Otherwise, only one item can be selected at a time.

Value

"YES" or "NO".

Default: "NO"

Affects

[IupList](#)

VISIBLE_ITEMS

Number of items that appear when a DROPDOWN list is activated.

Value

Integer number.

Default: Depends on the native system.

Note

Only makes sense when the [DROPDOWN](#) attribute is "YES".

Affects

[IupList](#)

MARGIN

Defines a margin, in pixels, between an element's border and the elements contained by that element. Valid only for elements that contain other elements.

Value

"*widthxheight*", where *width* and *height* are integer values corresponding to the horizontal and vertical margins, respectively.

Default: 0x0 (no margin).

Affects

[IupZbox](#), [IupHbox](#), [IupVBox](#), [IupFrame](#).

ALIGNMENT

Defines the elements' alignment. Values vary according to the elements.

The default value, when it is not specified in an element, is "ACENTER".

Affects

[IupZbox](#), [IupHbox](#), [IupVBox](#), [IupFrame](#), [IupLabel](#).

GAP

Defines the space, in pixels, between the interface elements.

Value

Any integer number.

Default: "0"

Affects

[IupHbox](#), [IupVBox](#)

IMAGE

Bitmap image. Must be created with function `IupImage`.

Value

Name of an image.

Default: NULL.

Note

The definition after mapping is only assured if the image has the same size as the image it is replacing.

Affects

[IupButton](#), [IupToggle](#), [IupLabel](#).

See Also

[IupImage](#)

IMINACTIVE

Image of the element when the ACTIVE attribute equals "NO". Must be created using function `IupImage`.

Value

Name of an image.

Default: NULL.

Affects

[IupButton](#), [IupToggle](#).

See Also

[IupImage](#)

IMPRESS

Image of the element while the user keeps the left mouse button pressed over it. Must be created with function `IupImage`.

Value

Name of an image.

Default: NULL.

Note

When [IMPRESS](#) and [IMAGE](#) are defined, IUP does not show the element's borders to provide a 3D effect. The user must define the borders on the image.

Affects

[IupButton](#), [IupToggle](#).

See Also

[IupImage](#)

NC

Maximum number of characters.

Value

Positive integer number.

Default: 32767

Affects

[IupList](#), [IupText](#), [IupMultiline](#)

APPEND

Inserts a text at the end of the current text, independently from the caret's position.

In the Multiline, a "\n" character will be automatically inserted.

Value

Any text.

Note

Only works if the element is mapped.

Affects

[IupList](#), [IupMultiline](#), [IupText](#)

CARET

Places the insertion point in a text-edition field. The first line and the first column begin at 1.

Value (Multiline)

String with the "*line,column*" format, where *line* and *column* are integer numbers corresponding to the caret's position.

Default: "1,1" (first character in the first line).

Value (Text, List)

String in the "*pos*" format. *Pos* is an integer number corresponding to the caret's position.

Default: "1" (first character).

Note

When the value set for the line is greater than the number of lines, the caret is placed after the last line (only `multiline`). When the value set for the column is greater than the number of characters in a line, the caret is placed after the last character in the line.

Affects

[IupList](#), [IupMultiline](#), [IupText](#)

INSERT

Inserts a text in the caret's position.

Value

Any text, even with '\n' characters indicating line change.

Affects

[IupList](#), [IupMultiline](#), [IupText](#)

READONLY

Defines whether an element can be entered text or not.

Value

"YES" or "NO"

Default: "NO"

Note

Even though this attribute prevents the user from editing text, it allows the user to use the navigation keys (right, left, etc.).

Affects

[IupList](#), [IupMultiline](#), [IupText](#)

SELECTION

Modifies or returns the selection of a text-edition field.

Value (Multiline)

A text in the "lin1,col1:lin2,col2" format, where lin1,col1,lin2 and col2 are integer numbers corresponding to the selection's interval. The first position is "1".

Default: "1,1:1,1"

Value (Text,List)

A text in the "beg:end" format, where beg and end are integer numbers corresponding to the selection's interval.

Default: "1:1".

Affects

[IupList](#), [IupText](#), [IupMultiline](#)

SELECTEDTEXT

Modifies or consults the selected text.

Value

Text.

Note

The text is modified even if the element uses the [READONLY](#) attribute.

Affects

[IupList](#), [IupText](#), [IupMultiline](#)

BORDER

Shows a border around the element.

Value

"YES" or "NO".

Default: "YES".

Note

In some elements, such as `IupDialog`, this attribute is only consulted when the element is first mapped ([IupMap](#), [IupShow](#), [IupShowXY](#) or [IupPopup](#)). After this, it cannot be changed.

Affects

[IupDialog](#), [IupText](#), [IupMultiline](#), [IupCanvas](#).

ICON

Dialog's icon.

Value

Name of a IUP image.

Default: NULL

Note

Icon sizes are usually less than or equal to 32x32. On Windows, names of resources (.RES) linked to the application are also accepted. On Motif, it only works with some window managers, like *mwm* and *gnome*. Icon colors can have the BGCOLOR values, but it works better if it is at index 0.

The Windows SDK recommends that cursors and icons should be implemented as resources rather than created at run time.

Affects

[IupDialog](#)

MAXBOX

Requests a maximize button from the window manager.

Value

"YES" or "NO".

Default: "YES"

Note

This attribute is only consulted when the dialog is first mapped ([IupMap](#), [IupShow](#), [IupShowXY](#) or [IupPopup](#)). After such, it cannot be changed. On Motif, it only works if the active window-management system is *mwm*.

Affects

[IupDialog](#)

MINBOX

Requests a minimize button from the window manager.

Value

"YES" or "NO".

Default: "YES"

Note

This attribute is only consulted when the dialog is first mapped ([IupMap](#), [IupShow](#), [IupShowXY](#) or [IupPopup](#)). After such, it cannot be changed. On Motif, it only works if the active window-management system is *mwm*.

Affects

[IupDialog](#)

MENUBOX

When set, this attribute shows the menu box in a dialog title area.

Value

"YES" or "NO".

Default: "YES "

Note

This attribute is only consulted when the dialog is first mapped ([IupMap](#), [IupShow](#), [IupShowXY](#) or [IupPopup](#)). After such, it cannot be changed. On Motif, it only works if the active window-management system is *mwm*.

Affects

[IupDialog](#)

RESIZE

Allows interactively changing the dialog's size.

Value

"YES" or "NO".

