IUP
Portable User Interface

Version 2.4

(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 it 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.

IUP Project Management:

Antonio Escaño Scuri

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 - the toolkit provides the application portability. Supported systems include: Motif and Microsoft Windows 2000/XP/2003.

IUP uses an abstract layout model based on the boxes-and-glue paradigm from the $\TeX$ 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`.

- **Additional** (elements built outside the main library): **`dial, gauge, matrix, tabs, valuator, OpenGL canvas, color chooser, color browser.`**
- **Dialogs** (useful predefined dialogs): **`file selection, message, alarm, data input, list selection.`**

Hence IUP has some advantages over other interface toolkits available:

- **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, FreeBSD and Linux) using Motif 2.x (and optionally 1.x)
- Microsoft Windows 2000/XP/2003

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 control or driver 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**.

We host **IUP** support features at **LuaForge**. It provides us Tracker, Lists, News, CVS and Files. The IUP page at **LuaForge** is available at: http://luaforge.net/projects/iup/.

The discussion list is available at: http://lists.luaforge.net/mailman/listinfo/iup-users.
You can also submit *Bugs*, *Feature Requests* and *Support Requests* at:
http://luaforge.net/tracker/?group_id=89.
Source code, pre-compiled binaries and samples can be downloaded at:
http://luaforge.net/frs/?group_id=89.
The CVS can be browsed at: http://luaforge.net/scm/?group_id=89.

If you want us to develop a specific feature for the toolkit, Tecgraf is available for partnerships and cooperation. Please contact **tcg AT tecgraf.puc-rio.br**.

Lua documentation and resources can be found at http://www.lua.org/.

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 Clinio
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 Cristóvão Gomes Coelho
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.

Fullscreen support in Motif copied from wxWidgets. Their license is as flexible as the Tecgraf Library License, you can find it here: http://www.wxwidgets.org/newlicen.htm.

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 ~1.1Mb) and Windows HTML Help (iup.chm ~1.5Mb) 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

All the products under this license are free software: they can be used for both academic and commercial purposes at absolutely no cost. There are no royalties or GNU-like "copyleft" restrictions. They are licensed under the terms of the MIT license reproduced below, and so are compatible with GPL and also qualifies as Open Source software. They are not in the public domain, Tecgraf and Petrobras keep their copyright. The legal details are below.

The spirit of this license is that you are free to use the libraries for any purpose at no cost without having to ask us. The only requirement is that if you do use them, 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 libraries are 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.

The Tecgraf products under this license are: IUP, CD and IM.

IUP Download

The main download site is the **LuaForge** site available at:

http://luaforge.net/project/showfiles.php?group_id=89

When LuaForge is offline, the **Tecgraf Download Center** is activated to offer a mirror site.

http://www.tecgraf.puc-rio.br/download

Before downloading any precompiled binaries, you should read before the Tecgraf Library Download Tips.

Some other files are available directly at the **IUP** download folder:

http://www.tecgraf.puc-rio.br/iup/download/

History of Changes

Version 2.4 (12/Dec/2005)

General

- New attribute ZORDER to change the zorder of any control or dialog.
- New 3STATE attribute for IupToggle to enable a three state text toggle.
- Reviewed and improved the creation of controls, so they can be added to an already created dialog.
- Reviewed and improved the natural size estimation for each standard controls. The estimation now is the same for Windows and Motif with some minor differences for border and scrollbar sizes. All the controls can have sizes bigger or smaller than the natural size using SIZE or RASTERSIZE attributes (natural size is the size of the control that fits all of its contents).
- Improved FULLSCREEN IupDialog attribute in Windows and Motif, so the application can set fullscreen and then restore to normal state any time.
- New attribute FLAT for IupButton to create a button with mouse over activation (Windows and Motif).
- New MULTISELECT_CB callback for IupList. It can replace the action callback for multiple selection lists.
- Fixed names of headers, initialization functions and libraries that did not have the "iup" prefix. Headers "iupolecontrol.h", "luacontrols.h" and "luagl.h" changed to "iupole.h", "iupluacontrols.h" and "iupluagl.h". Private headers and declarations removed from "iup/include" folder. Functions controlslua_open, gllua_open and iupluaim_open changed to iupcontrolslua_open, iupgllua_open and iupimlua_open.
- New documentation of the IupOleControl control, including a sample and Lua bindings. Thanks to Vinicius Almendra.

- New function IupRefresh to update the size and layout of controls after changing size attributes.

- Exported the internal functions: IupZboxv, IupHboxv, IupVboxv and IupMenuv.

- Fixed several memory leaks. Thanks to Visual Leak Detector.

- IupView application can now save imagens in C source code format.

- New additional library with several pre-defined images for buttons and labels. See IupImageLib.

- Optimization flags now are ON when building the library in all platforms.

- Now all the predefined dialogs consult the global attribute IUP_ICON.
- Missing key definitions: K_sDEL and K_sINS. This prevented the Del key to work when CAPSLOCK was active in some controls.
- Changed IUP_QUIET environment variable now default is YES.

Windows

- Support for MDI (Multiple Document Interface). See IupDialog documentation.
- Fixed IupLabel with IMAGE with invalid focus.
- New SUNKEN attribute for IupFrame.

- Fixed appearance of IupLabel with IMAGE when ACTIVE=NO.
- Fixed initial value in the IupList when EDITBOX=YES.
- Now it is not necessary anymore to use the "iup.rc" file for the HAND cursor. It is now build in.
- New value for PLACEMENT attribute, FULL to position the client area of the dialog in fullscreen.
- IupButton and IupToggle with images using Windows XP Visual Styles now uses a styled border. See IupButton documentation for samples.
- Missing documentation of ENTERWINDOW_CB and LEAVEWINDOW_CB for IupButton.
- Fixed button draw with BGCOLOR and empty text.
- New COMPOSITED attribute to create a window with an automatic double buffer for all controls.
- New LAYERED and LAYERALPHA attributes to set and configure layered windows using transparency.
- Fixed image offset in IupButton.
- Fixed invalid redraw for IupLabel using an IupImage when inside a IupTabs or IupSbox.
- Added an "ifndef IUP_NO_ABNT" enclosing the ABNT keyboard management so it will be easier to ignore this code from the makefile.
- Default FONT in Windows XP is now the Tahoma font.
- BGCOLOR for canvas was not being updated correctly when changed after canvas creation.

Motif

- SHOWDROPDOWN now works also in Motif.

- Removed horizontal scrollbar parameter from simple IupList (DROPDOWN=NO and EDITBOX=NO) to made it compatible with the other lists (including the simple IupList in Windows).

- Fixed KILLFOCUS_CB and GETFOCUS_CB for IupList with DROPDOWN=YES or EDITBOX=YES.

- Fixed invalid IupList resize when DROPDOW=Yes after inserting elements in the list.

- New BACKINGSTORE IupCanvas attribute so the backing store can be disabled.

- Changed IupToggle with IMAGE and IMPRESS to behave like in Windows, where the button border is always shown.

- Fixed error in menu item initialization.

IupControls

- IMPORTANT: for best results CD version 4.4 should be used.
- Fixed IupSpin keyboard response and mouse press & hold response.
- New MULTISELECTION_CB callback for IupTree.
- New IupCells control. It is an application controlled matrix. More simple and faster than IupMatrix. Can also span cells. Thanks to André Clinio.
- New IupCbox control for concrete layout positioning.

- Fixed IupTabs tab activation using mouse. It could activate a different tab using button press in one tab and button release in another tab.

- Fixed spin buttons were not calling the user callback in IupGetParam.

- Fixed IupVal non effective increment using keyboard when at minimum value.

- Fixed invalid IupSetAttribute for scrollbar parameters in IupTree that affects navigation of two or more trees in the same application.

- Fixed keyboard usage when CAPSLOCK is active for IupVal, IupTabs and IupDial.

- New functions iupMaskRemove and iupmaskMatRemove to remove the iupMask from a control.

- New RENAME action attribute for the IupTree.

- New attribute TABORIENTATION to change the tab text orientation. The active tab text is now bold.

- Changed CARET and SELECTION attributes of the IupTree when using an in-place rename text box, to RENAMECARET and RENAMESELECTION. This will avoid conflict with the SELECTION_CB callback in IupLua3.

IupMatrix

- Redefined REDRAW policy to a more precise and effective one. No redraw is done when the application sets cell, line or column graphics attributes attributes: **`0:0`**, **`0:C`**, **`L:0`**, **`L:C`**, **`ALIGNMENTn`**, **`BGCOLORL:*`**, **`BGCOLOR*:C`**, **`BGCOLORL:C`**, **`FGCOLORL:*`**, **`FGCOLOR*:C`**, **`FGCOLORL:C`**, **`FONTL:*`**, **`FONT*:C`**, **`FONTL:C`**. Global and size attributes always automatically redraw the matrix.

- Improved double click editing in Motif. Since OpenMotif 2.2.3 the double click to edit the cell works fine. For previous version there is still a workaround to show the controls and the need to click again in the control so it get the focus.

- All the edition mode code were rewritten and reorganized in a separated module. Any old code was removed and cleaned.

- Small change in focus feedback, its area was reduced to two pixels in each cell border.

- Cell focus management code reorganized to a more simple and efficient version.

- New SORTSIGN*C* attribute to show a sort sign (up or down arrow) in the column *C* title.

- New drawing in double buffer mode to minimize flicker.

- Fixed dropdown feedback drawing.

- Fixed focus feedback after double click editing.

- The alignment of the text in a cell with a dropdown feedback now considers the horizontal space occupied by the feedback.

- The DRAW_CB callback drawing area now does not includes the focus feedback area if HIDEFOCUS=NO (the default).

- NUMCOL_VISIBLE and NUMLIN_VISIBLE now when retrieved returns the current number of visible lines.

- Fixed problem after trying to edit a non editable cell the focus gets lost.

- Reviewed documentation and behavior of marks.

IupLua

- IupLua5 source code is now 100% compatible with Lua 5.1.

- The iuplua binding and all its libraries can now be dinamically loaded using "require" in Lua 5. IupOpen will be automatically called.

- iupkey_open can now be called from Lua 5, using iup.key_open.

- New IupGetParam binding.

- Changed the keys definitions (K_*) in Lua so now they are exactly the same as the definitions in C.

- Fixed invalid IupGetAllNames in IupLua5. Fixed missing IupGetAllNames binding in IupLua5.

- Fixed IupTree EXECUTELEAF_CB callback in IupLua5. It was expecting an invalid extra parameter.

- Fixed error in IupTabs memory initialization in IupLua5.
- Fixed missing IupGetText binding.
- Fixed missing pre-defined masks for iupMask.
- Fixed missing isxkey macro binding.
- Fixed missing callback scroll_cb in IupLua3.
- Fixed missing IupVersion documentation and binding.
- Fixed IupSetGlobal and IupStoreGlobal in IupLua5.

Version 2.3.1 (18/Apr/2005)

General

- New support for 64-bits Linux.
- New global attribute DLGBGCOLOR.
- Changed the KEYPRESS_CB and K_ANY callback are now compatible with Portuguese Brazilian ABNT keyboard layout in Windows and Linux.
- Changed key names **K_quoteright** and **K_quoteleft** renamed to **K_apostrophe** and **K_grave**, but there are backward compatible defines.
- Fixed IupOpen/IupClose for correct initialization/de-initialization.
- Fixed IupGetGlobal to retreive first from the driver.
- Fixed IupDestroy for correct memory deallocation.
- Fixed IupLoadImage to include BGCOLOR information. New function IupSaveImage.
- New Guide / C++ Usage section in the documentation, with additional C++ wrappers contributed by some users. Thanks to Danny Reinholds, Sergio Maffra and Frederico Abraham.

Windows

- Fixed K_ANY duplicate calls for some keys.
- Fixed popup menu bug. Sometimes when selecting an item the callback was not called.
- Changed IupText and IupMultiline now can have the ALIGNMENT attribute.

Motif

- Fixed use of variable parameter arguments in Motif calls to correct 64-bits compatibility.
- Fixed some small bugs in IupDestroy. GETFOCUS_CB callbacks were called during dialog destroy. Menu bars were incorrectly destroyed.

IupControls

- Changed IupGetParam now uses only the number of lines to determine the number of parameters. The last 0 is not necessary anymore.
- Fixed bug in IupColorBrowser destroy.
- Fixed IupTree initialization for LED usage.
- New IupTree feature to rename a node in place.
- New IupColorbar control. It is a palette of colors to allow the selection of primary and secondary colors. Thanks to

André Clinio.

IupGLCanvas

- New function IupGLIsCurrent.

IupLua

- Fixed callbacks for IupDial in IupLua5.

IupView

- Fixed data initialization in Motif.
- New menu items to save images in individual LED and Lua text files, and in Windows ICON files.
- New menu item to load an image using IM.

Version 2.3 (16/Mar/2005)

General

- Download, Discussion List, Submission of Bugs, Support Requests and Feature Requests, are now available thanks to LuaForge site.
- New organization of the documentation.
- New MacOS X libraries using OpenMotif and gcc.
- New CARET_CB callback for the IupText, IupMultiline and IupList controls. It is called every time the caret changes its position.

Windows

- IMPORTANT: Now the canvas background color is only redrawn if the ACTION callback is not defined. When defined the application must draw all the canvas contents. This will optimize the redraw of canvas based controls and application canvases. The TRANSPARENT value for the BGCOLOR is not supported anymore.
- New attribute IMMARGIN to control the spacing between the border and the image in IupButton.
- Optimized the IupButton and IupLabel drawing when IMAGE is specified.
- Fixed incorrect stop for the IupTimer. Improved start and stop control.
- Flicker now is significantly reduced. CLIPCHILDREN=YES is now default. IupFrame background drawing optimized.
- New dialog attribute "CONTROL" that enable the embedding of the dialog inside another window. Used by LuaCOM to create OLE (ActiveX) controls implemented in Lua.
- New IupText attribute "PASSWORD" to hide the typed character.
- IUP is now compatible with Windows XP Visual Styles. See the Win32 driver documentation.

Motif

- Fixed invalid return value when retreiving the FONT attribute.
- Added backward compatibility code for Motif 1.2. Must edit makefile to add the file "src/mot/ComboBox1.c".

IupControls

- Missing support for IupList with EDITBOX=YES in iupMask.
- BGCOLOR for images were ignored in the IupTree.

- Now some matrix cell attributes are not inherited from parent. Like "L:C", "ALIGNMENT*", "FGCOLOR*", "BGCOLOR*", "FONT*", "WIDTH*" and "HEIGHT*", for optimization reasons.

- IupTree now uses double buffer for optimal drawing.
  To avoid flicker during resize in Windows, do not use it inside a IupFrame, and use CLIPCHILDREN=YES.

- New utility functions: IupTreeSetAttribute, IupTreeStoreAttribute IupTreeGetFloat, IupTreeSetfAttribute, IupTreeGetAttribute, IupTreeGetInt.

- New IupMatrix callback DRAW_CB to allow a custom drawing of the cell contents.

- New IupTree DRAGDROP_CB callback.

- New IupSpin and IupSpinbox utility functions.

IupLua

- Fixed ihandle_gettable in iuplua.lua when iupGetTable is nil when object is created in C.
  This affected the object returned by iup.LoadImage.
- Fixed Zbox children names initialization.
- Missing DROPFILES_CB callback management.
- Missing FGCOLOR_CB and BGCOLOR_CB callback management for the IupMatrix. The returned values order was inverted.
- Missing MAP_CB callback management for IupCanvas in IupLua3.

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 IupFileDlg 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 abssolute 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 bellow).
- 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_SBDRAGV and IUP_SBDRAGH were not documented.

- Callback MENUSELECT_CB changed to HIGHLIGHT_CB. Now implemented also in Motif.
- New menu callback MENUCLOSE_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 "HELPBUTTON" 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 comboox 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_SBDRAGV and IUP_SBDRAGH 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 IupFileDlg.
- 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 FGCOLOR 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 erros 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 infinit loop. The matrix now uses the keypress_cb instead of the k_any callback.

- Fixed empty selection in the dropdown list if the user press 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.

- BGCOLOR now works also for titles.

- FONT attribute now can be set/get just like BGCOLOR and FGCOLOR. But the cell size is calculated always from the matrix attribute IUP_FONT.

Tree

- Documentation reviewed and reorganized.
- CTRL and SHIFT accepts 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 access through their exact name in the C API. Mostly add sufix "_cb" to name (most common callbacks renamed for ex: getfocus_cb, killfocus_cb). Also some names were fix: 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.colorbrowser.

- Use LoadLibrary to load IUP from Lua.
- There was no stack pop in color processing loop fo 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 and invalid ID returns NULL
- Fixed attributes IUP_CTRL e 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 being 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 performace
- 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 "DRAWSIZE" that returns the drawing area of the canvas (in Windows we may have an addicional border included in "RASTERSIZE").

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 IupFileDlg 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 "`BGCOLOR`" 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).