Default: "YES "

Note

This attribute is only consulted when the dialog is first mapped ([IupMap](#), [IupShow](#), [IupShowXY](#) or [IupPopup](#)). After such, it cannot be changed. On Motif, it only works if the active window-management system is *mwm*.

Affects

[IupDialog](#)

MENU

Associates a menu to the dialog.

Value

Name of a menu-type interface element.

Affects

[IupDialog](#)

STARTFOCUS

Name of the dialog's element that must receive the focus right after the dialog is opened.

Value

Name of an element.

Affects

[IupDialog](#)

PARENTDIALOG

The dialog that specifies this attribute is treated as the child dialog of the specified value.

Value

Name of a IUP dialog.

Default: NULL.

Note

This behavior is system dependent, but usually what happens is: this dialog does not move behind its PARENTDIALOG, even if the user clicks the PARENTDIALOG. If the PARENTDIALOG is minimized, this dialog is automatically hidden.

Affects

[IupDialog](#)

DEFAULTENTER

Name of the default button activated when the user hits ENTER on a dialog.

Value

Identifier of a button.

Default: NULL.

Affects

[IupDialog](#)

DEFAULTESC

Name of the default button activated when the user hits ESC on a dialog.

Value

Identifier of a button.

Default: NULL.

Affects

[IupDialog](#)

SHRINK

If this attribute is defined, the elements will try to adjust even when the dialog's size is smaller than its minimum limit.

Value

"YES" or "NO".

Default: "NO".

Notes

When the user changes the size of the dialog, the elements are automatically re-distributed inside the dialog. Some elements even have their size changed if the `EXPAND` attribute is active. When this size is smaller than a minimum limit in which all elements still fit the dialog, the elements' distribution is no longer modified. Actually, the virtual size of the dialog remains larger than its actual size on the screen, and some elements to the right are hidden by the borders.

The `SHRINK` attribute offers an alternative to this behavior. It makes the elements continue to rearrange, even if they must overlap.

The results of this new rearrangement may vary according to the elements' distribution on the dialog.

Affects

[IupDialog](#)

FILE

Filename initially shown in the dialog's "File Name" field. If the user clicks OK, this attribute will contain the filename selected by the user.

Value

Any text.

Default: `NULL`

Affects

[IupFileDlg](#)

FILTER

File filter.

Value

String containing a list of file filters valid in the native system, separated by ';' without spaces.

Default: NULL.

Example

```
"*.C;*.LED;teste.*"
```

Affects

[IupFileDlg](#)

FILTERINFO

Filter description.

Value

Any text.

Default: NULL.

Affects

[IupFileDlg](#)

DIRECTORY

Initial directory.

Value

Any text.

Default: NULL (dialog opens current directory).

Note: on Windows98 and Windows2000, if the current directory does not have files corresponding to the chosen filter, the directory opened will be "My Documents".

Affects

[IupFileDlg](#)

ALLOWNEW

Indicates if inexistent filenames are accepted. If an inexistent filename is chosen, a message box will be shown.

Value

"YES" or "NO".

Default: if the dialog is of type "OPEN", default is "NO"; if the dialog is of type "SAVE", default is "YES".

Affects

[IupFileDlg](#)

NOCHANGEDIR

Indicates if the initial directory is to be restored after the user has navigated.

Value

"YES" or "NO".

Default: "YES".

Affects

[IupFileDlg](#)

FILEEXIST

Indicates whether the file defined by the FILE attribute exists or not. It is only valid if the user has pressed OK in the dialog. Can only be consulted.

Value

"YES" or "NO".

Default: "YES" or "NO" if the user presses OK; otherwise NULL.

Affects

[IupFileDlg](#)

CURSOR

Defines the element's cursor.

Value

Name of a cursor predefined by IUP:

"NONE"
"ARROW"

```

"BUSY"
"CROSS"
"HAND" (*)
"IUP" (*)
"MOVE"
"PEN" (*)
"RESIZE_N"
"RESIZE_S"
"RESIZE_W"
"RESIZE_E"
"RESIZE_NE"
"RESIZE_SE"
"RESIZE_NW"
"RESIZE_SW"
"TEXT"

```

Default: "ARROW"

(*) To use these cursors on Windows, the Iup.rc file, provided with IUP, must be added to the project.

It can receive as a parameter the name of an image, to be used as an application-defined cursor (the cursor must be a [IupImage](#), but the image is not a regular one. See the notes below).

Notes

For the image to represent a cursor, it must use attribute [HOTSPOT](#) to define its hotspot (place where the mouse click is actually effective). Only color indices 0, 1 and 2 can be used in a cursor, where 0 will be transparent. The RGB colors corresponding to indices 1 and 2 are defined just as in regular images. In Windows the cursor can have more than 2 colors. Cursor sizes are usually less than or equal to 32x32.

In the interface system, the cursor will only change when the interface system regains control.

The Windows SDK recommends that cursors and icons should be implemented as resources rather than created at run time.

When the cursor image is no longer necessary, it must be destroyed through function [IupDestroy](#). Attention: the cursor cannot be in use when it is destroyed.

Affects

[IupDialog](#), [IupCanvas](#)

See Also

[IupImage](#)

CONID

Canvas identifier for GKS/puc. This attribute's value must be passed as a connection identifier when opening a IUP-type workstation. It can only be consulted.

Affects

[IupCanvas](#)

SCROLLBAR

Associates a horizontal and/or vertical scrollbar to the canvas.

Value

"VERTICAL" , "HORIZONTAL" , "YES" (both) or "NO" (none).

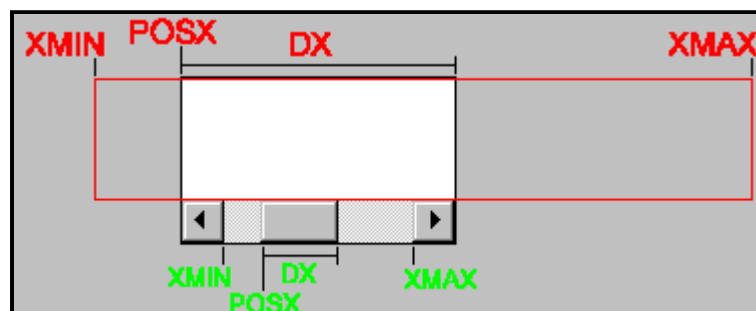
Default: "NO"

Notes

The scrollbar allows you to create a virtual space associated to the canvas. In the image below, such space is marked in **red**, as well as the attributes that affect the composition of this space. In **green** you can see how these attributes are reflected on the scrollbar.

Hence you can clearly deduce that POSX is limited to XMIN and XMAX-DX, or $XMIN \leq POSX \leq XMAX - DX$.

When the virtual space has the same size as the canvas, DX equals XMAX-XMIN, and at this moment the scrollbar could be hidden, as it is not useful (this behavior occurs only for the Win32 driver).



Affects

[IupCanvas](#)

POSX

Thumbnail position in the horizontal scrollbar in any unit.

Value

Any floating-point value. Must be a value between XMIN and XMAX-DX.

Default: "0 . 0"

Note

When the canvas is visible, a change in POSX generates a redraw in the horizontal scrollbar on the screen. This attribute does not generate a redraw of the canvas. Shall the user need this, he/she must call a redraw callback.

Affects

[IupCanvas](#)

See Also

[SCROLLBAR](#)

POSY

Thumbnail position in the vertical scrollbar in any unit.

Value

Any floating-point value. Must be a value between YMIN and YMAX-DY.

Default: "0 . 0"

Note

When the canvas is visible, a change in POSY generates a redraw in the vertical scrollbar on the screen. This attribute does not generate a redraw of the canvas. Shall the user need this, he/she must call a redraw callback.

Affects

[IupCanvas](#)

See Also

[SCROLLBAR](#)

DX

Size of the horizontal scrollbar's thumbnail in any unit.

Value

Any floating-point value greater than zero and smaller than the difference between [XMAX](#) and [XMIN](#).

Default:: "0.1".

Note

A change in these values will only be effective after attribute [POSX](#) or [POSY](#) has been changed.

Affects

[IupCanvas](#)

See Also

[SCROLLBAR](#)

DY

Size of the vertical scrollbar's thumbnail in any unit.

Value

Any floating-point value greater than zero and smaller than the difference between [YMAX](#) and [YMIN](#).

Default:: "0.1".

Note

A change in these values will only be effective after attribute [POSX](#) or [POSY](#) has been changed.

Affects

[IupCanvas](#)

See Also

[SCROLLBAR](#)

XMAX

Maximum value of the horizontal scrollbar, in any unit.

Value

Any floating-point value.

Default: "1 . 0"

Note

A change in this value will only be effective after attribute [POSX](#) or [POSY](#) is changed.

Affects

[IupCanvas](#)

See Also

[SCROLLBAR](#)

XMIN

Minimum value of the horizontal scrollbar, in any unit.

Value

Any floating-point value.

Default: "0 . 0"

Note

A change in this value will only be effective after attribute [POSX](#) or [POSY](#) is changed.

Affects

[IupCanvas](#)

See Also

[SCROLLBAR](#)

YMAX

Maximum value of the vertical scrollbar, in any unit.

Value

Any floating-point value.

Default: "1 . 0"

Note

A change in this value will only be effective after attribute [POSX](#) or [POSY](#) is changed.

Affects

[IupCanvas](#)

See Also

[SCROLLBAR](#)

YMIN

Minimum value of the vertical scrollbar, in any unit.

Value

Any floating-point value.

Default: "0 . 0"

Note

A change in this value will only be effective after attribute [POSX](#) or [POSY](#) is changed.

Affects

[IupCanvas](#)

See Also

[SCROLLBAR](#)

Global Attributes

All the attributes are for verification only.

VERSION

Returns the name of IUP's version.

Value

The value follows the "major.minor.driver" format, major referring to broader changes, minor referring to smaller changes and corrections, and driver referring to changes in the respective driver. Ex.: "1 . 7 . 2".

COPYRIGHT

Returns the IUP's copyright.

Value

Ex: "Copyright (C) 1994-2004 Tecgraf/PUC-Rio and PETROBRAS S/A".

DRIVER

Informs the current driver being used.

Value

Two drivers are available now, one for each platform: "MOTIF" and

"WIN32".

SYSTEM

Informs the current operating system.

Value

On UNIX, it is equivalent to the command `"uname -s"` (`sysname`). On Windows, it identifies if you are on NT, WinXP or 98.

Several values can be provided:

```
"Linux"
"SunOS"
"Solaris"
"IRIX"
"AIX"
"Win95"
"Win95OSR2"
"Win98"
"Win98SE"
"WinMe"
"WinNT"
"Win2K"
"WinXP"
```

SYSTEMVERSION

Informs the current operating system version.

Value

On UNIX, it is equivalent to the command `"uname -r"` (`release`). On Windows, it identifies the system version with build number and service pack version.

SCREENSIZE

Returns the screen size in pixels. In Windows it excludes the task bar area.

Value

String in the *"widthxheight"* format.

SCREENDEPTH

Returns the screen depth in bits per pixel.

LOCKLOOP

Locks the loop even when all dialogs have been closed. Possible values: "YES" or "NO".

CURSPOS

This attribute programatically changes the cursor position. Accept values in the format "posh \times posv", example "200 \times 200", in absolute coordinates relative to the upper left corner of the screen.

COMPUTERNAME

Returns the hostname.

USERNAME

Returns the user logged in.

Keyboard Codes

The table below shows the IUP codification of every key in the keyboard. Each key is represented by an integer value, defined in the **iupkey.h** file, which must be included in the application.

IUP uses the US default codification this means that if you installed a keyboard specific for your country the key codes will be different from the real keys for a small group of keys. For the Brazilian ABNT keyboard the keys ", ' , ~ , ^ , & ,] , } , [, { , ' , ` , usually will have a different codification. But this does not affect the IupText and IupMultiline text input.

Notice that some key combinations are not available, like: Shift+Ins, Shift+Del, Alt+Space, Alt/Ctrl/Shift+Backspace, Alt/Ctrl/Shift+Pause, Alt/Ctrl/Shift+Esc, Ctrl/Alt+Enter. When CapsLock is active the Shift+<Key> combination is used, except for Esc and Backspace that will ignore the combination.

The **isxkey(key)** macro defined in the **iupkey.h** file informs whether a given key is an extended code instead of an alphanumeric key.

In IUP the codification implies that some keys have the same code: K_BS=K_cH, K_TAB=K_cI and K_CR=K_cM.