History of Version 1.x

To Do

General

- Organize controls and driver initialization and management, improve the possibility of implementing new drivers.
- A gtk driver in Linux.
- A MacOS X native driver using Carbon.
- A wxWidgets driver?
- A tutorial section in the documentation. Add more complete samples.

- To allow `IupShow` after a `IupPopup.`
- To show a border for visual location of `VBOX`, `HBOX` and `FILL`. Can be a dialog attribute. Or most likely a function to display the layout.
- Change all comments in the source code to english. Add comments to the internal includes for Doxygen.
- Drag&Drop between controls or dialogs in the same application.
- Allow the functions `IupAppend` and `IupDetach` to be used for dynamic creation of menus, IupSbox, IupCbox and IupTabs.
- Buttons with image and text simultaneously.
- Support for RGBA images. Support for alpha in colors for images?

Motif

- Support for MDI.
- Reduce flicker when dialog is resized.
- Callback `SHOW_CB` is not called when the dialog is hidden because of `PARENTDIALOG`.
- Some 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.
- The menu does not inherit attributes from the dialog like in the Windows driver.
- Sometimes the control initialization is incomplete and its size is miscalculated. To solve this call IupRefresh (dialog). This will fix the sizes.

IupControls

- Move iupMask to the mail library using attributes instead of functions.
- A vertical IupGauge?
- IupSbox can be resized above the maximum size so some controls go to outside the dialog area at right or bottom. In fact thi is part of the dynamic layout default reposition of controls inside the dialog. See the IupRefresh function. The IUP layout does not have a maximum limit only a minimum.

IupMatrix

- When removing a line, if it has the focus an invalid call to enteritem_cb/leave_item_cb will occur for the removed cell.
- Lines and columns are not unmarked when clicking on the title.

IupTree

- Review IupTree redering. Improve render attributes like background color. Define minimum size based on tree nodes. New callback mode. Change internal list to real tree?
- Images with variable sizes for nodes.

IupLua

- Optimize callback calling in Lua 5. This affects some IupMatrix callbacks in Lua: VALUE_CB, FGCOLOR_CB and BGCOLOR_CB.

New Controls

- IupPlot to plot a XY graph
- A detachable toolbar?
- Image Listbox
- Grid Container (to distribute elements in a grid)
- RTF editor in Windows
- HTML viewer?

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*	1997-2004/05	3.52	C	Win, X, Mac	own	1	Small and interesting.
FOX	LGPL*	1997-2005/03	1.5.0	C++	Win, X	own	3+15	great look, license totaly free?
FLTK	LGPL*	1998-2004/11	1.1.6	C++	Win, X, Mac	own	3+16	was from Digital Domain. Easy to learn.
GTK+	LGPL*	1997-2005/03	2.6.4	C	Win, X	own	9	target for X-Windows, basis of GNOME, Windows is apart
Qt	GPL	1994-2004/08	3.3.3	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*	1992-2005/02	2.5.4	C++	Win, X, Mac	native	6+11	
IUP	MIT*	1994-2005/03	2.3	C	Win, X	native	2	

Table Last Update: March 2005

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 gray ones are not updated anymore or the development is very slow or needs a better organization.

FOX has a great look but the license can be restrictive in some cases.

FLTK promises 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 was target for X-Windows. Nowadays GTK 2 is a great free C toolkit. But some predefined dialogs could be the native ones, like the File Selection.

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 would be wxWidgets because of the native controls and its portability. But since version 2, GTK+ is a very strong option because it is in C and had its visual improved.

It is very hard to compare IUP with wxWidgets and Qt since they are much more than an Interface Toolkit. They are complete development platform that includes several secondary libraries. In IUP we focus only in Graphical User Interface.

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. Today IUP can be built in Mac OS X using X11 and Motif.
>
> Motif is still very important for non Linux systems, some of our applications run on old AIX, SGI and Sun, that only have Motif installed and we can not force the installation of other toolkits like GTK. On the other hand in Linux we should be using GTK instead of Motif.

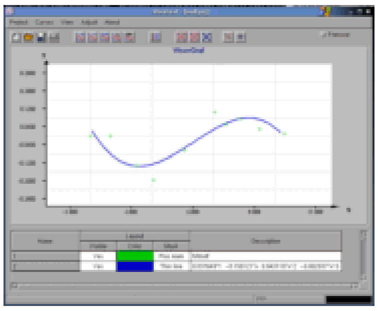
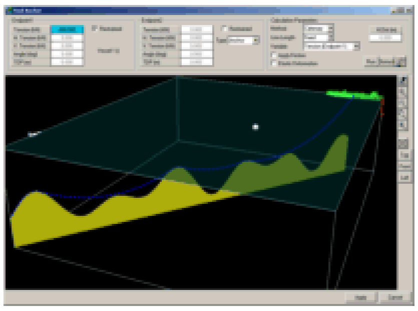
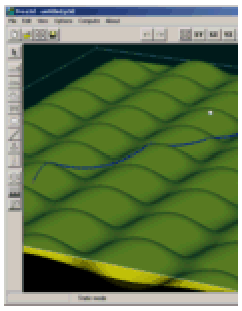
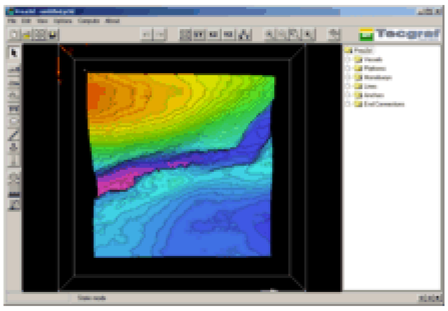
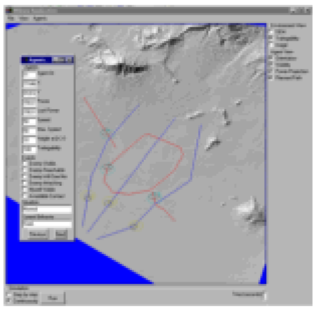
IUP is in C, is small and powerful. We have a small but very active team and we have many Tecgraf and foreign applications that today use IUP, collaborating for its evolution. Our objective is to surpass the Tecgraf needs, keeping backward compatibility and improving the internal code.

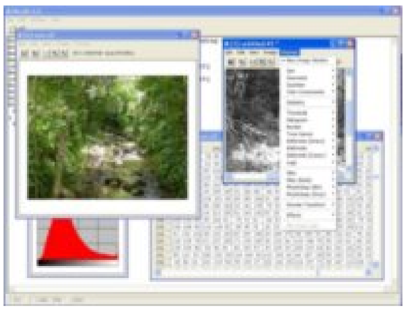
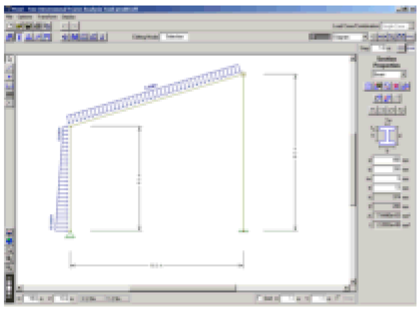
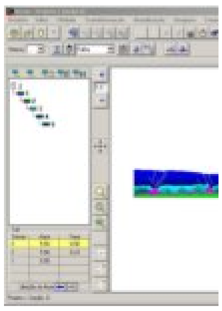
IUP does not have a wide localization feature, it only includes support for messages in English and Portuguese. And it does not have support for Unicode characters.

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

Screenshots

Click on the picture to enlarge image.

Guide

Getting Started

IUP has 4 important concepts that are implemented in a very different way from other toolkits.

First is the control creation timeline. When a control is created it is not immediately mapped to the native system. So some attributes will not work until the control is mapped. The mapping is done when the dialog is shown or manually calling **IupMap** for the dialog. You can not map a control without inserting it into a dialog.

Second is the attribute system. IUP has only a few functions because it uses string attributes to access the properties of each control. So get used to **IupSetAttribute** and **IupGetAttribute**, because you are going to use them a lot.

Third is the abstract layout positioning. IUP controls are never positioned in a specific (x,y) coordinate inside the dialog. The positioning is always calculated dynamically from the abstract layout hierarchy. So get used to the **IupFill**, **IupHbox** and **IupVbox** controls that allows you to position the controls in the dialog.

Fourth is the callback system. Because of the LED resource files IUP has an indirect form to associate a callback to a control. You must associate A C function with a name using **IupSetFunction**, and you must associate the callback attribute with that name using **IupSetAttribute**.

LED is the original IUP resource file which has been deprecated in favor of Lua files. But keep in mind that you can use IUP without using LED or Lua, using only the C API.

Building Applications

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.

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.

The generation of applications is highly dependent on each system, but at least the **iup.lib/libiup.a/libiup.so** library must be linked.

To use the Lua Binding, you need to link the program with the **iuplua.lib/libiuplua.a/libiuplua.so** library and with the **lua.lib/liblua.a/liblua.so** and **lualib.lib/liblualib.a/liblualib.so** libraries. IupLua is available for Lua 3.2 and Lua 5.0.

The download files list includes the Tecgraf/PUC-Rio Library Download Tips document, with a description of all the available binaries.

Windows

In Windows, you must link also with the libraries **ole32.lib** and **comctl32.lib** (provided with the compilers). 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.

There is also a guide on using the Dev-C++ IDE Project Options and Visual C++ IDE Project Properties.

Motif

In Motif, IUP uses the Motif (`Xm`), the Xtoolkit (`Xt`) and the 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 only). They must be listed after `Xt` and before `X11`. For instance:

```
-liup -lXm -lXpm -lXmu -lXt -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:

Standard	`-L/usr/lib -I/usr/include`
Linux	`-L/usr/X11R6/lib -I/usr/X11R6/include`
IRIX	`-L/usr/lib32 -I/usr/include/X11`
Solaris	`-L/usr/openwin/lib -I/usr/openwin/share/include/X11`
AIX	`-I/usr/include/Motif2.1`

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.

Multithread

User interface is usually not thread safe and IUP is not thread safe. The general recommendation when you want more than one thread is to build the application and the user interface in the main thread, and create secondary threads that communicates with the main thread to update the interface. The secondary threads should not directly update the interface.

Dynamic Loading

Although we have dynamic libraries we do not recommend the dynamic loading of the main IUP library. This is because it depends on Motif and X11, you will have to load these libraries first. So it is easier to build an base application that already includes X11, Motif and the main IUP library than trying to load them all. The IUP secondary libraries can be easily dynamic loaded.

Here is an example in Lua 5:

```
local init =
{
  {"cd.dll", nil},
  {"iupcontrols.dll", "IupControlsOpen"},
  {"iuplua5.dll", "iuplua_open"},
  {"iupluacontrols5.dll", "iupcontrolslua_open"},
}

local function DllExecute(i,v)
  local name = v[1]
  local func = v[2]
  if not func then func = "" end
  local fnc, err = loadlib(name, func)
  if v[2] then
    if fnc then
      fnc()
    else
      print(v[1]..": "..err)
    end
  end
end

table.foreach(init, DllExecute)
```

Building The Library

The easiest way to build the library is to install the Tecmake tool into your system. It is easy and helps a lot. The Tecmake configuration files (*.mak) available at the "src" folder are very easy to understand also.

Tecmake is a command line multi compiler build tool available at http://www.tecgraf.puc-rio.br/tecmake. Tecmake is used by all the Tecgraf libraries and many applications.

In **IUP**'s main directory, and in each source directory, there are files named *make_uname* (*make_uname.bat* in Windows) that build the libraries using **Tecmake**. To build the **IUP** libraries for Windows using Visual C 7.0 for example, just execute *make_uname.bat vc7* in the iup root folder.

But we also provide a stand alone makefile for Linux systems and a Visual Studio workspace with the respective projects. The stand alone makefile is created using Premake and a configuration file in lua called "premake.lua".

IUP runs on many different systems and interact with many different libraries such as Motif, OpenGL, Canvas Draw (CD) and Lua (3 and 5). You have to install some these libraries to use the IUP libraries. IUP standalone only depends on the Windows core libraries (alreay installed in the system) and on the Motif 2.x+X11-R6. In Linux you should use Open Motif 2.x. If you only have Motif 1.2 some features will be limited and you must add the file "src/mot/ComboBox1.c".

Using IUP in C++

IUP is a low level API, but at the same time a very simple and intuitive API. That's why it is implemented in C, to keep the API simple. But most of the actual IUP applications today use C++. To use C callbacks in C++ classes, you can declare the callbacks as static members or friend functions, and store the pointer "this" at the "Ihandle*" pointer as an user attribute. For example, you can create your dialog by inheriting from the following dialog.

```
class iupDialog
{
private:
  Ihandle *hDlg;
  int test;

  static int ResizeCB (Ihandle* self, int w, int h);
  friend int ShowCB(Ihandle *self, int mode);

public:
  iupDialog(Ihandle* child)
  {
    hDlg = IupDialog(child);
    IupSetAttribute(hDlg, "iupDialog", (char*)this);
    IupSetAttribute(hDlg, "RESIZE_CB", "iupDialogResizeCB");
    IupSetFunction("iupDialogResizeCB", (Icallback)ResizeCB);
    IupSetAttribute(hDlg, "SHOW_CB", "iupDialogShowCB");
    IupSetFunction("iupDialogShowCB", (Icallback)ShowCB);
  }

  void ShowXY(int x, int y) { IupShowXY(hDlg, x, y); }

protected:

  // implement this to use your own callbacks
  virtual void Show(int mode) {};
  virtual void Resize (int w, int h){};
};

int iupDialog::ResizeCB(Ihandle *self, int w, int h)
{
  iupDialog *d = (iupDialog*)IupGetAttribute(self, "iupDialog");
  d->test = 1; // private members can be accessed in private static members
  d->Resize(w, h);
  return IUP_DEFAULT;
}

int ShowCB(Ihandle *self, int mode)
{
  iupDialog *d = (iupDialog*)IupGetAttribute(self, "iupDialog");
  d->test = 1; // private members can be accessed in private friend functions
  d->Show(mode);
  return IUP_DEFAULT;
}
```

This is just one possibility on how to write a wrapper class around IUP functions. Some users contributed with C++ wrappers:

RSSGui by Danny Reinhold. Described by his words:

- It works fine with the C++ STL and doesn't define a set of own string, list, vector etc. classes like many other toolkits do (for example wxWidgets).

- It has a really simple event handling mechanism that is much simpler than the system that is used in MFC or in wxWidgets and that doesn't require a preprocessor like Qt. (It could be done type safe using templates as in a signal and slot library but the current way is really, really simple to understand and to write.)
- It has a Widget type for creating wizards.
- It is not complete, some things are missing. It was tested only on the Windows platform.

For more see the documentation page of RSSGui.

IupTreeUtil by Sergio Maffra and Frederico Abraham. It is an utility wrapper for the IupTree control.

The code available here uses the same license terms of the IUP license.

C++BuilderX IDE Project Options Guide

http://www.borland.com/products/downloads/download_cbuilderx.html

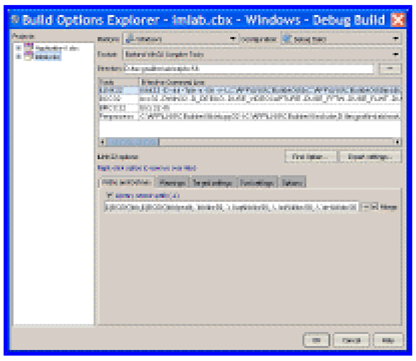
Borland C++ Builder X is an Integrated Development Environment (IDE) for Java and C/C++ languages. It can use several sets of compilers, including the Borland command line compilers version 5.6.

It also has many features, with the Borland name behind it. Its download is free. To use IUP with C++BuilderX you will need to download the "bc56" binaries in the download page.

After unpacking the file in your conputer, you must create a new Project for a "New GUI Application" and configure your Project Options. In the Project Build Options Explorer dialog there are 3 important places:

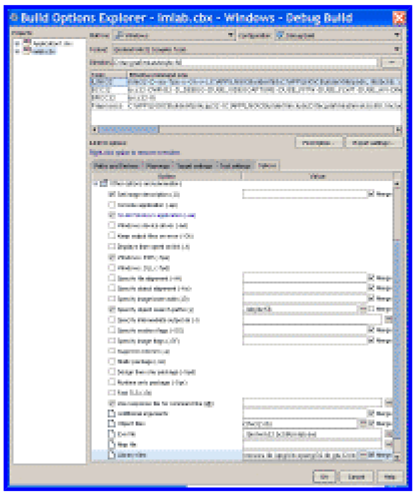
- In the Tools list, click on ILINK32. Then bellow select the Path and Defines tab - there you are going to add the path of the libraries you use, for example:

```
.\lib\bc56;..\..\iup\lib\bc56;..\..\cd\lib\bc56;..\..\im\lib\bc56
```



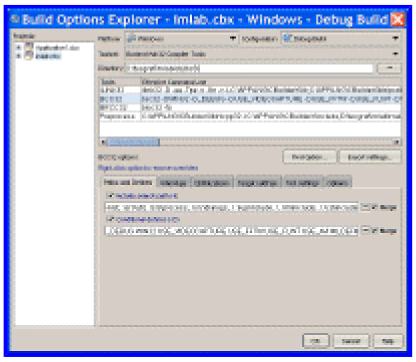
- In the same ILINK32 options, in the tab Options, select Other Options and Parameters, then Library files - there you are going to list the libraries, for example:

```
cw32.lib import32.lib vfw32.lib comctl32.lib iup.lib iupcontrols.lib cd.lib cdiu
```



- In the Tools list, click on IBCC32. Then bellow select the Path and Defines tab - there you are going to list the include path, for example:

  ```
  ..\include;..\..\iup\include;..\..\cd\include;..\..\im\include
  ```



Dev-C++ IDE Project Options Guide

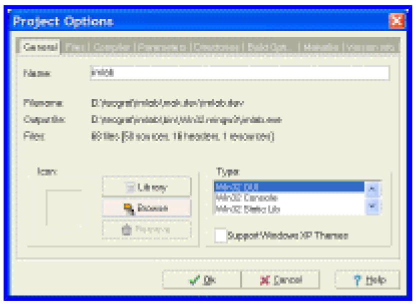
http://www.bloodshed.net/devcpp.html

"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 "mingw3" binaries in the download page.

After unpacking the file in your conputer, you must create a new Project and 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".



Name: imlab

Filename: D:\tecgraf\imlab\mak.dev\imlab.dev

Output file: D:\tecgraf\imlab\bin\Win32.mingw3\imlab.exe

Files: 88 files [58 sources, 18 headers, 1 resources]

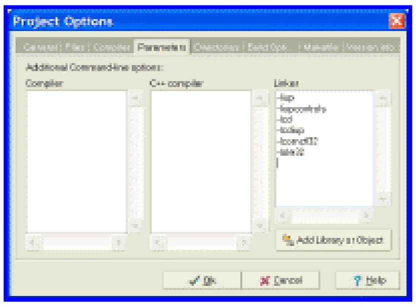
Type: Win32 DLL / Win32 Console / Win32 Static Lib

Support Windows XP Themes

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

  ```
  -liup
  -liupcontrols
  -lcd
  -lcdiup
  -lcomctl32
  -lole32
  -lgdi32 (if Win32 Console)
  -lcomdlg32 (if Win32 Console)
  ```

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



Additional Command-line options:

Compiler:

C++ compiler:

Linker:

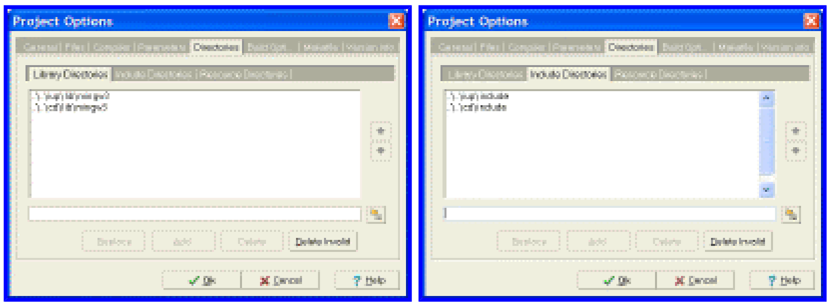
```
-liup
-liupcontrols
-lcd
-lcdiup
-lcomctl32
-lole32
```

- 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:

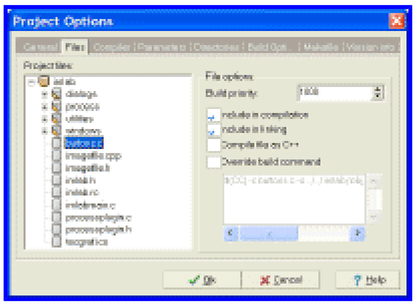
  ```
  ..\..\iup\include
  ..\..\cd\include
  or
  c:\tecgraf\iup\include
  c:\tecgraf\cd\include
  ```



```
..\..\iup\lib\mingw3
..\..\cd\lib\mingw3
```

```
..\..\iup\include
..\..\cd\include
```

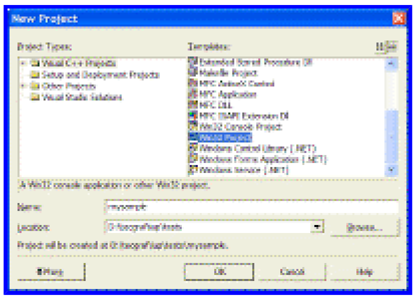
In some cases the IDE may force the compilation of C files as C++. If do not want that then uncheck the option in the settings for each file. Still in the Project Options dialog, in the Files tab, select the file and uncheck "Compile File as C++".



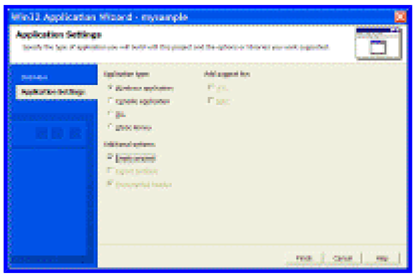
Visual C++ IDE Project Properties Guide

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".

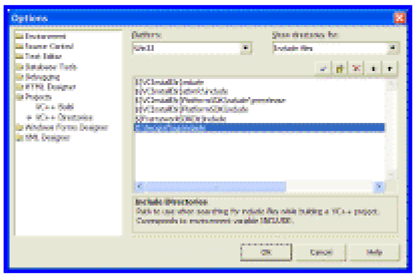


You can also create a "Console application", and whenever you execute your application a text console will also be displayed. But this is a very useful situation so you can the use standard C printf function to display textual information for debugging purposes.

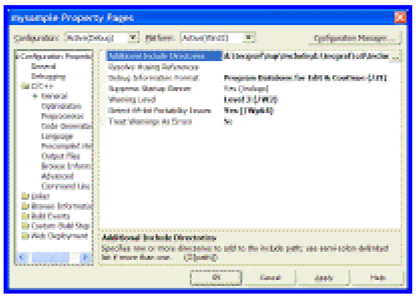
Then add your files in the menu "Project / Add New Item" or "Project / Add Existing Item".

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.

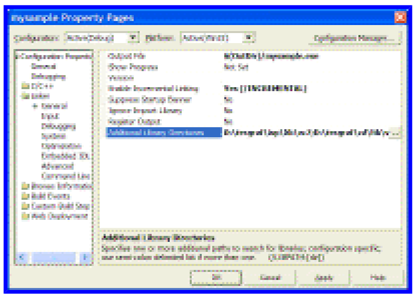
One is in the menu "Tools / Options", then select "Project / Visual C++ Directories". Select "Include Files" or "Library Files" in "Show directories for:". In this dialog you will configure parameters that will affect all the projects you open.



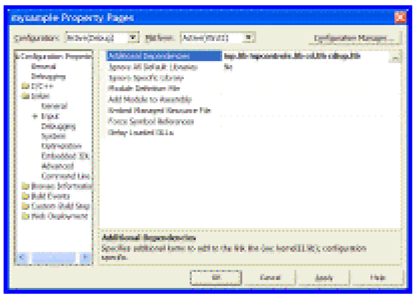
Or you can configure the parameters only for the project you created. In this case go the menu "Project / Properties". To configure the include files location select "C/C++ / General" in the left tree, then write the list of folders separated by ";" in "Additional Include Directories".



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".



Now you must add the libraries you use. In this same dialog, select "Linker / Input" in the left tree, then write the list of files separated by spaces " " in "Additional Dependencies".

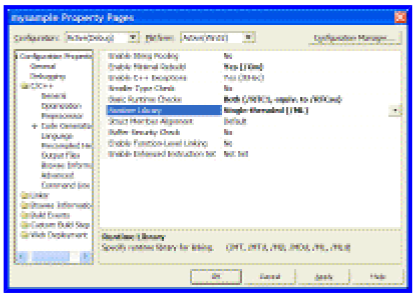


In this sample configuration the project is using the additional library of Controls that uses the CD library, also available at the download page.

When you build the project the Visual C++ linker will display the following message:

```
    LINK : warning LNK4098: defaultlib 'LIBC' conflicts with use of other libs; use
```

The default configuration use the C run time library with debug information, and IUP uses the C run time library without debug information. You can simply ignore this warning or change your project properties in "C/C++ / Code Generation" in the left tree, then change "Run Time Library" to "Single Threaded (/ML)".



mysample Property Pages

Configuration: Active(Debug) Platform: Active(Win32) Configuration Manager...

Enable String Pooling	No
Enable Minimal Rebuild	Yes (/Gm)
Enable C++ Exceptions	Yes (/EHsc)
Smaller Type Check	No
Basic Runtime Checks	Both (/RTC1, equiv. to /RTCsu)
Runtime Library	Single-threaded (/ML)
Struct Member Alignment	Default
Buffer Security Check	No
Enable Function-Level Linking	No
Enable Enhanced Instruction Set	Not Set

Runtime Library

Specify runtime library for linking. (/MT, /MTd, /MD, /MDd, /ML, /MLd)

If you want to use multithreading then you must use the DLL version of the IUP libraries. They are built with the "Multi-threaded DLL (/MD)" option. Or you must rebuild the libraries with your own parameters.

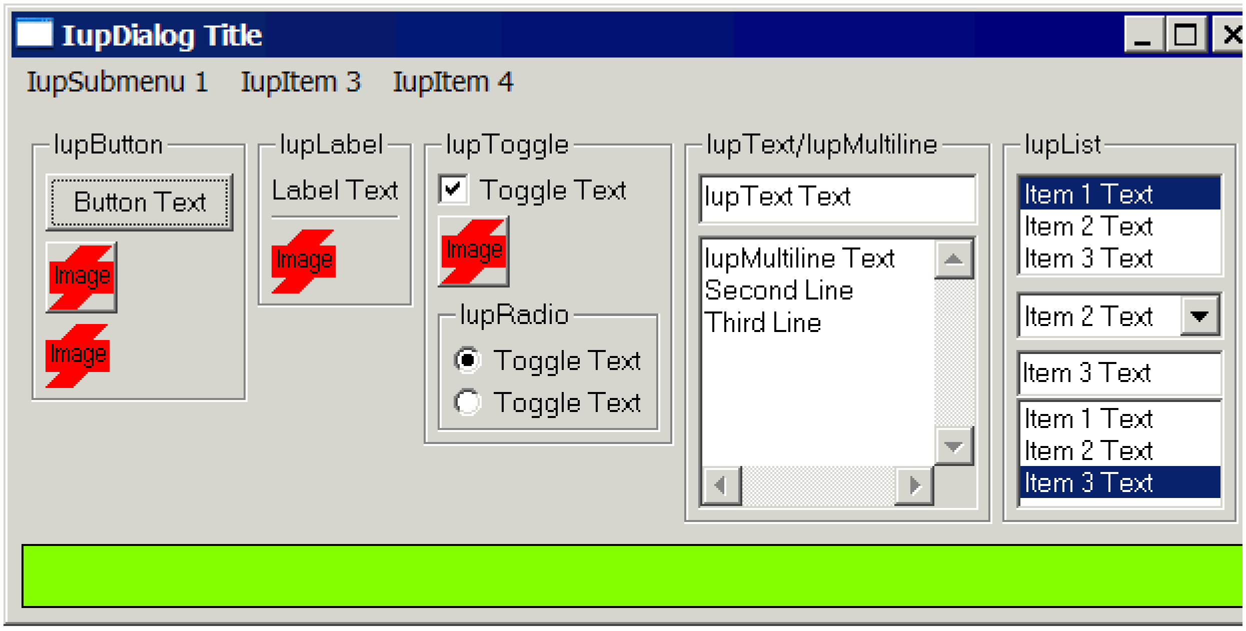
Complete Samples

Standard Controls

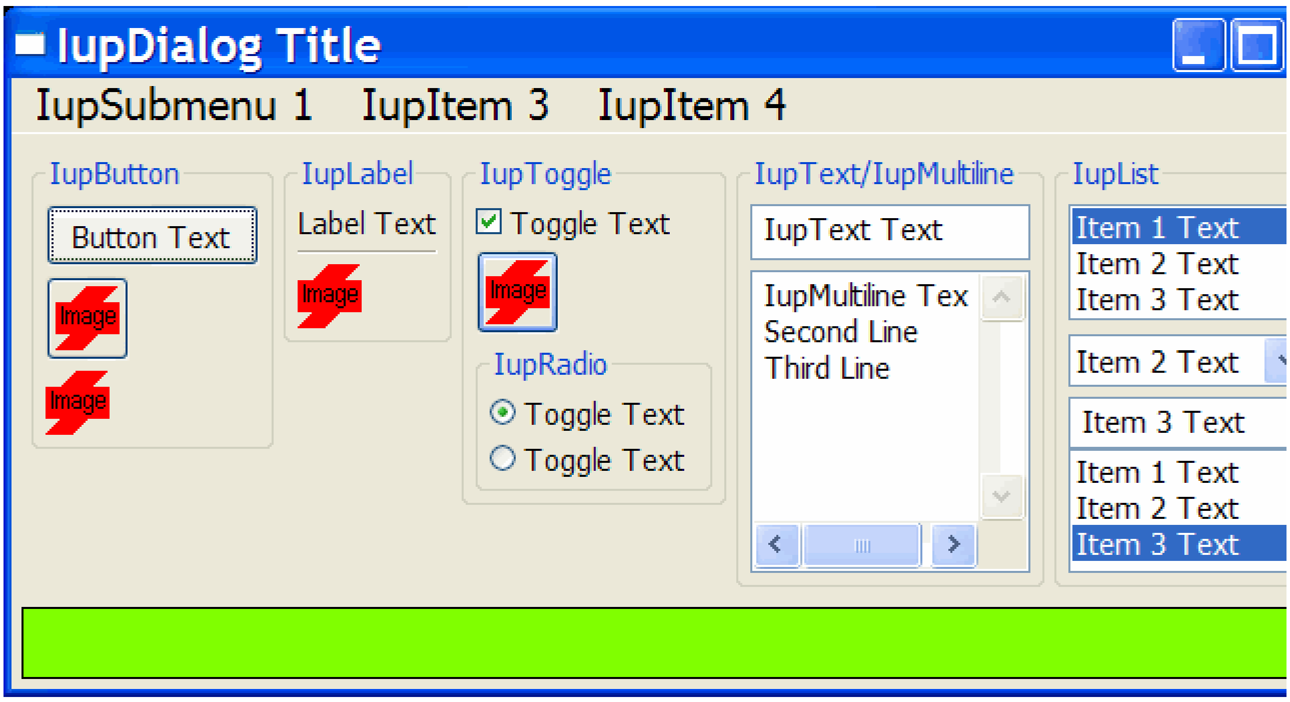
The following example creates a dialog with virtually all of IUP's elements as well as some variations of them, with some attributes changed. The same example is implemented in C, LED and Lua. Both screens presented are from the same example, one in Windows 95 and the other in IRIX. The C code is ready to compile. The LED code can be loaded and viewed in the **IupView** application. The Lua code can be loaded and executed in the **IupLua** standalone application.

in C	**in LED**	**in IupLua3**	**in IupLua5**
sample.c	sample.led	sample.lua	sample.lua5

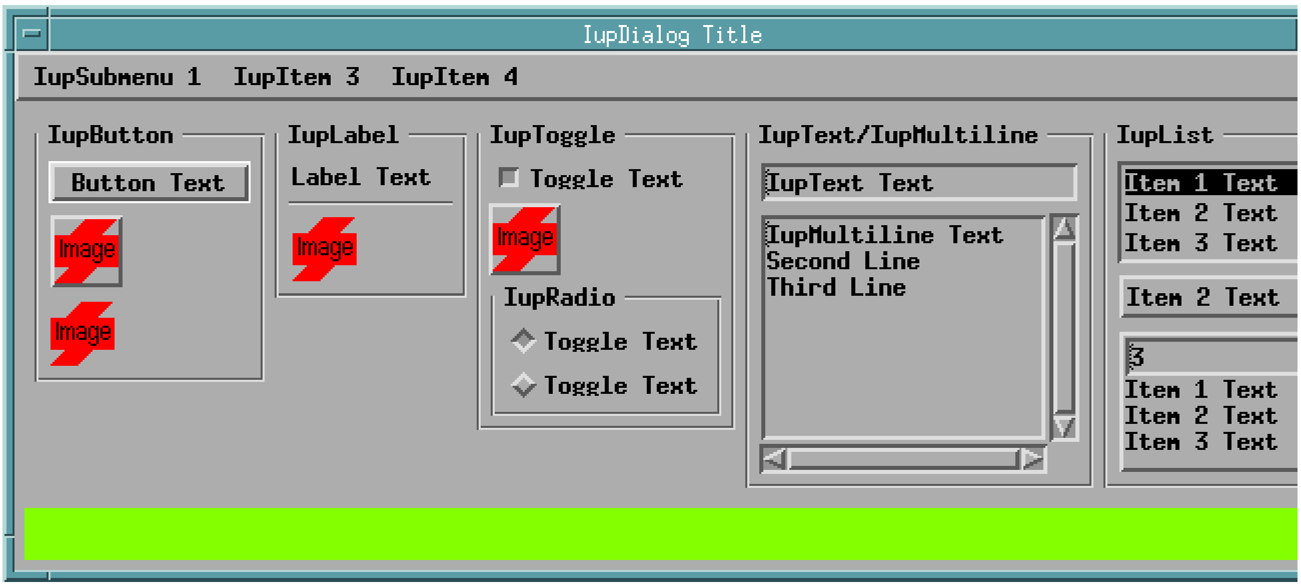
The Result in Windows



The Result in Windows XP

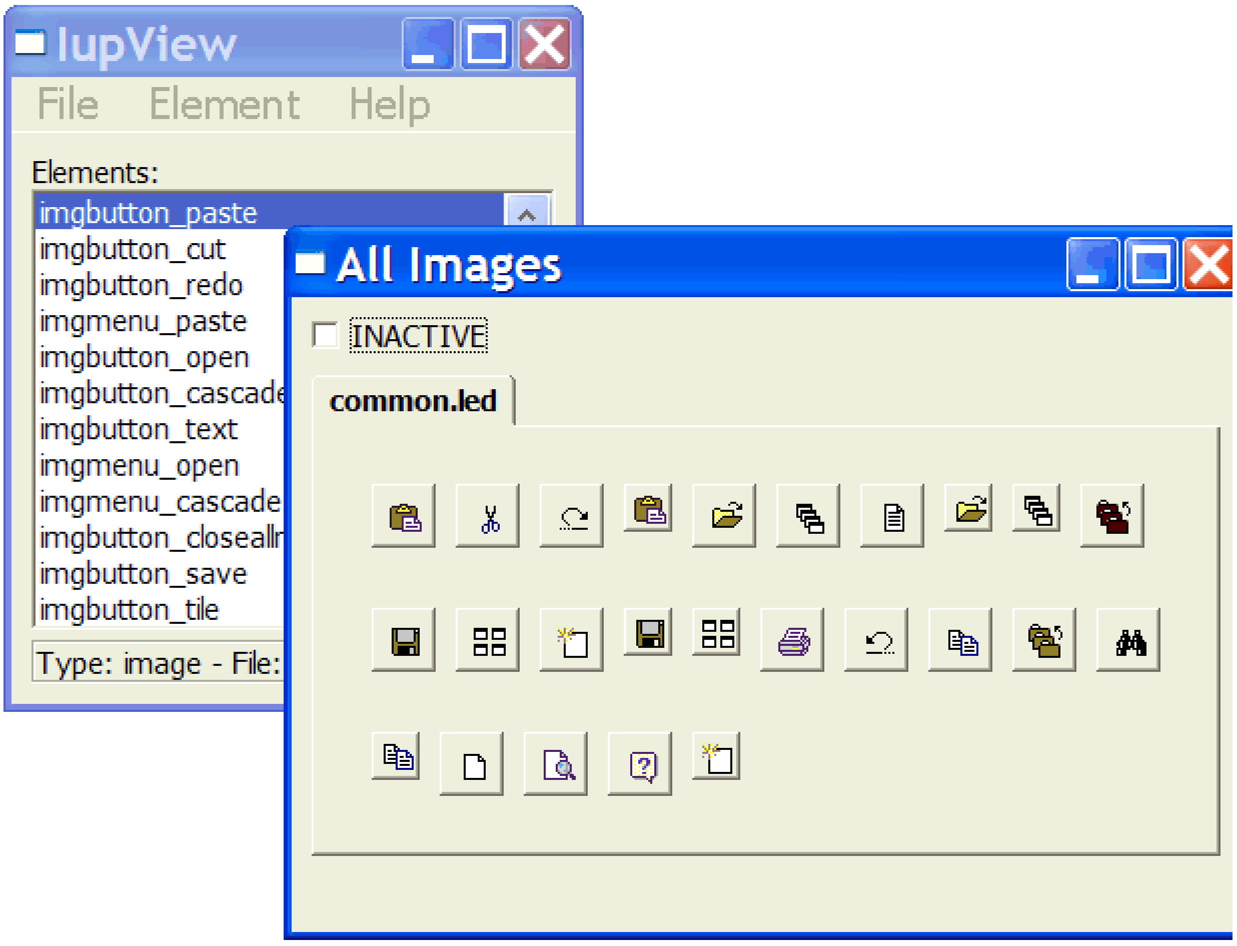


The Result in Motif

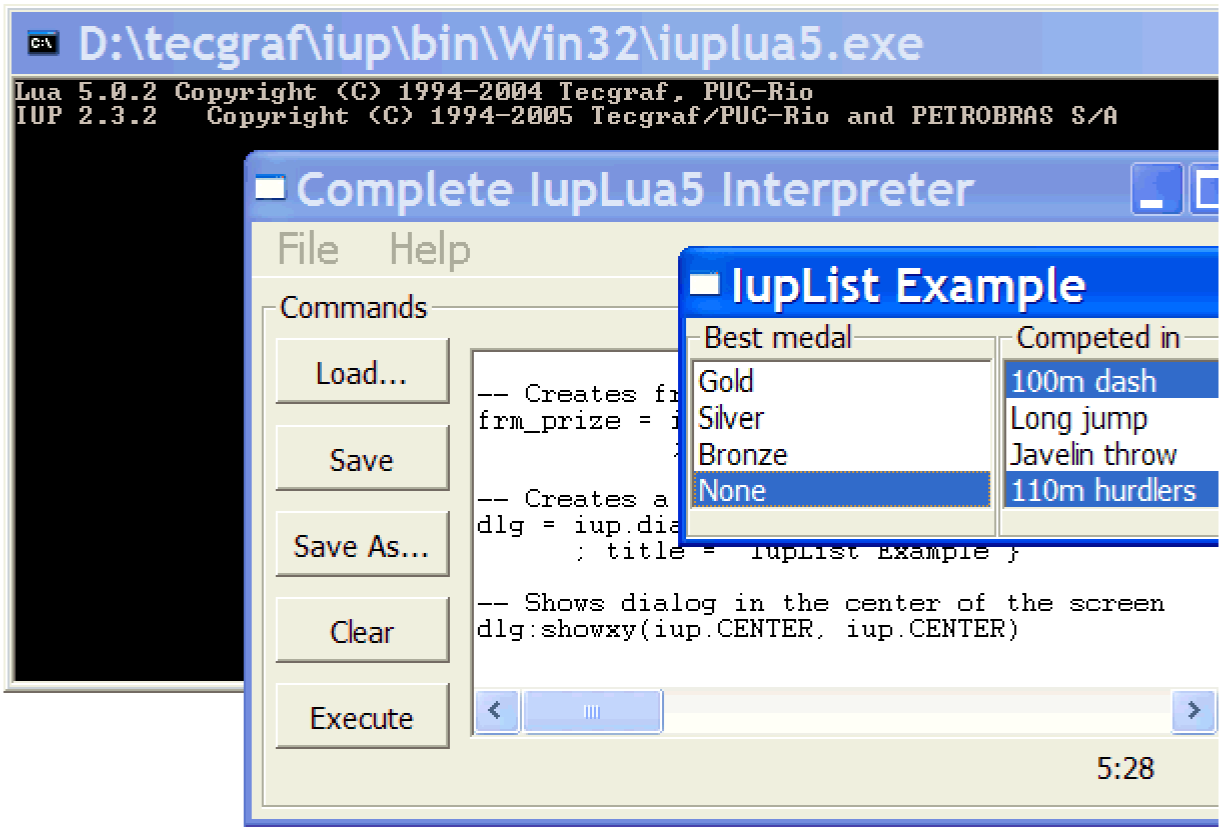


IupView and IupLua Executables

The **IupView** application can be used to test LED files, load and save images for IupImage or for ICONS, display all images and test them when disabled, display dialogs and popup menus.



The **IupLua** application can load and execute Lua scripts using the IupLua binding. Lua print calls are output in the console.