Key	Code / Attribute	Key	Code / Attribute	Key	Code / Attribute
SPACE	K_SP	Alt-A	K_mA	Ctrl-SPACE	K_cSP
!	K_exclam	Alt-B	K_mB	Ctrl-A	K_cA
"	K_quotedbl	Alt-C	K_mC	Ctrl-B	K_cB
#	K_numbersign	Alt-D	K_mD	Ctrl-C	K_cC
\$	K_dollar	Alt-E	K_mE	Ctrl-D	K_cD

%	K_percent	Alt-F	K_mF	Ctrl-E	K_cE
&	K_ampersand	Alt-G	K_mG	Ctrl-F	K_cF
'	K_quoteright	Alt-H	K_mH	Ctrl-G	K_cG
(K_parentleft	Alt-I	K_mI	Ctrl-H	K_cH
)	K_parentright	Alt-J	K_mJ	Ctrl-I	K_cI
*	K_asterisk	Alt-K	K_mK	Ctrl-J	K_cJ
+	K_plus	Alt-L	K_mL	Ctrl-K	K_cK
,	K_comma	Alt-M	K_mM	Ctrl-L	K_cL
-	K_minus	Alt-N	K_mN	Ctrl-M	K_cM
.	K_period	Alt-O	K_mO	Ctrl-N	K_cN
/	K_slash	Alt-P	K_mP	Ctrl-O	K_cO
0	K_0	Alt-Q	K_mQ	Ctrl-P	K_cP
1	K_1	Alt-R	K_mR	Ctrl-Q	K_cQ
2	K_2	Alt-S	K_mS	Ctrl-R	K_cR
3	K_3	Alt-T	K_mT	Ctrl-S	K_cS
4	K_4	Alt-U	K_mU	Ctrl-T	K_cT
5	K_5	Alt-V	K_mV	Ctrl-U	K_cU
6	K_6	Alt-W	K_mW	Ctrl-V	K_cV
7	K_7	Alt-X	K_mX	Ctrl-W	K_cW
8	K_8	Alt-Y	K_mY	Ctrl-X	K_cX
9	K_9	Alt-Z	K_mZ	Ctrl-Y	K_cY
:	K_colon	Alt-1	K_m1	Ctrl-Z	K_cZ
;	K_semicolon	Alt-2	K_m2	Ctrl-Tab	K_cTAI
<	K_less	Alt-3	K_m3	Ctrl-Home	K_cCHOM
=	K_equal	Alt-4	K_m4	Ctrl-UP	K_cUP
>	K_greater	Alt-5	K_m5	Ctrl-PgUp	K_cPGU
?	K_question	Alt-6	K_m6	Ctrl-LEFT	K_cLEI

@	K_at	Alt-7	K_m7	Ctrl-MIDDLE	K_cMID
A	K_A	Alt-8	K_m8	Ctrl-RIGHT	K_cRIE
B	K_B	Alt-9	K_m9	Ctrl-END	K_cEND
C	K_C	Alt-0	K_m0	Ctrl-DOWN	K_cDOV
D	K_D	Alt-Tab	K_mTAB	Ctrl-PgDn	K_cPGI
E	K_E	Alt-Home	K_mHOME	Ctrl-Insert	K_cINS
F	K_F	Alt-UP	K_mUP	Ctrl-Del	K_cDEL
G	K_G	Alt-PgUp	K_mPGUP	Ctrl-F1	K_cF1
H	K_H	Alt-LEFT	K_mLEFT	Ctrl-F2	K_cF2
I	K_I	Alt-RIGHT	K_mRIGHT	Ctrl-F3	K_cF3
J	K_J	Alt-END	K_mEND	Ctrl-F4	K_cF4
K	K_K	Alt-DOWN	K_mDOWN	Ctrl-F5	K_cF5
L	K_L	Alt-PgDn	K_mPGDN	Ctrl-F6	K_cF6
M	K_M	Alt-Insert	K_mINS	Ctrl-F7	K_cF7
N	K_N	Alt-Del	K_mDEL	Ctrl-F8	K_cF8
O	K_O	Alt-F1	K_mF1	Ctrl-F9	K_cF9
P	K_P	Alt-F2	K_mF2	Ctrl-F10	K_cF10
Q	K_Q	Alt-F3	K_mF3	Ctrl-F11	K_cF11
R	K_R	Alt-F4	K_mF4	Ctrl-F12	K_cF12
S	K_S	Alt-F5	K_mF5		
T	K_T	Alt-F6	K_mF6		
U	K_U	Alt-F7	K_mF7		
V	K_V	Alt-F8	K_mF8		
W	K_W	Alt-F9	K_mF9		
X	K_X	Alt-F10	K_mF10		
Y	K_Y	Alt-F11	K_mF11		
Z	K_Z	Alt-F12	K_mF12		

[K_bracketleft
\	K_backslash
]	K_bracketright
^	K_circum
_	K_underscore
`	K_quotleft
a	K_a
b	K_b
c	K_c
d	K_d
e	K_e
f	K_f
g	K_g
h	K_h
i	K_i
j	K_j
k	K_k
l	K_l
m	K_m
n	K_n
o	K_o
p	K_p
q	K_q
r	K_r
s	K_s
t	K_t
u	K_u

v	K_v
w	K_w
x	K_x
y	K_y
z	K_z
{	K_braceleft
	K_bar
}	K_braceright
~	K_tilde
ESC	K_ESC
Enter	K_CR
BackSpace	K_BS
Insert	K_INS
Del	K_DEL
Tab	K_TAB
Home	K_HOME
UP	K_UP
PgUp	K_PGUP
LEFT	K_LEFT
MIDDLE	K_MIDDLE
RIGHT	K_RIGHT
END	K_END
DOWN	K_DOWN
PgDn	K_PGDN
Pause	K_PAUSE
F1	K_F1
F2	K_F2

F3	K_F3
F4	K_F4
F5	K_F5
F6	K_F6
F7	K_F7
F8	K_F8
F9	K_F9
F10	K_F10
F11	K_F11
F12	K_F12

Character Fonts

"HELvetica_NORMAL_8"	"COURIER_NORMAL_8"	"TIMES_NORMAL_8"
"HELvetica_ITALIC_8"	"COURIER_ITALIC_8"	"TIMES_ITALIC_8"
"HELvetica_BOLD_8"	"COURIER_BOLD_8"	"TIMES_BOLD_8"
"HELvetica_NORMAL_10"	"COURIER_NORMAL_10"	"TIMES_NORMAL_10"
"HELvetica_ITALIC_10"	"COURIER_ITALIC_10"	"TIMES_ITALIC_10"
"HELvetica_BOLD_10"	"COURIER_BOLD_10"	"TIMES_BOLD_10"
"HELvetica_NORMAL_12"	"COURIER_NORMAL_12"	"TIMES_NORMAL_12"
"HELvetica_ITALIC_12"	"COURIER_ITALIC_12"	"TIMES_ITALIC_12"
"HELvetica_BOLD_12"	"COURIER_BOLD_12"	"TIMES_BOLD_12"
"HELvetica_NORMAL_14"	"COURIER_NORMAL_14"	"TIMES_NORMAL_14"
"HELvetica_ITALIC_14"	"COURIER_ITALIC_14"	"TIMES_ITALIC_14"
"HELvetica_BOLD_14"	"COURIER_BOLD_14"	"TIMES_BOLD_14"

Events

IUP is a graphic-interface library, so most of the time it waits for an event to occur, such

as a button click or a mouse leaving a window. The user can inform IUP that he/she wishes callbacks to be called, informing that an event has taken place. For further information on callbacks, see [Guide / Events](#).

Attention: in a callback if one of the parameters is a string, this string may be modified during the callback if another IUP function (such as `IupGetAttribute`) is called.

ACTION

Action generated when the element is activated. Affects each element differently.

Callback

```
int function(Ihandle *self); [in C]
elem:action() -> (ret: number) [in IupLua]
```

self: identifier of the element that activated the function.

In some elements, this callback may receive more parameters, apart from **self**. Please refer to each element's documentation.

Affects

[IupButton](#), [IupItem](#), [IupList](#), [IupText](#), [IupCanvas](#),
[IupMultiline](#), [IupToggle](#)

BUTTON_CB

Action generated when a mouse button is pressed or released.

Callback

```
int function(Ihandle* self,int but,int pressed,int x,int y,char* status); [in C]
elem:button(but, pressed, x, y: number, status: string) -> (ret: number) [in :
elem:button_cb(but, pressed, x, y: number, status: string) -> (ret: number) [:
```

self: identifies the canvas that activated the function's execution.

but: identifies the activated mouse button:

```
IUP_BUTTON1 left mouse button (button 1);
IUP_BUTTON2 middle mouse button (button 2);
IUP_BUTTON3 right mouse button (button 3).
```

pressed: indicates the state of the button:

```
0 mouse button was released;
1 mouse button was pressed.
```

x, **y**: position in the canvas where the event has occurred, in pixels.

status: status of the mouse buttons and some keyboard keys at the moment the event is generated. The following macros must be used for verification:

```

isshift(status)
iscontrol(status)
isbutton1(status)
isbutton2(status)
isbutton3(status)
isdoubling(status)

```

They return 1 if the respective key or button is pressed, and 0 otherwise.

Affects

[IupCanvas](#), [IupButton](#)

CLOSE_CB

Called just before a dialog is hidden due to some action over it - for example, double clicking the system's menu box, usually located to the left in the title bar.

Callback

```

int function(Ihandle *self); [in C]
elem:close() -> (ret: number) [in IupLua3]
elem:close_cb() -> (ret: number) [in IupLua5]

```

Returning `IUP_IGNORE`, it prevents the dialog from being hidden. If you destroy the dialog in this callback, you must return `IUP_IGNORE`.

Affects

[IupDialog](#)

DEFAULT_ACTION

Predefined IUP action, generated every time an action has no associated function.

Callback

```

int function(Ihandle *self); [in C]
[There is no Lua equivalent]

```

self: identifier of the element that activated the function.

Note

Often a programmer defines an action with a name and, when associating it to a function, he/she mistypes the action name, or vice-versa. This kind of mistake is very common, and IUP is not able to automatically detect it. The predefined `IUP_DEFAULT_ACTION` action, combined with function `IupGetActionName`, can help the programmer detect this problem. All you have to do is define a default action and verify which is the name of the action that activated it.

Affects

Global callback.

See Also

[IupSetFunction](#), [IupGetActionName](#).

ENTERWINDOW_CB

Action generated when the mouse enters the canvas.

Callback

```
int function(Ihandle *self); [in C]
elem:enterwindow() -> (ret: number) [in IupLua3]
elem:enterwindow_cb() -> (ret: number) [in IupLua5]
```

self: identifier of the canvas the mouse has entered.

Affects

[IupCanvas](#)

GETFOCUS_CB

Action generated when an element is given keyboard focus. This callback is called after the KILLFOCUS_CB.

Callback

```
int function(Ihandle *self); [in C]
elem:getfocus() -> (ret: number) [in IupLua3]
elem:getfocus_cb() -> (ret: number) [in IupLua5]
```

self: identifier of the element that received keyboard focus.

Affects

All elements with user interaction, except menus.

See Also

[KILLFOCUS_CB](#)

HELP_CB

Action generated when the user press F1 at a control. In Motif is also activated by the Help button in some workstations keyboard.

Callback

```
void function(Ihandle *self); [in C]
elem:help() -> (ret: number) [in IupLua3]
```

```
elem:help_cb() -> (ret: number) [in IupLua5]
```

self: identifier of the element that received keyboard focus.

Affects

All elements with user interaction.

HIGHLIGHT_CB

Callback triggered every time the mouse pointer hovers an IupItem.

Callback

```
int function(Ihandle *self); [in C]
elem:highlight() -> (ret: number) [in IupLua3]
elem:highlight_cb() -> (ret: number) [in IupLua5]
```

Comments

This callback should not be used with popup menus.

Affects

[IupItem](#)

IUP_IDLE_ACTION

Predefined IUP action, generated when there are no events.

Callback

```
int function(); [in C]
```

Note

Often used to perform background operations. For example, a time-consuming drawing operation may allow the user to take a decision before the operation is over.

Lua Binding

To modify this action, function **IupSetIdle(myfunction)** must be used. Use **iup.SetIdle(myfunction)** in IupLua5. Using `nil` as a parameter removes the association.

Examples

Affects

Global callback.

See Also

[IupSetFunction.](#)

K_ANY

Action generated when a keyboard event occurs.

Callback

```
int function(Ihandle *self, int c); [in C]
elem:k_any() -> (ret: number) [in IupLua]
```

self: identifier of the element where the user typed something.

c: identifier of typed key. Please refer to the [Keyboard Codes](#) table for a list of possible values.

If the function returns IUP_IGNORE, the system will ignore the typed character. If the function returns the code of any other key, IUP will treat this new key instead of the one typed by the user.

If the function returns IUP_CONTINUE, the event will be propagated to the parent of the element receiving it.

Notes

All defined keys are also callbacks of any element, called when the respective key is activated. For example: "K_cC" is also a callback activated when the user press Ctrl+C. A shortcut key or hot key can also be associated to any existing callback, either in a menu or any other element, using this same mechanism. These callbacks do not work in IupLua.