```
Lua 5.0.2 Copyright (C) 1994-2004 Tecgraf, PUC-Rio
IUP 2.3.2   Copyright (C) 1994-2005 Tecgraf/PUC-Rio and PETROBRAS S/A
```

```
-- Creates fr
frm_prize = i
            }

-- Creates a
dlg = iup.dia
      ; title = "IupList Example" }

-- Shows dialog in the center of the screen
dlg:showxy(iup.CENTER, iup.CENTER)
```

For the **IupView** and **IupLua** applications see the distribution files, source code and pre-compiled binaries are available at the Download.

All Samples

The IUP samples are spread in the documentation. Each control, dialog, menu has its own set of examples in C, LED and Lua.

You can browse the examples here.

External Samples

The CD and IM libraries have samples that use IUP, check in their documentation.

Some freely available applications also use IUP:

IMLAB - Image Processing Laboratory

EdPatt - Pattern Editor

Ftool - Two-dimensional Frame Analysis Tool

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 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 Programing Interface**) was created for this purpose: it was called CPI (**Controls Programing 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.

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.)

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.

```
Iclass *iupCpiCreateNewClass(char *name, char *format);
```

- **name**: Stores the name given to the control. This name allows the control to be used in LED.
- **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 *`)
`N`	from this character on, a list of control-instance names or a list of 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 *`)

Next step is to replace, if necessary, some of the control's CPI methods. This is done through function `iupCpiSetClassMethod`, which receives the control's class as an argument and the pointer to the new method. The Function `iupCpiCreateNewClass` will set the class with default methods, which provide the control a default behavior.

This initialization function should be named `IupXxxOpen`, where *Xxx* is the name of the control. The control will be automatically unregistered, but any other memory allocated in `IupXxxOpen` should be dealocated in a `IupXxxClose`.

Example (class creation for a control named Xxx):

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

static Ihandle *XxxCreate(...)
{
  ...
}

void IupXxxOpen(void)
{
  Iclass *new_class = iupCpiCreateNewClass("xxx","n");

  iupCpiSetClassMethod(new_class, ICPI_CREATE, DialCreate);
}
```

Control Instances

The new control should make a function available whose name would be `IupXxx`, where *Xxx* 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, then 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 the ICPI_CREATE method.

Example 1:

```
Ihandle* IupXxx(void)
{
  return IupCreate("xxx");
}
```

Example 2:

```
Ihandle* IupXxx (Ihandle* first,...)
{
  Ihandle **params = NULL;
```

```
    Ihandle *elem;
    va_list arg;

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

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

      va_start (arg, first);
      params[0] = first;

      for (i = 1; i < n; i++)
      {
        params[i] = va_arg(arg, Ihandle *);
      }

      params[n] = NULL;
      va_end (arg);
    }

    elem = IupCreatev("xxx", params);
    if (params) free(params);
    return elem;
  }
```

CPI Methods

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 methods 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`.

In several occasions, the default behavior defined by the CPI is not adequate for the new control's implementation. In this case, a new function must be set, providing the desired implementation for such method. If convenient, this new function can call the function implementing the method's default behavior, either before or after performing the specific treatment of the new control. Generally, a method that will always be redefined to a new control is the one in charge of creating instances of this control. To redefine (replace) a control method, function `iupCpiSetClassMethod` must be used. It receives as parameters the values described below:

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

Method & Default Method	**Description**

`ICPI_CREATE`
 	
`iupCpiDefaultCreate`
	This method is called by IUP when an instance for such control must be created. When this function is called, IUP already has a registered `Iclass` for the control instance (represented by the `ic` parameter). The array parameter is an array with the required attributes, specified in the call to the control creation function. The default function creates an `IupCanvas` control.
	
	Prototype:
	`Ihandle *(*create) (Iclass* ic, void** array);`

`ICPI_DESTROY`	This method is called by IUP when the control is about to be destroyed. If necessary, this method can be redefined to dispose of

`iupCpiDefaultDestroy`	structures maintained by the control. Prototype: `void (*destroy) (Ihandle* self);`

`ICPI_MAP`
 	
`iupCpiDefaultMap`
	This method is called by IUP to map the control in the native system. The parent parameter indicates of which parent the control is a child (This parent can either be a dialog or any other control). If you do not directly map the control to the native system then `iupCpiDefaultMap` must be called to let IUP map the control.
	
	Prototype:
	`void (*map) (Ihandle* self, Ihandle* parent);`

`ICPI_UNMAP`
 	
`iupCpiDefaultUnmap`
	This method is called by IUP to "destroy" the control's mapping in the native system without removing the control from the control hierarchy of a IUP dialog.
	
	Prototype:
	`void (*unmap) (Ihandle* self);`

`ICPI_SETNATURALSIZE`
 	
`iupCpiDefaultSetNaturalSize`
	This method is called by IUP for the control to specify its natural size. For such, this function must call functions `iupSetNaturalWidth` and `iupSetNaturalHeight`. Its implementation might call the `ICPI_GETSIZE` method to compute the natural size of the control. This function must return the same value returned by the `ICPI_GETSIZE` method.
	
	Prototype:
	`int (*setnaturalsize) (Ihandle* self);`

`ICPI_SETCURRENTSIZE`
 	
`iupCpiDefaultSetCurrentSize`
	This method is called by IUP for the control to change its current size. For such, this function must call functions `iupSetCurrentWidth` and `iupSetCurrentHeight`. Parameters represent the maximum size the control can have in pixels.
	
	Prototype:
	`void (*setcurrentsize) (Ihandle* self, int max_w, int max_h);`

`ICPI_SETPOSITION`
 	
`iupCpiDefaultSetPosition`
	This method is called by IUP for the control to define its position inside the parent window. Parameters `x` and `y` represent the position in pixels (upper left corner of the control) the control must have, computed by IUP. The default behavior for this method need only be changed if the control has sub-controls.
	
	Prototype:
	`void (*setposition) (Ihandle* self, int x, int y);`

`ICPI_SETATTR`
 	
`iupCpiDefaultSetAttr`
	This method is called to provide a new value to a given control attribute. When this method is called, the attribute's value is already stored in the control's attribute environment.
	
	Prototype:
	`void (*setattr) (Ihandle* self, char* attr, char* value);`

ICPI_GETATTR iupCpiDefaultGetAttr	This method is called by IUP to verify the value of a control attribute, determined by parameter attr. This method is called before IUP verifies the control's attribute environment. If this method returns `null`, IUP verifies the control's attribute environment. If the control's attribute environment check also returns `null`, then the `ICPI_GETDEFAULTATTR` method is called. Prototype: `char* (*getattr) (Ihandle* self, char* attr);`
ICPI_GETDEFAULTATTR iupCpiDefaultGetDefaultAttr	This method is called by IUP when verifying an attribute value, when both the call to the `ICPI_GETATTR` method and the verification of the control's attribute environment fail (returned `null`). Prototype: `char* (*getdefaultattr) (Ihandle* self, char* attr);`
ICPI_GETSIZE iupCpiDefaultGetSize	This method is called by IUP to verify the size the control must have. This function must write to the `w` and `h` parameters the control size in pixels, respectively. The return value for this function can only be one of the following: `0` - The control size does not vary when the dialog size varies. `1` - The control width may vary when the dialog width varies. `2` - The control height may vary when the dialog height varies. `3` - The control width and height may vary when the dialog size varies. Prototype: `int (*getsize) (Ihandle *self, int *w, int *h);`
ICPI_POPUP (no default)	This method is called by IUP when wishing the control to show a popup dialog. The `x` and `y` parameters indicate the position the dialog must initially have. This method must return `IUP_NOERROR`, if no error occurs, or `IUP_ERROR` if an error occurs. Prototype: `int (*popup) (Ihandle *self, int x, int y);`

System

IUP has several global tables as togheter with some system tools must be initialized before any dialog is created. And the IupLua binding must be initialized also.

The default system language used by predefined dialogs and messages is Portuguese. But it can be changed to English.

System Guide

Initialization

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.

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();
}
```

IupLua Initialization

Before running any function from the Lua Binding, you must run the **`iuplua_open`** function to initialize the toolkit. This function should 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 */
  iupcontrolslua_open(); /* Initialize CPI controls binding Lua */

/* do other things, like running a lua script */
  lua_dofile("myfile.lua");

  IupMainLoop();

  lua_close();

  IupControlsClose();
  IupClose();
  return 0;
}
```

See the examples: iuplua_init.c for Lua 3 and iuplua5_init.c for Lua 5.

It is also allowed to call **`iuplua_open`** without calling **`IupOpen`**. Then **`IupOpen`** will be internally called. This enable you to dynamically load IUP using Lua 5 "require". This is also valid for all the additional controls when IUP is dynamically loaded. To call **`IupClose`** in this way you must call **`iuplua_close`**.

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.