The K_ANY callback is a callback that depends on the keyboard focus and the keyboard usage of the control with the focus. It is usually only set for dialogs, but if a control set the K_ANY callback the dialog callback will only be called if the control callback returns IUP_CONTINUE.

Also some keys may not activate the callback since they are reserved keys, like Enter and Esc.

Affects

All.

KEYPRESS_CB

Action generated when a key is pressed or released. If the key is pressed and held several calls will occur.

Callback

```
int function(Ihandle *self, int c, int press); [in C]
elem:keypress(c, press: number) -> (ret: number) [in IupLua3]
elem:keypress_cb(c, press: number) -> (ret: number) [in IupLua5]
```

self: identifier of the element.

c: identifier of typed key. Please refer to the [Keyboard Codes](#) table for a list of possible values.

press: 1 is the user pressed the key or 0 otherwise.

This function may return IUP_CLOSE.

Affects

[IupCanvas](#).

KILLFOCUS_CB

Action generated when an element loses keyboard focus. This callback is called before the GETFOCUS_CB.

Callback

```
int function(Ihandle *self); [in C]
elem:killfocus() -> (ret: number) [in IupLua3]
elem:killfocus_cb() -> (ret: number) [in IupLua5]
```

self: identifier of the element that lost keyboard focus.

Affects

All elements with user interaction, except menus.

See Also

[GETFOCUS_CB](#)

LEAVEWINDOW_CB

Action generated when the mouse leaves a canvas.

Callback

```
int function(Ihandle *self); [in C]
elem:leavewindow() -> (ret: number) [in IupLua3]
elem:leavewindow_cb() -> (ret: number) [in IupLua5]
```

self: identifier of the canvas the mouse left

Affects

[IupCanvas](#)

MAP_CB

Called right after an element is mapped.

Callback

```
int function(Ihandle *self); [in C]
elem:mapcb() -> (ret: number) [in IupLua3]
elem:map_cb() -> (ret: number) [in IupLua5]
```

Affects

[IupDialog](#), [IupCanvas](#)

MENUCLOSE_CB

Called just before a submenu is closed.

Callback

```
int function(Ihandle *self); [in C]
elem:menuclose() -> (ret: number) [in IupLua3]
elem:menuclose_cb() -> (ret: number) [in IupLua5]
```

Comments

Does not work for popup menus.

Affects

[IupMenu](#), [IupSubMenu](#)

MOTION_CB

Action generated when the mouse moves.

Callback

```
int function(Ihandle *self, int x, int y, char *r); [in C]
elem:motion(x, y: number, r: string) -> (ret: number) [in IupLua3]
elem:motion_cb(x, y: number, r: string) -> (ret: number) [in IupLua5]
```

self: identifier of the canvas that activated the function's execution.

x, **y**: position in the canvas where the event has occurred, in pixels.

r: status of mouse buttons and certain keyboard keys at the moment the event was generated. The following macros must be used for verification:

```
isshift(r)
iscontrol(r)
isbutton1(r)
isbutton2(r)
isbutton3(r)
isdoube(r)
```

Affects

[IupCanvas](#)

OPEN_CB

Called just before a submenu is opened.

Callback

```
int function(Ihandle *self); [in C]
elem:open() -> (ret: number) [in IupLua3]
elem:open_cb() -> (ret: number) [in IupLua5]
```

Comments

Does not work for popup menus.

Affects

[IupMenu](#), [IupSubMenu](#)

RESIZE_CB

Action generated when the canvas size is changed.

Callback

```
int function(Ihandle *self, int width, int height); [in C]
elem:resize(width, height: number) -> (ret: number) [in IupLua3]
elem:resize_cb(width, height: number) -> (ret: number) [in IupLua5]
```

self: identifier of the canvas that activated the function's execution.

width: new canvas width, in pixels.

height: new canvas height, in pixels.

Note

This action is also generated right after the dialog is viewed by means of functions [IupShow](#), [IupShowXY](#) or [IupPopup](#).

In Windows, it is also generated after a map and before show.

Affects

[IupCanvas](#)

SCROLL_CB

Called when some manipulation is made to the scrollbar. When this attribute is defined, the [ACTION](#) callback is not called in such cases.

Callback

```
int function(Ihandle *self, int op, float posx, float posy); [in C]
elem:scroll(op, posx, posy: number) -> (ret: number) [in IupLua3]
```

```
elem:scroll_cb(op, posx, posy: number) -> (ret: number) [in IupLua5]
```

op: indicates the operation performed on the scrollbar.

If the manipulation was made on the vertical scrollbar, it can have the following values:

```
IUP_SBUP - line up
IUP_SBDN - line down
IUP_SBPgup - page up
IUP_SBPgdn - page down
IUP_SBPOSV - vertical position
IUP_SBDRAgv - vertical drag
```

If it was on the horizontal scrollbar, the following values are valid:

```
IUP_SBLEFT - column left
IUP_SBRIGHT - column right
IUP_SBPgleft - page left
IUP_SBPgright - page right
IUP_SBPOSH - horizontal position
IUP_SBDRAgh - horizontal drag
```

posx, **posy**: the same as the [ACTION](#) canvas callback (corresponding to the values of attributes IUP_POSX and IUP_POSY).

Affects

[IupCanvas](#)

SHOW_CB

Called right after the dialog is opened, minimized or restored from a minimization.

Callback

```
int function(Ihandle *self, int mode); [in C]
elem:showcb(mode: number) -> (ret: number) [in IupLua3]
elem:show_cb(mode: number) -> (ret: number) [in IupLua5]
```

Parameter **mode** indicates which of the following situations generated the event:

0 - Show, 1 - Restore, 2 - Minimize.

Affects

[IupDialog](#)

WHEEL_CB

Action generated when the mouse wheel is rotated. If this callback is not defined the wheel will automatically scroll the canvas in the vertical direction by some lines, the SCROLL_CB callback if defined will be called with the IUP_SBDRAgv operation.

Callback

```
int function(Ihandle *self, float delta, int x, int y, char *r); [in C]
elem:wheel(delta, x, y: number, r: string) -> (ret: number) [in IupLua3]
elem:wheel_cb(delta, x, y: number, r: string) -> (ret: number) [in IupLua5]
```

self: identifier of the canvas that activated the function's execution.

delta: the amount the wheel was rotated in notches.

x, y: position in the canvas where the event has occurred, in pixels.

r: status of mouse buttons and certain keyboard keys at the moment the event was generated. The following macros must be used for verification:

```
isshift(r)
iscontrol(r)
isbutton1(r)
isbutton2(r)
isbutton3(r)
isdoublr(r)
```

Notes

In Motif delta is always 1 or -1. In Windows in some situations delta can reach the value of two. In the future with more precise wheels this increment can be changed.

The wheel will only work if the focus is at the canvas.

Affects

[IupCanvas](#)

WOM_CB

Action generated when an audio device receives an event.

[WINDOWS DRIVER ONLY]

Callback

```
void function(Ihandle *self, int v); [in C]
(not implemented in Lua)
```

where v is -1, 0, 1 meaning closing, ending and opening respectively.

Affects

[IupCanvas](#)

Drivers

IUP's ability to work in several platforms is due to the existence of individually implemented drivers for different platforms.

Motif Driver

Driver for the X-Windows/Motif environment.

Environment Variables

QUIET

When this variable is set, IUP does not generate the message indicating the driver's version when initializing.

DEBUG

This variable's existence makes the driver operate in synchronous mode with the X server. This slows down all operations, but allows immediately detecting errors caused by X.

Exclusive Attributes

MOTIFVERSION (global)

Returns the version of the run time Motif.

TRUECOLORCANVAS (global)

Indicates if the display allows creating TrueColor (> 8bpp) windows, even if PseudoColor is the default. Returns "YES" or "NO".

AUTOREPEAT (global)

Turns on/off ("YES" or "NO") the autorepeat of keyboard keys in the whole system - may be used as an optimization in high performance applications.

XDISPLAY (all)

Returns a `Display*`, indicating the control's X display. It is a verification-only attribute, available after the control is mapped.

XWINDOW (all)

Returns a `Window`, indicating the control's X window. It is a verification-only attribute, available after the control is mapped.

XSCREEN (all)

Returns a `Screen*`, indicating the control's X screen. It is a verification-only attribute, available after the control is mapped.

MOTIF_FONT (all)

This attribute, as well as `FONT`, can be used to change a control's font. Is should not be used, being specified only to keep compatibility. To change a control's font, use `FONT`.

Differentiated Attributes

[ENTERWINDOW_CB](#) / [LEAVEWINDOW_CB](#) (all)

The same callback available for CANVAS can be used for other elements.

[ICON](#) (DIALOG)

This attribute's value must be a string containing the name of the IUP image to be used as an icon when the dialog is minimized. The current window manager will determine how, or if, the icon will be displayed.

[WID](#)

This attribute returns a `Widget` which identifies the Intrinsic control. It is a verification-only attribute, available after the control is mapped.

[CURSOR](#) (all menus and submenus)

This attribute can be used in other elements as well as the canvas and dialog.

Window Manager Dependent Attributes

The attributes below can have different behaviors depending on the window manager controlling the display where the application is being viewed. They are typically attributes that control dialog visual characteristics, since these are drawn by the window manager.

[MAXBOX](#) (DIALOG)

[MINBOX](#) (DIALOG)

[MENUBOX](#) (DIALOG)

[RESIZE](#) (DIALOG)

[BORDER](#) (DIALOG)

[ICON](#) (DIALOG)

Differentiated Functions

[IupFlush](#)

Does not always work. If there is an example sent by the X server which is not yet in the event queue, after a call to `IupFlush` the queue might not be empty.

Default Values – Resource Files

Some default values used by the driver, such as background color, foreground color and font, can be set by the user by means of a resource file called `Iup`. It must be in the user's home or in a directory pointed to by the `XAPPLRESDIR` environment variable. Below you can see an example of this file's contents:

```
*background: #ff0000
*foreground: #a0ff00
*fontList: -misc-fixed-bold-r-normal-*--13-*
```

The values used in the example above are the ones used by IUP if these resources are not defined.

Generating Applications

IUP/Motif is composed by two libraries: `iupmot` and `iup`. They use the Motif (Xm), Xtoolkit (Xt) and Xlib (X11) libraries. To link an application to IUP, use the following options in the linker call (in the same order):

```
-liup -lXm -lXmu -lXt -lX11 -lm
```

Though these are the minimum requirements, depending on the platform other libraries might be needed. Typically, they are X extensions (Xext), needed in SunOS, and Xpm, needed in Linux. They must be listed after Xt and before X11. For instance:

```
-liup -lXm -lXmu -lXt -lXpm -lXext -lX11 -lm
```

Usually these libraries are placed in default directories, being automatically located by the linker. When the linker warns you about a missing library, add their location directories with option `-L`. In Tecgraf, some machines require such option:

Padrão	<code>-L/usr/lib -I/usr/include</code>
Linux	<code>-L/usr/X11R6/lib -I/usr/X11R6/include</code>
IRIX	<code>-L/usr/lib32 -I/usr/include/X11</code>
Solaris	<code>-L/usr/openwin/lib -I/usr/openwin/share/include/X11</code>
AIX	<code>-I/usr/include/Motif2.1</code>

In systems that support dynamic libraries, the library name is `libiup.so`. To force a link with static libraries in these systems, use option `-static`. In this case, the library name will be `libiup.a`.

Following are some makefile suggestions. All of them can be used in SunOS ([Sun](#)), IRIX ([Silicon](#)) and AIX ([IBM](#)) systems. For [Linux](#), `-lXpm` must be added at the end of the SYSLIBS variable.

- [Simple Makefile](#)
This makefile can be used to generate simple applications which use only IUP.
- [Makefile for IUP with CD](#)
For applications that use the CD graphics system.
- [Makefile to generate several versions](#)
This makefile is a base to generate several versions of the application, one for each platform. Each version is stored in a separate directory, managed by the makefile.

The available IUP binaries and the tests were done in the following systems:

- Linux24 = Red Hat 7.3 (i686) / Kernel 2.4.18-27.7.x / gcc 2.95.3 / Motif 2.1.30
- Linux24g3 = Red Hat Fedora (i686) / Kernel 2.4.22-1.2199 / gcc 3.3.2 / Motif 2.2.2
- AIX43 = IBM AIX 4.3 / gcc 2.95.2 / Motif 2.1.0
- IRIX65 = SGI IRIX 6.5 / gcc 3.0.4 / Motif 2.1.20
- IRIX6465 = SGI IRIX 6.5 (64 bits OS, but libs are still 32 bits) / gcc 3.0.4 / Motif 1.2.4
- IRIX6465cc = SGI IRIX 6.5 (") / cc MIPSpro 7.30 / Motif 1.2.4

- SunOS57 = Sun Solaris 7 (sparc) / gcc 2.95.2 / Motif 2.1.0
- SunOS58 = Sun Solaris 8 (sparc) / gcc 2.95.3 / Motif 2.1.0

Tips

- **During linking in the Solaris environment: Can not find libresolv.so.2**

This error occurs if the system does not have an applied patch containing this library.