In LED, attributes and expressions follow this form:

```
elem = 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.

The LED files are dynamically loaded and must be sent together with the application's executable. However, this often becomes an inconvenience. To deal with it, there is the LEDC compiler that creates a C module from the LED contents.

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

IupLua

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.

In IupLua5, attributes and expressions follow this form:

```
elem = iup.element{...expression...;
attribute1=value1,attribute2=value2,...}
```

The names of element creation functions are in lower case, since they are actually constructors of Lua tables.

Callbacks can be implemented directly in Lua see Events and Callbacks Guide.

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.

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 in are access through their exact name in the C API.
- Numeric definitions where 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.

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 Lua files are dynamically loaded and must be sent together with the application's executable. However, this often becomes an inconvenience. To deal with it, there is the **LuaC** compiler that creates a C module from the Lua contents. For example:

```
luac -o myfile.lo myfile.lua
bin2c myfile.lo > myfile.loh
```

In C, you can used a define to interchanged the use of .LOH files:

```
#ifdef _DEBUG
  ret_val = lua_dofile("myfile.lua");
#else
#include "myfile.loh"
#endif
```

The distribution files include two executables, one for Lua 3 (**IupLua**) and one for Lua 5 (**IupLua5**), that you can use to test your Lua code. Both applications have support for all the addicional controls and are available at the Download.

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
```

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

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 `Iup`*`Name`*, 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.

IupLua Advanced Guide

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 and to improve the error report, the following 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_name);
```

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

If the user chooses to use `lua_dofile` and `lua_dostring`, 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
```

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 should be called by the host program. If IUP is dynamically loaded in Lua 5 then you should call **`iuplua_close`**.

See Also

`IupOpen`

iuplua_open

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

It is also allowed to call **`iuplua_open`** without calling **`IupOpen`**. Then **`IupOpen`** will be internally called. This enable you to dynamically load IUP using Lua 5 "require". This is also valid for all the additional controls when IUP is dynamically loaded. To call **`IupClose`** in this way you must call **`iuplua_close`**.

Parameters/Return

```
int iuplua_open(void); [in C for Lua 3]
int iuplua_open(lua_State *L); [in C for Lua 5]
[There is no equivalent in Lua]

For Lua 3 returns a non zero value if successfull.
For Lua 5 returns 0 (the number of elements in the stack).
```

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 for Lua 3]
int iupkey_open(lua_State *L); [in C for Lua 5]
iup.key_open() [in Lua 5]
```

See Also

K_ANY callback, KEY attribute

IupVersion

Returns a string with the IUP version number.

Parameters/Return

```
char* IupVersion(void); [in C]
IupVersion() -> (version: string) [in IupLua3]
iup.Version() -> (version: string) [in IupLua5]
```

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.

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;
}
```

Motif System Driver

Driver for the X-Windows/Motif 2.x environment. But it can run in Motif 1.x.

Environment Variables

`QUIET`

When this variable is set to NO, IUP will generate a message in console indicating the driver's version when initializing. Default: YES.

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.

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.

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	`startx --bpp 24`

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 usually means that you are using a Motif with a different version than the Motif used to build IUP. Reinstall Motif or rebuild IUP using your Motif.

- **Some Control Sizes are wrong**

Sometimes the control initialization is incomplete and its size is miscalculated. To solve this call IupMap (dialog) and set the dialog size to NULL "IupSetAttribute(dlg, IUP_SIZE, NULL);" before calling IupShow. This will fix the sizes. We hope to solve this problem in future versions.

Win32 System Driver

This driver was designed for the modern Microsoft Windows in 32 bits (2000/XP/2003).

Environment Variables

`QUIET`

When this variable is set to NO, IUP will generate a message in console indicating the driver’s version when initializing. Default: YES.

`VERSION`

When this variable is set, IUP generates a message dialog indicating the driver's version when initializing.

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

- **`InitCommonCtrl` Link Error**

  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.

- **Custom IupFileDlg**

  To use some cursors and the preview area of **`IupFileDlg`** you must include the "iup.rc" file into your makefile. Or include the contents of it into your resource file, you will need also to copy the cursor files.

- **Windows XP Visual Styles**

  XP Visual Styles can be enabled using a manifest file. Uncomment the manifest file section in "iup.rc" file or copy it to your own resource file, you will need also to copy the manifest file "iup.manifest".

- **Black Canvas**

  The **`IupGLCanvas`** does not work when inside an **`IupFrame`**, the result is a black canvas with no drawing.

Attributes

Attributes are used to change properties of 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 value can be computed or simply verified also it can be internally stored or not.

The attribute is first checked at the element specific implementation at the driver (if mapped) or at the custom control. If not defined then it checks in the hash table. If not defined in its hash table, the attribute will be inherited from its parent and so forth, until it reaches the dialog. But if still then the attribute is not defined a default value for the element is returned (the default value can also be NULL).

Only a few attributes are not inherited: `"TITLE"`, `"VALUE"`, `"ALIGNMENT"`, `"X"`, `"Y"`, `"RASTERSIZE"` and `"SIZE"`.

When an attribute is set it is always stored at the hash table unless the driver disable the storage. If the value is `NULL`, the attribute will be removed from the hash table. Then the driver or the custom control is updated. Finally the attribute is also updated for the children of the element in the driver if they do not have the attribute defined in their own hash table.

Attributes Guide

Using

Attributes are strings, and there are two functions to change them:

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

With `IupSetAttribute` you can also store application pointers that can be strings or not. This can be very useful, for instance, used together with **callbacks**. For example, by storing a C pointer of an application defined structure, the application can retrieve this pointer 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.

When an attribute is set it is always stored at the hash table unless the driver disable the storage. If the value is `NULL`, the attribute will be removed from the hash table. Then the driver or the custom control is updated. Finally the attribute is also updated for the children of the element in the driver if they do not have the attribute defined in their own hash table.

There are attributes common to all the elements. These attributes sometimes are not mentioned in each element documentation. We assume that the programmer knows they exist. In some cases, common attributes behave differently in different elements, but in such cases, there are comments explaining the behavior.

In LED there is no need to add the prefix `IUP_` or quotation marks for attributes, names or values.

Inheritance

Elements included in other elements can inherit their attributes. This means there is an **inheritance** mechanism inside a given dialog.

This means, for example, that if you set the `"MARGIN"` attribute of a `vbox` containing several other elements, including other `vbox`es, all the elements depending on the attribute `"MARGIN"` will be affected, except for those who the `"MARGIN"` attribute is already defined.

Please note that not all attributes are inherited. Exceptions are: `"TITLE"`, `"VALUE"`, `"ALIGNMENT"`, `"X"`, `"Y"`, `"RASTERSIZE"` and `"SIZE"`.

The attribute is first checked at the element specific implementation at the driver (if mapped) or at the custom control. If not defined then it checks in the hash table. If not defined in its hash table, the attribute will be inherited from its parent and so forth, until it reaches the dialog. But if still then the attribute is not defined a default value for the element is returned (the default value can also be NULL).

IupLua

Each interface element is created as a Lua table, and its attributes are 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. However, not all attributes are transferred to IUP.

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, are not transfered. 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.

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

```
myButton = iupbutton{title = "Ok", bgcolor = "0 255 0"}
(IupLua3)
myButton = iup.button{title = "Ok", bgcolor = "0 255 0"}
(IupLua5)
```

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

```
myButton.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.

In the creation of an element 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 paremeters that are necessary for the element to be created in C.

Some interface elements can contain one or more 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:

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

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 method 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.

The attributes in charge of treating the actions associated to objects are not directly transferred to IUP. 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.

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 IupLua3]
iup.SetAttribute(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 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.

A very common mistake when using IupSetAttribute is to use local string arrays to set attributes. For ex:

```
      {
        char value[30];
        sprintf(value, "%d", i);
        IupSetAttribute(dlg, "BADEXAMPLE", value)   //  WRONG  (value pointer will be 
      }                                             //          but its memory will be
                                                    // Use IupStoreAttribute in this ca

      {
        char *value = malloc(30);
        sprintf(value, "%d", i);
        IupSetAttribute(dlg, "EXAMPLE", value)     //  correct  (but to avoid memory le
      }                                                          after the dialog has 

      IupSetAttribute(dlg, "VISIBLE", "YES")        //  correct (static values still ex

      char attrib[30];
      sprintf(attrib, "ITEM%d", i);
      IupSetAttribute(dlg, attrib, "Test")          //  correct (attribute names are a

      struct myData* mydata = malloc(sizeof(struct myData));
      IupSetAttribute(dlg, "MYDATA", (char*)mydata)     //  correct  (unknown attribut
```

When an attribute is set it is always stored at the hash table. If the value is `NULL`, the attribute will be removed from the hash table. Then the driver or the custom control is updated. Finally the attribute is also updated for the children of the element in the driver if they do not have the attribute defined in their own hash table.

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", "Hello!");

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: ihandle, a: string) -> value: string [in IupLua3]
iup.GetAttribute(element: ihandle, 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.

The attribute is first checked at the element specific implementation at the driver (if mapped) or at the custom control. If not defined then it checks in the hash table. If not defined in its hash table, the attribute will be inherited from its parent and so forth, until it reaches the dialog. But if still then the attribute is not defined a default value for the element is returned (the default value can also be NULL).

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 [in IupLua
iup.SetAttributes(element: iulua_tag, attributes: string) -> elem: iulua_tag [in IupLu
```

`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, RASTERSIZI
        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 `v1=a1,v2=a2,...` where `vi` is the name of an attribute and `ai` 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` may cancels IUP's internal buffer. This means that after the call to `IupGetFloat`, the contents previously returned by function `IupGetAttribute` may 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` may invalidates IUP's internal buffer. This means that after a call to this function the contents previously returned by `IupGetAttribute` may 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`

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. It also accepts values on the native format.

In **Windows**, the native format is as 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"
```

Default: "Tahoma" for Windows 2000 and Windows XP, "MS Sans Serif" for others. Size default is 8, or 10 if the resolution is greatter than 100 DPI.

In **Motif**, the native format uses the X-Windows font string format. You can use program **`xfontsel`** to select a font and obtain the string. For example:

```
"-*-times-medium-r*-*-10-*"
"-*-sans serif-*-o-*-*-19-*"
"-*-courier-*-r-*-*-14-*"
```

Default: "-misc-fixed-bold-r-normal-*-13-*" if not defined in a user resource file.

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. When this attribute is not defined it will be calculated to fit the contents of the control.

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 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 the control is mapped, the returned size in IupGetAttribute will still be the current size.

Affects

All.

See Also

`EXPAND`, `RASTERSIZE`

WID

Element identifier in the native interface system.

Value

In Motif, returns a `Widget` which identifies the Intrinsics control.

In Windows, returns a handle (`HWND`) that identifies the window in the native system.

Note

Verification-only attribute, available after the control is mapped.

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

"`widthxheight`", 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.

ZORDER

Change the ZORDER of a dialog or control. It is commonly used for dialogs, but it can be used to control the z-order of controls in a IupCbox.

Value

Can be "TOP" or "BOTTOM".

Affects

All controls that have visual representation.

Motif Common Attributes

XDISPLAY

Returns a `Display*`, indicating the control's X display. It is a verification-only attribute, available after the control is mapped.

XWINDOW

Returns a `Window`, indicating the control's X window. It is a verification-only attribute, available after the control is mapped.

XSCREEN

Returns a `Screen*`, indicating the control's X screen. It is a verification-only attribute, available after the control is mapped.

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 "*`width`*`x`*`height`*" format.

SCREENDEPTH

Returns the screen depth in bits per pixel.

LOCKLOOP

Locks the loop even when an all dialogs have been closed. Possible values: "YES" or "NO".

CURSORPOS

This attribute programaticaly changes the cursor position. Accept values in the format "posh**x**posv", example "200x200", in absolute coordinates relative to the upper left corner of the screen.

COMPUTERNAME

Returns the hostname.

USERNAME

Returns the user logged in.

DLGBGCOLOR

Returns the default dialog background color.

Win32 Global Attributes

HINSTANCE

This attribute returns a handle (**HINSTANCE**) that identifies the application in the native system. It is a verification-only attribute.

SYSTEMLANGUAGE

Return respectively a text with a description of the system language.

WIN_DEFAULTFONT

Stores the name of the default font used in the interface controls.

SHIFTKEY

Returns the state of the Shit keys (left and right). Possible values: "ON" or "OFF".

CONTROLKEY

Returns the state of the Control keys (left and right). Possible values: "ON" or "OFF".

Motif Global Attributes

MOTIFVERSION

Returns the version of the run time Motif.

TRUECOLORCANVAS

Indicates if the display allows creating TrueColor (> 8bpp) windows, even if PseudoColor is the default. Returns `"YES"` or `"NO"`.

AUTOREPEAT

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.

Events and Callbacks

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 application can inform IUP which callback to be called, informing that en event has taken place. So events are handled through callbacks. And callbacks are just functions that the application register in IUP.

The events are processed only when IUP has the control of the application. After the application create and show a dialog it must return the control to IUP so it can process incoming events. This is done in the IUP main event loop. And it is usually done once at the application "main" function. One exception is the display of modal dialogs. This dialog will have its own event loop and the previous shown dialogs will stop receiving events until the modal dialog returns.

Events and Callbacks Guide

Using

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 application 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:

```
int myButton_action(Ihandle* self);
...
IupSetAttribute(myButton, "ACTION", "my_button_action");
IupSetFunction("my_button_action", (Icallback)myButton_action);
```

In LED, callback are only assigned by their names. It will be still necessary to associate the name with the corresponding function in C using `IupSetFunction`. For example:

```
bt = button("Title", my_button_action)  # In LED, is equivalent to the IupSetAtt
```

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 (self).

The callbacks implemented in C by the application must return one of the following 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 it 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 it applies or not.
- `IUP_CONTINUE`: Makes the element to ignore the callback and pass the treatment of the execution to the parent element.

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.

Inside 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.

Main Loop

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 application must call the `IupMainLoop` function, which is generally used right before `IupClose`.

When the application is closed by returning IUP_CLOSE in a callback or by user closing the last dialog, the function `IupMainLoop` will return.

The `IupLoopStep` and the `IupFlush` functions force the processing of incoming events while inside an application callback.

IupLua

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 IupLua3 do not contain the suffix “`_CB`”, in IupLua5 the names are the same.

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. The following example shows the definition of an action for a button.

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

Or you can do

```
function myButton_action(self)
  ...
end

myButton.action = myButton_action
```

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.

In Motif, if the X server sent an event which is not yet in the event queue, after a call to `IupFlush` the queue might not be empty.

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`.

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 IupLua3]
elem:button_cb(but, pressed, x, y: number, status: string) -> (ret: number) [in IupLua
```

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)
isdouble(status)
```

They return 1 if the respective key or button is pressed, and 0 otherwise.

Notes

This callback can be used to customize a button behavior. For a standard button use the ACTION callback of the IupButton.

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`.

DROPFILES_CB

Action called when a file is "dragged" to the application. When several files are dragged, the callback is called several times, once for each file.

Callback

```
int function(Ihandle *self, char* filename, int numFile, int posx, int posy); [in C]
elem:dropfiles(filename: string; numFile, posx, posy: number) -> (ret: number) [in Iup
elem:dropfiles_cb(filename: string; numFile, posx, posy: number) -> (ret: number) [in
```

`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.

The callback must return `IUP_DEFAULT` to be called again for each of the dragged files.

Returning `IUP_IGNORE`, the process is interrupted.

Notes

The callback must be set before the element is mapped.

Windows Only, not available in Motif.

Affects

IupDialog, IupCanvas

ENTERWINDOW_CB

Action generated when the mouse enters the canvas or button.

Callback

```
int function(Ihandle *self); [in C]
elem:enterwindow() -> (ret: number) [in IupLua3]
elem:enterwindow_cb() -> (ret: number) [in IupLua5]
```

`self`: identifier of the control to where the mouse has entered.

Affects

IupCanvas, IupButton

In Motif can also be used by other controls.

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.

In Windows this callback changes the message loop to a more CPU consuming one. Set to NULL when not using.

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 returns the code of any other key, IUP will treat this new key instead of the one typed by the user. If returns `IUP_CONTINUE`, the event will be propagated to the parent of the element receiving it. If returns IUP_DEFAULT is terminated normally.

Notes

All defined keys are also callbacks of any element, called when the respective key is activated. For exemple: "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.`

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 or button.

Callback

```
int function(Ihandle *self); [in C]
elem:leavewindow() -> (ret: number) [in IupLua3]
elem:leavewindow_cb() -> (ret: number) [in IupLua5]
```

`self`: identifier of the control from where the mouse left

Affects

`IupCanvas`

In Motif can also be used by other controls.

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)
isdouble(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)
isdouble(r)
```

Notes

In Motif delta is always 1or -1. In Windows is 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

Dialogs

In IUP you can create your own dialogs or use one of the predefined dialogs. To create your own dialogs you will have to create all the controls of the dialog before the creation of the dialog. All the controls must be composed in a hierarchical structure so the root will be used as a parameter to the dialog creation.