This library is important for all installations of Solaris 2.5 and 2.5.1 (SunOS 5.5 and 5.5.1, respectively). It is a correction of the DNS system, involving security.

The web address to get these patches is SunSolve's

<http://sunsolve1.sun.com/sunsolve/pubpatches/patches.html>. Select the Solaris version you wish (2.5 or 2.5.1 for Sparc) and download the patches 103667-09, 102980-17, 103279-03, 103708-02, or more recent for 2.5 (the number after the '-' is the patch version, and the more recent number is the patch), or 103663-12, 103594-14, 103680-02 and 103686-02 for 2.5.1. All of them have a README file explaining installation, and groups have to be installed together.

- **TrueColor canvas**

Whenever a canvas is created, one tries to create it with a TrueColor resolution Visual. This is not always possible, since it is subject to many conditions, such as hardware (graphics board) and the X server's configuration.

The **xdpyinfo** program informs which Visuals are available in the X server where the display is being made, so that you can see if your X allows creating a canvas with a TrueColor Visual. In some platforms, however, the X server may not make a TrueColor Visual available, even though the graphics board is able to display it. In this case, restart the server with parameters that force this. Below is a table with such parameters to some systems where the IUP library has been tested. If the command does not work, or if it is not possible, then the graphics library really does not support 24 bpp.

System	Execution Command
Linux	<code>startx --bpp 24</code>
AIX	(not necessary)
IRIX	(not necessary)
Solaris	(not necessary)

Since color requests are “always” successful in TrueColor/24bpp windows, we have minimized visualization problems for images that make use of complex color palettes (when there is a high color demand, not always all colors requested can be obtained). The IUP applications also coexist more “peacefully” with other applications and among themselves, since the colors used by TrueColor/24bpp windows do not use the colormap cells used by all applications.

- **XtAddCallback failed**

When a warning about XtAddCallback appears during the application initialization, and it aborts, this means that you are using a Motif with a different version than the Motif

used to build IUP. Reinstall Motif or rebuilds IUP using your Motif.

Win32 Driver

This driver was designed for the modern Microsoft Windows in 32 bits (98/2000/XP).

Environment Variables

VERSION

When this variable is set, IUP generates a message indicating the driver's version when initializing.

Exclusive Attributes

HINSTANCE (global)

This attribute returns a handle (**HINSTANCE**) that identifies the application in the native system. It is a verification-only attribute.

SYSTEMLANGUAGE (global)

Return respectively a text with a description of the system language.

WIN_DEFAULTFONT (global)

Stores the name of the default font used in the interface controls.

SHIFTKEY (global)

Returns the state of the Shift keys (left and right). Possible values: "ON" or "OFF".

CONTROLKEY (global)

Returns the state of the Control keys (left and right). Possible values: "ON" or "OFF".

WINFONT (all)

This attribute is still used to maintain compatibility with previous versions. Use attribute [FONT](#).

This attribute's value must be a string with the following format:

"name:attributes:size"

name: The name the user will see (Times New Roman, MS Sans Serif, etc.).

attributes: Can be empty, or a list separated by commas with the following names: BOLD ITALIC UNDERLINE STRIKEOUT

size: Size in pixels

Examples:

```
"Times New Roman::10"
"Ms Sans Serif:ITALIC:20"
"Courier New:BOLD,STRIKEOUT:15"
```

Differentiated Attributes

[ICON](#) (DIALOG)

This attribute's value must be a string with the name of a Iup image or an icon in a Windows resource file linked to the application. This icon will be used when the dialog is minimized.

If a Iup image with a name associated to this attribute exists, it will be used to define the dialog's icon. The Iup images used as icons must necessarily have **32x32** points. If there is no Iup image whose name was passed to the attribute, Iup will look for a native icon (Windows) linked to the program.

[WID](#)

This attribute returns a handle (HWND) that identifies the window in the native system. It is a verification-only attribute, only available after the control is mapped.

Exclusive Callbacks

DROPPFILES_CB (DIALOG, CANVAS)

Callback called when a file is "dragged" to the application. When several files are dragged, the callback is called several times, once for each file. The callback must return IUP_DEFAULT to be called again for each of the dragged files. Returning IUP_IGNORE, the process is interrupted.

```
int function(Ihandle *self, char* filename, int numFile, int posx, :
```

self: Indicator of the element that received the file drop.

filename: Name of the dragged file.

numFile: Number of the dragged file. If several files are dragged, numFile counts the number of dragged files up to zero.

posx: X coordinate of the point where the user released the mouse button.

posy: Y coordinate of the point where the user released the mouse button.

Note: The callback must be set before the element is mapped.

Generating Applications

To link applications, libraries **iup.lib**, **ole32.lib** and **comctl32.lib** (the last two provided with the compilers) must be added.

The **iup.rc** resource file must be included in the application's project/makefile so that HAND, IUP, PEN and SPLITH cursors can be used.

DLL

To use DLL, it is necessary to link the application with the `IUP.lib` and `IUPSTUB.lib` libraries (for technical reasons, these libraries cannot be unified). Note that `IUP.lib` is a library specially generated to work with `iup.dll`, and is usually distributed in the same directory as `iup.dll`. the IUP DLL depends on the `MSVCRT.DLL`, that it is already installed in Windows.

For the program to work, `IUP.dll` must be inside a `PATH` directory. Usually the program does not need to be relinked when the DLL is updated.

Debug Versions

While using the debug version, two types of fatal errors can occur:

- 1) Protection errors: "Unhandled exception: access violation"
- 2) Assertive errors: "Assertion failed!"

In the second type, a dialog is shown with buttons `Abort`, `Retry` and `Ignore`, as well as a number of information, among which:

- + Name of the font file where the error occurred
- + Line number

The bug-correction process (if it exists) becomes a lot faster when this information is provided. Therefore, if you receive such error, please send this information along in the e-mail.

Tips

- On Windows a common error occurs: "Cannot find function `InitCommonCtrl ()`" This error occurs if you forgot to add the `comctl32.lib` library to be linked with the program. This library is **not** usually in the libraries list for the Visual C++, you must add it.