When a control is created, its parent is not known. 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 are inserted, using the dialog as a parent. This feature creates some limitations for IUP, usually related to the insertion and removal of controls.

Since the controls are created in a different order from the native system, native controls can only be created after the dialog. This will happen automatically when the application call the `IupShow` function to show the dialog. But we often need the native controls to be created so we can use some other functionality of those before they are visible to the user. For that purpose, the `IupMap` function was created. It forces IUP to map the controls to their native system controls. The `IupShow` function internally uses `IupMap` before showing the dialog on the screen. IupShow can be called many times, but the map process will occur only once.

`IupShow` can be replaced by `IupPopup`. In this case the result will be a modal dialog and all the other previously shown dialogs will be unavailable to the user. Also the program will interrupt in the function call until the application return `IUP_CLOSE`. When a modal dialog is displayed the application can display other modal dialogs using `IupPopup` again, but not other dialogs using `IupShow`.

After showing the dialog and before IupClose, the application is in charge of destroying all the dialogs.

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.

Creation

```
Ihandle* IupDialog(Ihandle *element); [in C]
iupdialog{element: ihandle} -> (elem: ihandle) [in IupLua3]
iup.dialog{element: ihandle} -> (elem: ihandle) [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

`BORDER`: Shows a thick border around the dialog. Default: `"YES"`. Used only in the creation. Can not be changed after that.

`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. In Motif you may have to click in the dialog to set its focus.

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. Default: YES.

BRINGFRONT (`Windows Only`)

This attribute makes the dialog the foreground window. Use `"YES"` to activate it. Useful for multithreaded applications.

COMPOSITED (Windows Only)

This attribute controls if the created window will have an automatic double buffer for all controls. Default is "NO". Creation-only attribute.

LAYERED (Windows Only)

This attribute sets and removes the layered style bit. Use "YES" to activate it. Default is "NO". The LAYERALPHA attribute must also be set, just after this one.

LAYERALPHA (Windows Only)

This attribute sets the dialog transparency alpha value. The dialog must have LAYERED=YES. Valid values range from 0 (completely transparent) to 255 (opaque).

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 shown. Values: "FULL", "MAXIMIZED", "MINIMIZED" and "NORMAL". After IupShow the attribute is set to "NORMAL" if it was different. "NORMAL" is equivalent of not defining the attribute. FULL is simular to FULLSCREEN but all the dialog client area covers the screen area, menu is not included and decoration is not removed, they just stay out of the screen.

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.

CONTROL (Windows Only): Embeds the dialog inside another window. Creation-only attribute.

MDIMENU (Windows Only): Name of a IupSubmenu to be used as the Window list of a MDI frame window. The system will automatically add the list of MDI child windows there. Used in the IupDialog configured as a MDI frame. This dialog must contains one IupCanvas configured with MDICLIENT=YES.

MDICHILD (Windows Only): Configure this dialog to be a MDI child window. The MDICLIENT attribute must also be set. Each MDI child is automatically named (IupSetHandle) to "mdichild%d", where "%d" is a sequencial number.

MDICLIENT (Windows Only): Name of the IupCanvas used as MDI client window. The MDI frame window must have one and only one MDI client window.

MDIARRANGE (Windows Only): Action to arrange MDI child windows. Possible values: TILEHORIZONTAL, TILEVERTICAL, CASCADE and ICON (arrange the minimized icons).

MDIACTIVATE (Windows Only): Name of a MDI child window to be activated. If value is "NEXT" will activate the next window after the current active window. If value is "PREVIOUS" will activate the previous one.

MDIACTIVE (Windows Only): Returns the the name of the current active MDI child.

MDINEXT (Windows Only): Returns the name of the next available MDI child. Must use MDIACTIVE to retrieve the first child. If the application is going to destroy the child retreive the next child before destroying the current.

MDICLOSEALL (Windows Only): Action to close and destroy all MDI child windows. The CLOSE_CB callback will be called for each child.

> IMPORTANT: When a MDI child window is closed it is automatically destroyed, the application can override this returning IUP_IGNORE in CLOSE_CB.

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 IupLua3]
elem:trayclick_cb(but, pressed, dclick: number) -> (ret: number) [in IupLua5]
```

but: identifies the activated mouse button.
pressed: indicates the state of the button.
dclick: indicates a double click.

Returning CLOSE closes the dialog.

MDIACTIVATE_CB: (Windows Only) Called when a MDI child window is activated. Only the MDI child receive this message. It is not called when the child is shown for the first time.

```
int function(Ihandle *n); [in C]
elem:mdiactivatecb() -> (ret: number) [in IupLua3]
elem:mdiactivate_cb() -> (ret: number) [in IupLua5]
```

Notes

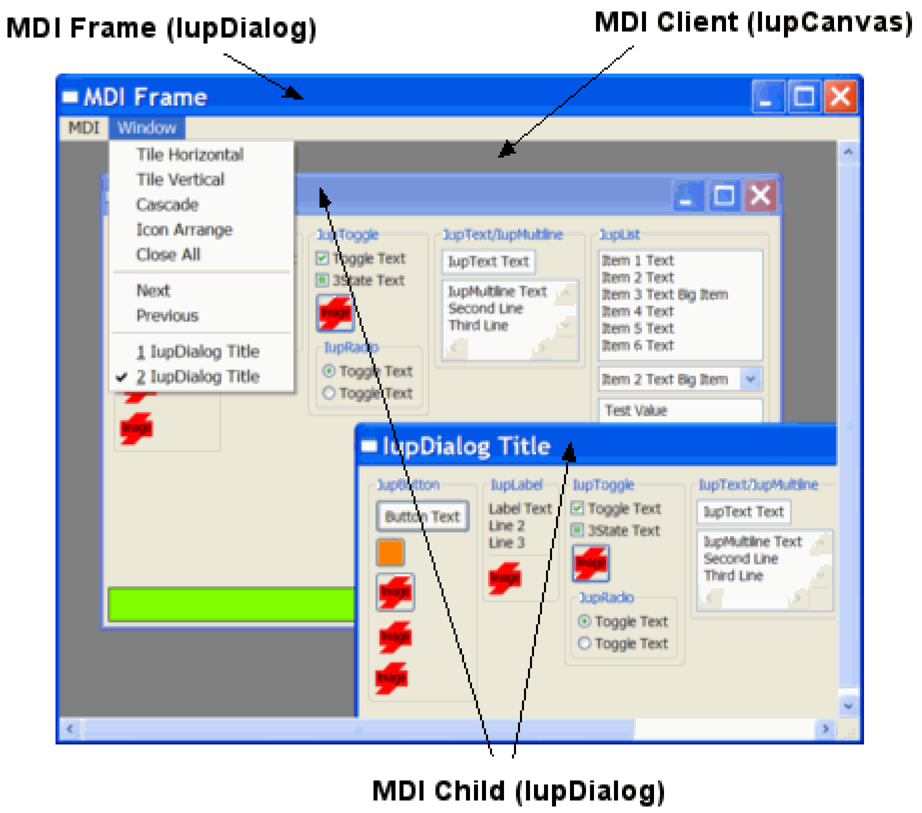
Except for the menu, all other elements must be inside a dialog to interact with the user. Therefore, an interface element will only be visible when its VISIBLE attribute and that of the dialog are "YES".

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 ICON, MENUBOX, MINBOX, MAXBOX, RESIZE and BORDER will work only if the running Window Manager supports the Motif WM hints.

The MDI support is composed of 3 components: the MDI frame window (IupDialog), the MDI client window (IupCanvas) and the MDI children (IupDialog). Altough the MDI client is a IupCanvas it is not used directly by the application, but it must be created and included in the dialog that will be the MDI frame, other controls can also be available in the same dialog, like buttons and other canvases composing toolbars and status area. The following picture illustrates the e components:



[Examples](#)

IupDestroy

Destroys an interface element and all of its descendants. Only dialogs, timers, popup menus and images should be normally destroyed, but detached controls can also be destroyed.

Parameters/Return

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

element: Identifier of the interface element to be destroyed.

Notes

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

Menu bars associated with dialogs are automatically destroyed. Images associated with controls are not destroyed, because images can be reused in several controls the application must destroy them when they are not used anymore.

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: ihandle) -> (ret: number) [in IupLua3]
iup.Hide(element: ihandle) -> (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, or the element mapped, before a dialog is made visible.

See Also

`IupShowXY`, `IupShow`, `IupPopup`.

IupRefresh

Updates the size and layout of controls after changing size attributes. Can be used for any element inside a dialog or for the dialog itself. It can change the layout of all the controls inside the dialog because of the dynamic layout positioning.

Parameters/Return

```
int IupRefresh(Ihandle *element); [in C]
IupRefresh(element: ihandle) -> (ret: number) [in IupLua3]
iup.Refresh(element: ihandle) -> (ret: number) [in IupLua5]
```

`element`: identifier of the interface element.

Notes

This function will not change the size of the dialog. Changing the size of the controls may position some controls outside of the dialog on the left or bottom borders.

If you want to also change the size of the dialog use:

```
IupSetAttribute(dialog, "SIZE", NULL);
IupShow(dialog);
```

so the dialog will be redisplayed with the new size.

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 exemple using **`IupHide`**.

Parameters/Return

```
int IupPopup(Ihandle *element, int x, int y); [in C]
IupPopup(element: ihandle, x, y: number) -> (ret: number) [in IupLua3]
iup.Popup(element: ihandle, 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 a dialog in the current position. If the dialog needs to be mapped and the current position is not known the dialog is centered. This function has the same effect as setting value `IUP_YES` to the `IUP_VISIBLE` attribute of the dialog.

Parameters/Return

```
int IupShow(Ihandle *element); [in C]
IupShow(element: ihandle) -> (ret: number) [in IupLua3]
iup.Show(element: ihandle) -> (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: ihandle, x, y: number) -> (ret: number) [in IupLua3]
iup.ShowXY(element: ihandle, 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.

IupFileDlg

Creates the File Dialog element. It is a predefined dialog for selecting files or a directory. The dialog can be shown with the `IupPopup` function only.

Creation

```
Ihandle* IupFileDlg (void); [in C]
iupfiledlg() -> (elem: ihandle) [in IupLua3]
iup.filedlg() -> (elem: ihandle) [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. If the user clicks OK, this attribute will contain the filename selected by the user.

`FILTER`: String containing a list of file filters valid in the native system, separated by `';'` without spaces. Example: `"*.C;*.LED;teste.*"`.

`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. If not defined the dialog opens in the current directory. In Windows, if the current directory does not have files corresponding to the chosen filter, the directory opened will be "My Documents".

`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. Default: if the dialog is of type `"OPEN"`, default is `"NO"`; if the dialog is of type `"SAVE"`, default is `"YES"`.

`NOCHANGEDIR`: Indicates if the initial working directory must be restored after the user navigation. Default: `"YES"`.

`FILEEXIST`: Indicates if the file defined by the `FILE` attribute exists or not. It is only valid if the user has pressed OK in the dialog.

`STATUS`: Indicates the status of the selection made:

"**`1`**": New file.
"**`0`**": Normal, existing file.
"**`-1`**": Operation cancelled.

`VALUE`: Name of the selected file(s), or `NULL` if no file was selected. In Windows there is a limit of 32Kb for this string.

`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 retreive 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); [in C]
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 currect 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 the a
"PAINT" - the preview area must be repainted. This is used only when the pr
```

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 modal dialog containing a message and up to three buttons.

Creation and Show

```
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

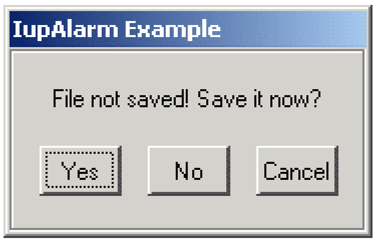
This function shows a dialog centralized on the screen, with the message and the buttons. The '`\n`' character can be added to the message to indicate line change.

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. It also uses a global attribute called "ICON" as the dialog icon if it is defined.

Examples



File not saved! Save it now?

See Also

`IupMessage`, `IupScanf`, `IupListDialog`, `IupGetFile`.

IupGetFile

Shows a modal dialog of the native interface system to select a filename. Uses the IupFileDlg element.

Creation and Show

```
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. It also uses a global attribute called "ICON" as the dialog icon if it is defined.

Examples

See Also

`IupMessage`, `IupScanf`, `IupListDialog`, `IupAlarm`, `IupSetLanguage`.

IupGetText

Shows a modal dialog to edit a multiline text.

Creation and Show

```
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 occured returns nil.

Notes

The dialog uses a global attribute called "PARENTDIALOG" as the parent dialog if it is defined. It also uses a global attribute called "ICON" as the dialog icon if it is defined.

See Also

`IupMessage`, `IupScanf`, `IupListDialog`, `IupAlarm`, `IupSetLanguage`.

IupListDialog

Shows a modal dialog to select options from a simple or multiple list.

Creation and Show

```
int IupListDialog(int type, char *title, int size, char *list[], int option, int max_c
IupListDialog(type: number, title: string, size: number, list: table of strings, optio
iup.ListDialog(type: number, title: string, size: number, list: table of strings, opti
```

`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 operationthe 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. It also uses a global attribute called "ICON" as the dialog icon if it is defined.

Examples

See Also

IupMessage, IupScanf, IupGetFile, IupAlarm

IupMessage

Shows a modal dialog containing a message.

Creation and Show

```
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. It also uses a global attribute called "ICON" as the dialog icon if it is defined (used only in Motif, in Windows MessageBox does not have an icon in the title bar).

Examples

See Also

IupGetFile, IupScanf, IupListDialog, IupAlarm

IupScanf

Shows a modal dialog for capturing values with a format similar to the `scanf` function in the C stdio library.

Creation and Show

```
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. It also uses a global attribute called "ICON" as the dialog icon if it is defined.

Examples

Captures an integer number, a floating-point value and a character string.

See Also

IupGetFile, IupMessage, IupListDialog, IupAlarm, IupGetParam

IupGetColor

Shows a modal dialog which allows the user to select a color.

This dialog is included in the Controls Library. It requires an addicional initialization, see the Controls Library documentation.

Creation and Show

```
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.

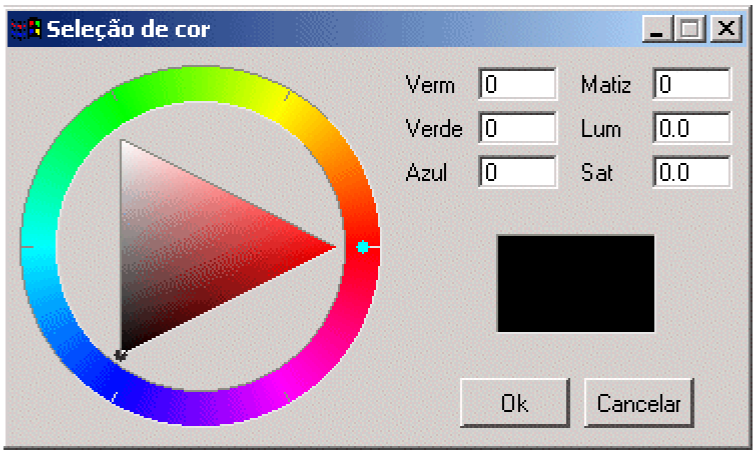
The function returns 1 if the OK button is pressed, or 0 otherwise.

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. It also uses a global attribute called "ICON" as the dialog icon if it is defined.

Examples



See Also

`IupMessage`, `IupScanf`, `IupListDialog`, `IupAlarm`, `IupGetFile`.

IupGetParam

Shows a modal dialog for capturing parameter values using several types of controls.

This dialog is included in the Controls Library. It requires an addicional initialization, see the Controls Library documentation.

Creation and Show

```
int IupGetParam(const char* title, Iparamcb action, void* user_data, const char* forma
IupGetParam(title: string, action: function, format: string,...) -> (ret: number, ...)
iup.GetParam(title: string, action: function, format: string,...) -> (ret: number, ...
```

`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 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);
action(dialog: ihandle, param_index: number) -> (ret: number) [in IupLua]
```

`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 couting 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 Valuators).
"`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.

In Lua to retreive a parameter you must use the following function:

```
IupGetParamParam(dialog: ihandle, param_index: number)-> (param: ihandle) [in IupLua3]
iup.GetParamParam(dialog: ihandle, param_index: number)-> (param: ihandle) [in IupLua5
```

`dialog`: Identifier of the `dialog`.
`para_index`: parameter to be retrieved.

Notes

The format string must have the following format, notice the "\n" at the end "`text%x[extra]\n`", where:

> `text` is a descriptive text, to be placed to the left of the entry field in a label.
>
> `x` 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 the given type
>
> > `[min,max]` are optional limits for integer and real types. The maximum value can be omited. 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 '`]`'.
> > `mask` is an optional mask for the string and multiline types. The dialog uses the IupMask internally. In this case we do no 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.

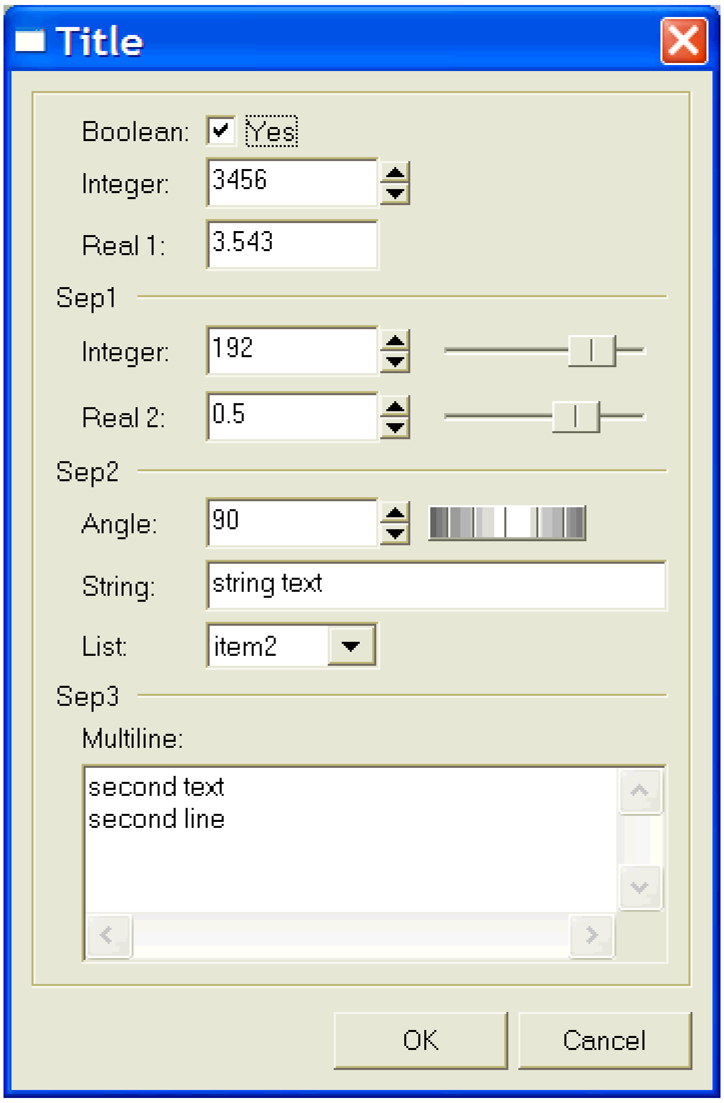
The number of lines in the format string ("\n"s) will determine the number of required parameters. But separators will not count as parameters.

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. It also uses a global attribute called "ICON" as the dialog icon 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`.

Layout Composition

Abstract Layout

Most interface toolkits employ the concrete layout model, that is, control 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 controls 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 controls 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 containers expand or retract to adjust the positioning of the controls 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.

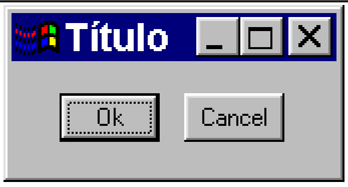
The composition elements are vertical boxes (**`vbox`**), horizontal boxes (**`hbox`**) and filling (**`fill`**). There is also a depth box (**`zbox`**) in which layers of elements can be created for the same dialog, and the elements in each layer are only visible when that given layer is active.

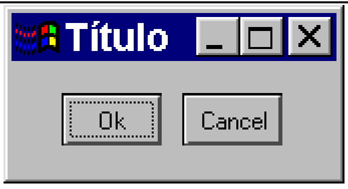
in Windows

```
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
```

in IupLua

```
dialogo = iupdialog
{
  iuphbox
  {
    iupfill{},
    iupbutton{title="Ok",size="40"},
    iupbutton{title="Cancel",size="40"},
    iupfill{}
   ;margin="15x15", gap="10"
  }
 ;title="Título"
}
```





Now see the same dialog in LED and in C:

in LED

```
DIALOG[TITLE="Título"]
(
  HBOX[MARGIN="15x15", GAP="10"]
  (
    FILL(),
    BUTTON[SIZE="40"]("Ok",do_nothing),
    BUTTON[SIZE="40"]("Cancel",do_nothing),
    FILL()
  )
)
```

in C

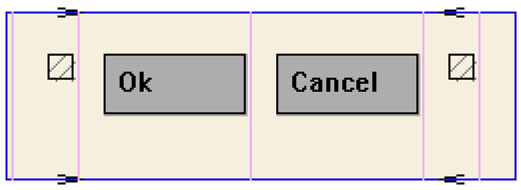
```
dialog = IupSetAttributes(IupDialog
(
  IupSetAttributes(IupHbox
  (
    IupFill(),
    IupSetAttributes(IupButton("Ok", "do_n
    IupSetAttributes(IupButton("Cancel", "
    IupFill(),
    NULL
  ), "MARGIN=15x15, GAP=10"),
), "TITLE = Título")
```

Following, the abstract layout representation of this dialog:

Layout Hierarchy

```
Dialog
  HBox
    Fill
    Button
    Button
    Fill
```

Layout Visualization



Layout Hierarchy

The layout of the elements of a dialog in IUP has a natural hierarchy because of the way they are composed together. The dialog is the root of the hierarchy tree. To retreive the dialog of a control you can simply call `IupGetDialog`, but there are other ways to navigate in the hierarchy tree.

To get all the children of a container use `IupGetNextChild`. To get just the next control with the same parent use IupGetBrother.

The hierarchy tree can also be dynamically created, but before mapping to the native system. You can add and remove elements from a container using IupAppend and IupDetach, but only before mapping into the native system.

IupFill

Creates a `Fill` composition element, which dynamically occupies empty spaces.

Creation

```
Ihandle* IupFill(void); [in C]
iupfill{} -> elem: ihandle [in IupLua3]
iup.fill{} -> elem: ihandle [in IupLua5]
fill() [in LED]
```

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

Attributes

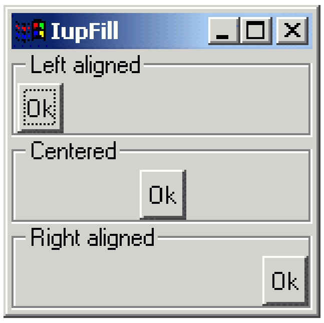
SIZE: Defines the width, if the `Fill` is inside a horizontal box, or the height, if it is inside a vertical box. Default: "`0`".

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` container for composing elements. It is a box that arranges the elements it contains, horizontally and from left to right.

Creation

```
Ihandle* IupHbox(Ihandle *elem1, ...); [in C]
Ihandle* IupHboxv(Ihandle **elems); [in C]
iuphbox{elem1, elem2, ...: ihandle} -> (elem: ihandle) [in IupLua3]
iup.hbox{elem1, elem2, ...: ihandle} -> (elem: ihandle) [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 in C.

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. Default: "0".

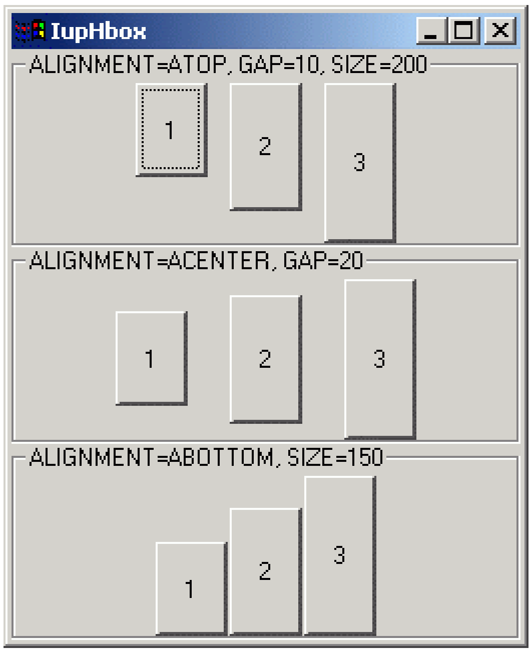
`MARGIN`: Defines a margin in pixels. Its value has the format "*`widthxheight`*", where *`width`* and *`height`* are integer values corresponding to the horizontal and vertical margins, respectively. Default: `"0x0"` (no margin).

`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


IupHbox
ALIGNMENT=ATOP, GAP=10, SIZE=200
1
2
3
ALIGNMENT=ACENTER, GAP=20
1
2
3
ALIGNMENT=ABOTTOM, SIZE=150
1
2
3

See Also

`IupZbox`, `IupVBox`

IupVbox

Creates a `vbox` container for composing elements. It is a box that arranges the elements it contains, vertically and from the top down.

Creation

```
Ihandle* IupVbox(Ihandle *elem1, ...); [in C]
Ihandle* IupVboxv(Ihandle **elems); [in C]
iupvbox{elem1, elem2, ...: ihandle} -> (elem: ihandle) [in IupLua3]
iup.vbox{elem1, elem2, ...: ihandle} -> (elem: ihandle) [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 in C.

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. Default: "0".

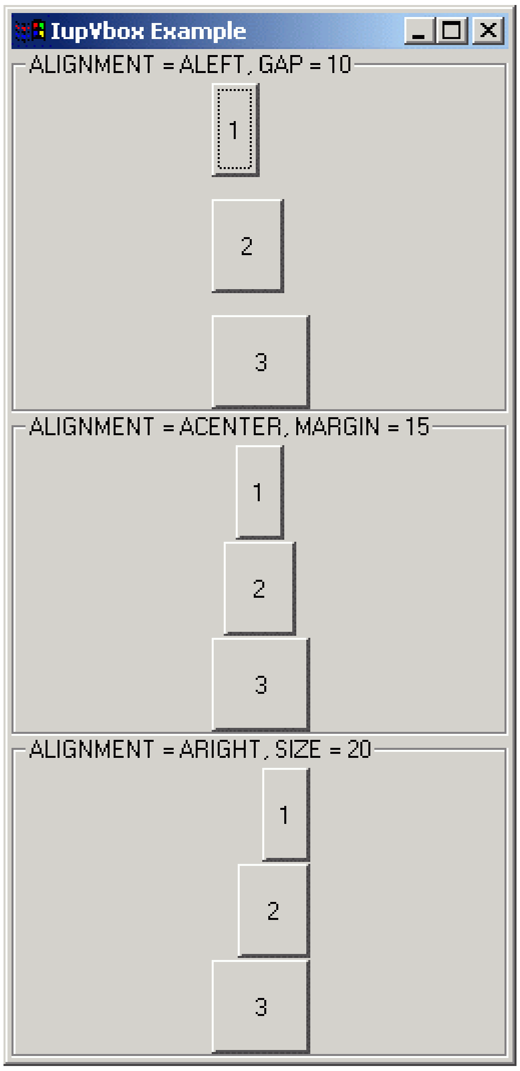
`MARGIN`: Defines a margin in pixels. Its value has the format "*`width`*`x`*`height`*", where *`width`* and *`height`* are integer values corresponding to the horizontal and vertical margins, respectively. Default: `"0x0"` (no margin).

`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


IupVbox Example
ALIGNMENT = ALEFT, GAP = 10
1
2
3
ALIGNMENT = ACENTER, MARGIN = 15
1
2
3
ALIGNMENT = ARIGHT, SIZE = 20
1
2
3

See Also

[IupZbox](), [IupHbox]()

IupZbox

Creates a `zbox` container for composing elements. It is a box that piles up the elements it contains, only the active element is visible.

Creation

```
Ihandle* IupZbox (Ihandle *elem1, ...); [in C]
Ihandle* IupZboxv (Ihandle **elems); [in C]
iupzbox{elem1, elem2, ... : ihandle} -> (elem: ihandle) [in IupLua3]
iup.zbox{elem1, elem2, ... : ihandle} -> (elem: ihandle) [in IupLua5]
zbox(elem1, elem2,...) [in LED]
```

`elem1, elem2, ...`: List of the elements that will be placed in the box. `NULL` defines the end of the list in C.

Important: in C, each element must have a name defined by [`IupSetHandle`](). In Lua a name is always automatically created, but you can change it later.

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 active element. Possible values:

```
"NORTH", "SOUTH", "WEST", "EAST",
"NE", "SE", "NW", "SW",
"ACENTER".
```

Default: "NE".

MARGIN: Defines the margin of the visible element. Its value has the format "*width*x*height*", where *width* and *height* are integer values corresponding to the horizontal and vertical margins, respectively. Default: "0x0" (no margin).

VALUE: Changes the active element. The value passed must be the name of one of the elements contained in the zbox. Default: the first element. To set the name of an element, use the IupSetHandle function. In Lua you can also use the element reference directly.

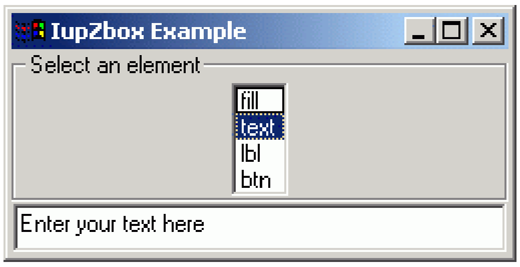
SIZE: Defines the zbox size. Default: the smallest size that fits its largest element.

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

IupRadio

Creates the radio element for grouping mutual exclusive toggles. Only one of its descendet toggles will be active at a time. The toggles can be at any composion.

Creation

```
Ihandle* IupRadio(Ihandle *element); [in C]
iupradio{element: ihandle} -> (elem: ihandle) [in IupLua3]
iup.radio{element: ihandle} -> (elem: ihandle) [in IupLua5]
radio(element) [in LED]
```

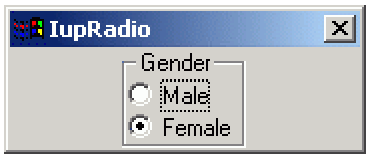
element: Identifier of an interface element. Usually it is a vbox or an hbox containing the toggles associated to the `radio`.

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

Attributes

`VALUE`: Identifier of the active toggle. The identifier is set by means of `IupSetHandle`.

Examples



IupAppend

Inserts an interface element at the end of the container. Valid for `hbox`, `vbox`, `zbox` or `menu`.

Parameters/Return

```
Ihandle* IupAppend(Ihandle *box, Ihandle *element); [in C]
IupAppend(box, element: ihandle) -> (box: ihandle) [in IupLua3]
iup.Append(box, element: ihandle) -> (box: ihandle) [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 can be used when the interface elements that will compose an `hbox`, `vbox`, `zbox` or `menu` are not known *a priori* and should be dynamically constructed.

ATTENTION: Currently, for menus this function only works before the element is mapped.

For boxes, if the dialog is already mapped you must explicitly call **`IupMap`** after inserting a new element.

See Also

`IupDetach`, `IupHbox`, `IupVbox`, `IupZbox`, `IupMenu`.

IupDetach

Disassociates an interface element from its parent.

Parameters/Return

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

`element`: Identifier of the interface element to be detached.

Notes

This function does not destroy the interface element, just remove it from the child list of its parent. If you remove a control from an already mapped dialog you must explicitly call IupDestroy for that control.

See Also

`IupAppend`, `IupDestroy`.

IupGetNextChild

Returns the children of the given control.

Parameters/Return

```
Ihandle *IupGetNextChild(Ihandle *parent, Ihandle *lastchild); [in C]
IupGetNextChild(parent, lastchild: ihandle) -> ret: ihandle [in IupLua3]
iup.GetNextChild(parent, lastchild: ihandle) -> ret: ihandle [in IupLua5]
```

`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. To get the first child use `lastchild=NULL`.

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: ihandle) -> ret: ihandle [in IupLua3]
iup.GetBrother(control: ihandle) -> ret: ihandle [in IupLua5]
```

control: Brother of interface control given.

See Also

IupGetNextChild

IupGetDialog

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

Parameters/Return

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

elem: Identifier of an interface element.

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

Controls

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

But this is valid only for the standard controls, many additional controls are drawn by IUP. Composition controls are not visible, so they are independent from the native system.

Each control has an unique creation function, and all of its management is done by means of **attributes** and **callbacks**, using functions common to all the controls. This simple but powerfull approach is one of the advantages of using IUP.

Controls are automatically destroyed when the dialog is destroyed.

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.

Creation

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

`TITLE`: Text of the button.

FLAT: (Windows Only) Hides the button borders until the mouse enter the button area.

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.

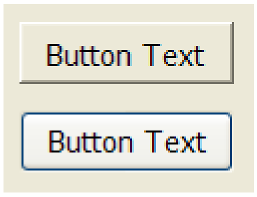
Text and images are always centered.

Buttons are activated using Enter or Space keys.

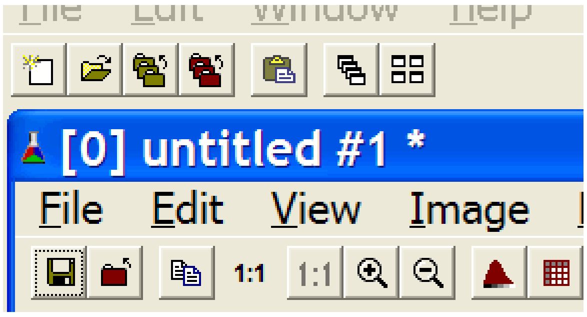
When `IMPRESS` and `IMAGE` are defined together, IUP does not show the element's border to provide a 3D effect; the user has to define the border in the image itself.

In Windows, when using Windows XP Visual Styles the `BGCOLOR` attribute is ignored when a non empty text button is created.

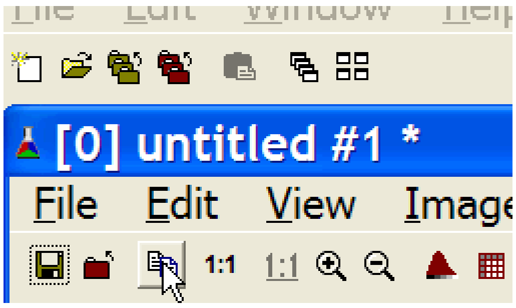
Here are some examples of buttons using Windows XP Visual Styles:



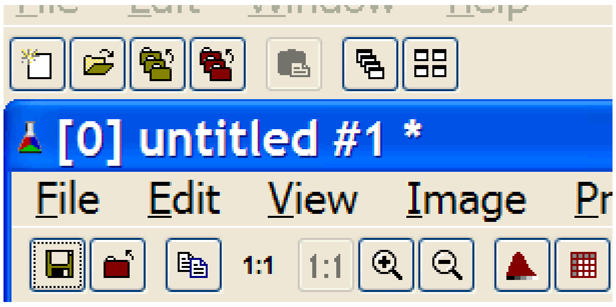
Text button without and with Visual Styles.



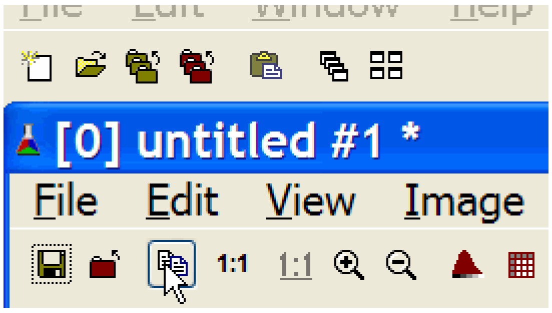
Windows XP without Visual Styles.



Windows XP without Visual Styles and FLAT=YES.
The border is displayed only when the mouse is over the button.
The image of the button should be smaller when FLAT=YES.



Windows XP with Visual Styles.



Windows XP with Visual Styles and FLAT=YES.

Callbacks

`ACTION`: Action generated when the button 1 (usually left) is selected. This callback is called only after the mouse is released and whe it is released inside the button area.

`BUTTON_CB`: Action generated when any mouse button is pressed and released.

ENTERWINDOW_CB: Action generated when the mouse enters the button.

LEAVEWINDOW_CB: Action generated when the mouse leaves the button.

Examples

See Also

IupImage, IupToggle.

IupCanvas

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

Creation

```
Ihandle* IupCanvas(char *action); [in C]
iupcanvas{} -> (elem: ihandle) [in IupLua3]
iup.canvas{} -> (elem: ihandle) [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 background is painted only if the ACTION callback is not defined. If the callback is defined the application must draw all the canvas contents. For better results also let 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: (Windows Only) Shows a border around the canvas. It can only be changed before the element is mapped. Default: "YES".

EXPAND: The default value is "YES".

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 contains a border.

BACKINGSTORE: (Motif Only). Controls the canvas backing store flag. The default value is "YES".

MDICLIENT: (Windows Only). Configure this canvas as an MDI client window. No callbacks will be called. This canvas will be used internally only. The default value is "NO".

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.

The mouse cursor position can be programatically controled using the global attribute CURSORPOS.

Examples

IupFrame

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

Creation

```
Ihandle* IupFrame(Ihandle *element); [in C]
iupframe{element: ihandle} -> (elem: ihandle) [in IupLua3]
iup.frame{element: ihandle} -> (elem: ihandle) [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. If not defined during creation it can not be added lately, to be changed it must be at least "" during creation.

`MARGIN`: Margin of the visible element. Its value has the format "*`widthxheight`*", where *`width`* and *`height`* are integer values corresponding to the horizontal and vertical margins, respectively. Default: `"0x0"` (no margin).

`SUNKEN`: (Windows Only) When not using a title, the frame line defines a sunken area (lowered area). Valid values: `YES` or `NO`*.* Default: `NO`.

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

IupLabel

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

Creation

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

`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.

Creation

```
Ihandle* IupList(char *action); [in C]
```

```
iuplist{} -> (elem: ihandle) [in IupLua3]
iup.list{} -> (elem: ihandle) [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.

`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. Default: Depends on the native system.

`MULTIPLE`: Allows selecting several items simultaneously (multiple list). Default: `"NO"`. Creation only attribute in Windows. Valid only for simple lists with no edit box.

`SIZE`: Size of the list. Default: room for 5 characters in 1 item.

`VALUE`: Depends on the list type:

> **Simple 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: Action to open or close the dropdown list. Can be "YES" or "NO". Valid only when DROPDOWN=YES.

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: number) -> (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.

MULTISELECT_CB: Action generated when the state of an item in the multiple selection list is changed. But it is called only when the interaction is over.

```
int function (Ihandle *self, char *value); [in C]
elem:multiselect(value: string) -> (ret: number) [in IupLua3]
elem:multiselect_cb(value: string) -> (ret: number) [in IupLua5]
```

value: Similar to the VALUE attribute for a multiple selection list, but non changed items are marked with an 'x'.

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:edit(c: number, after: string) -> (ret: number) [in IupLua3]
elem:edit_cb(c: number, after: string) -> (ret: number) [in IupLua5]
```

text: Represents the new text value. This is the same callback definition as for the IupText.

CARET_CB: Action generated when the caret/cursor position is changed. Valid only when EDITBOX=YES.

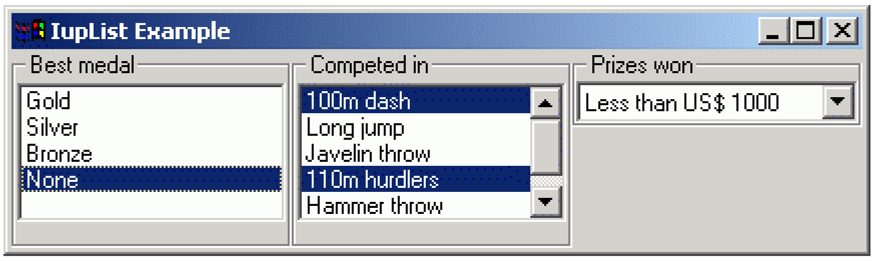
```
int function(Ihandle *self, int row, int col); [in C]
elem:caretcb(row, col: number) -> (ret: number) [in IupLua3]
elem:caret_cb(row, col: number) -> (ret: number) [in IupLua5]
```

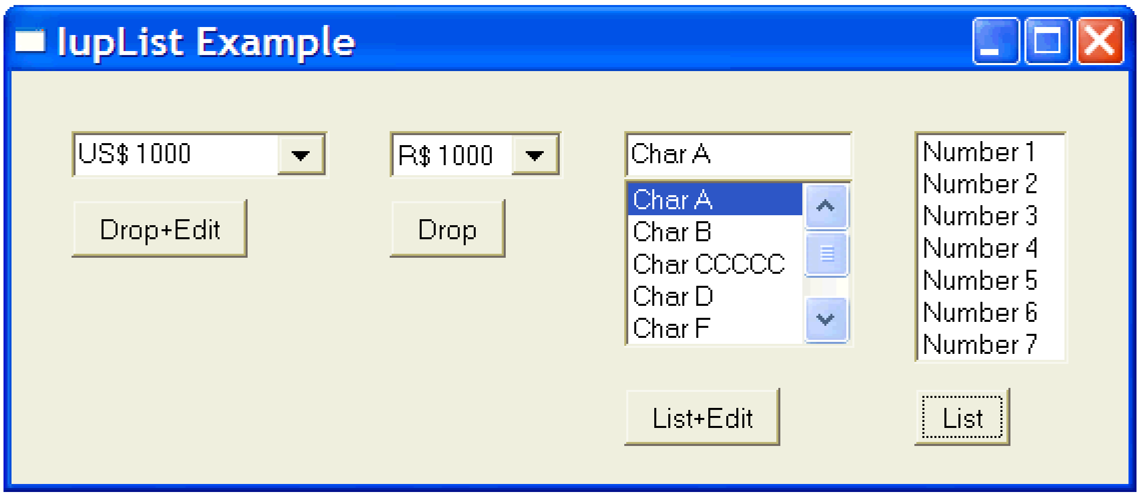
row, col: Row and collumn number.

Notes

Text is always left aligned.

Examples





See Also

IupListDialog, Iuptext

IupMultiLine

Creates an editable field with one or more lines.

Creation

```
Ihandle* IupMultiLine(char *action); [in C]
iupmultiline{} -> (elem: ihandle) [in IupLua3]
iup.multiline{} -> (elem: ihandle) [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. Default: `"YES"`.

`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: room for 5 characters in 1 line.

ALIGNMENT: (Windows Only) Label's alignment. Possible values: "ALEFT", "ARIGHT", "ACENTER". Default: "ALEFT".

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.

CARET_CB: Action generated when the caret/cursor position is changed.

```
int function(Ihandle *self, int row, int col); [in C]
elem:caretcb(row, col: number) -> (ret: number) [in IupLua3]
elem:caret_cb(row, col: number) -> (ret: number) [in IupLua5]
```

row, col: Row and collumn number.

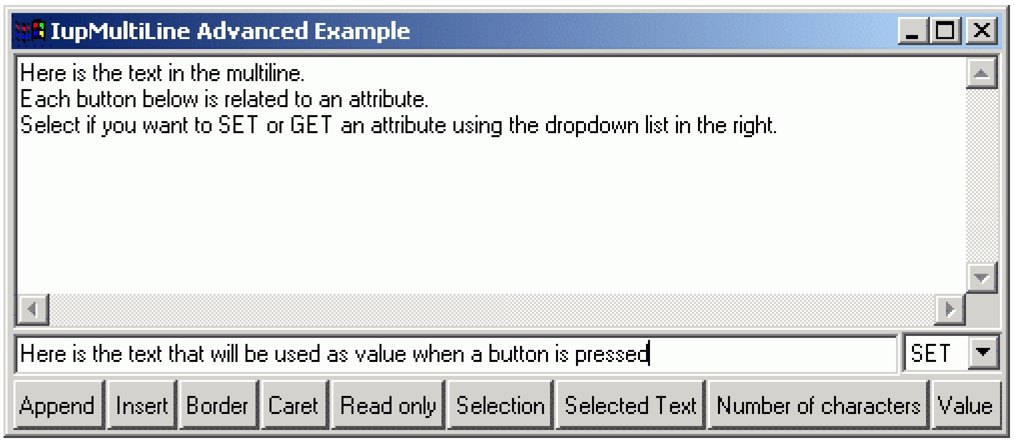
Notes

Text is always left aligned.

The multiline 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



Here is the text in the multiline.
Each button below is related to an attribute.
Select if you want to SET or GET an attribute using the dropdown list in the right.

Here is the text that will be used as value when a button is pressed

IupText

Creates an editable field with one line.

Creation

```
Ihandle* IupText(char *action); [in C]
iuptext{} -> (elem: ihandle) [in IupLua3]
iup.text{} -> (elem: ihandle) [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. Default: `"YES"`.

`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: room for 5 characters.

`ALIGNMENT`: (Windows Only) Label's alignment. Possible values: `"ALEFT"`, `"ARIGHT"`, `"ACENTER"`. Default: `"ALEFT"`.

`PASSWORD`: (Windows Only) Hide the typed character using an "*".

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.

CARET_CB: Action generated when the caret/cursor position is changed.

```
int function(Ihandle *self, int row, int col); [in C]
elem:caretcb(row, col: number) -> (ret: number) [in IupLua3]
elem:caret_cb(row, col: number) -> (ret: number) [in IupLua5]
```

row, col: Row and collumn number.

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

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.

Creation

```
Ihandle* IupToggle(char *title, char *action); [in C]
iuptoggle{title = title: string} -> (elem: ihandle) [in IupLua3]
```

```
iup.toggle{title = title: string} -> (elem: ihandle) [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`". If 3STATE=YES then can also be "`NOTDEF`".

`TITLE`: Toggle's text.

3STATE: Enable a three state toggle. Valid for toggles with text only. Can be "`YES`" or `NO`". Default: "`NO`".

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 funcion(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

Toggle with image or text 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.

To build a set of mutual exclusive toggles, insert them in a `IupRadio` container. They must be inserted before creation, and their behavior can not be changed.

Unlike buttons, toggles always display the button border when IMAGE and IMPRESS are both defined.

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

In Windows XP if IMAGE is used and Visual Styles are enabled the focus feedback is not drawn.

Examples

See Also

`IupImage`, `IupButton`, `IupLabel`, `IupRadio`.

Additional Controls

Controls Library

Most of the addicional controls are included in only one library. Some of these controls are drawn by IUP and are not native controls.

The **`iupcontrols.h`** file must be included in the source code. If you plan to use the control in Lua, you should also include **`iupluacontrols.h`**.

The **`IupControlsOpen`** function must be called after **`IupOpen`**. To make the controls available in Lua, use the initialization function in C, **`iupcontrolslua_open`**, after calling **`iuplua_open`**.

Your application must be linked to the CPI control library (`iupcontrols.lib` on Windows and `libiupcontrols.a` on Unix), and with the CD library. To use its bindings to Lua, the program must also be linked to the luacontrol’s libraries (`iupluacontrols[5].lib` on Windows and `libiupluacontrols[5].a` on Unix).

When closing the application, the user must call the function **`IupControlsClose()`** to free the resources used.

OpenGL Canvas

The drawing canvas compatible with OpenGL is called IupGLCanvas.

Speech Control

Creates a speech engine that allows speech recognition and speech. Uses Microsoft Speech SDK 5.1. See **`IupSpeech`**.

IupCbox

Creates a `Cbox` element. It is concrete layout container, i.e. its children are positioned in specified coordinates. The IupCbox inherits from the IupCanvas, so all the canvas attributes and callbacks are valid. The box must have a specified size. The IupCbox contains a IupHbox where all the children are inserted, but their positioning ignores the IupHbox.

Creation

```
Ihandle* IupCbox(Ihandle* elem1, Ihandle* elem2, ...); [in C]
Ihandle* IupCboxv(Ihandle** elems); [in C]
iupcbox{elem1, elem2, ...: ihandle} -> (elem: ihandle) [in IupLua3]
iup.cbox{elem1, elem2, ...: ihandle} -> (elem: ihandle) [in IupLua5]
cbox(elem1, elem2, ...) [in LED]
```

elem1, elem2, ...: List of the elements that will be placed into `Tabs`.

This function returns the created `Cbox`'s identifier, or `NULL` if an error occurs. The second form in C must end the array with a NULL. The order of the controls in the creation function is irrelevant.

Attributes

CX, CY: (children only) Position in pixels relative to the top-left corner of the box. Must be set for each child inside the box.

Examples

See Also

`IupCanvas`

IupCells

Creates a grid widget (set of cells) that enables several application-specific drawing, such as: chess tables, tiles editors, degradeé scales, drawable spreadsheets and so forth.

This element is mostly based on application callbacks functions that determine the number of cells (rows and coluns), their appearence and interation. This mechanism offers full flexibility to applications, but requires programmers attention to avoid infinite loops inside this functions. Using callbacks, cells can be also grouped to form major or hierarchical elements, such as headers, footers etc.

Since the size of each cell is given by the application the size of the control also must be given using SIZE or RASTERSIZE attributes.

This callback approach was intentionally chosen to allow all cells to be dinamically and directly changed based on application's data structures.

This control implementation is directly inherited on IupCanvas, and is originally implemented by André Clinio.

Creation

```
Ihandle* IupCells(void); [in C]
iupcells{} -> (elem: ihandle) [in IupLua3]
iup.cells{} -> (elem: ihandle) [in IupLua5]
cells() [in LED]
```

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

Attributes

BOXED: Determines if the bounding cells' regions should the drawn with black lines. It can be `"YES"` or `"NO"`. Default: `"YES"`. If the span atributtes are set set this attribute to `"NO"` to avoid grid drawing over spanned cells.

CLIPPED: Determines if, before cells drawing, each bounding region should be clipped. This attribute should the changed in few specific cases. It can be `"YES"` or `"NO"`. Default: `"YES".`

NON_SCROLLABLE_LINES: Determines the number of non-scrollable lines (vertical headers) that should allways be visible despite the vertical scrollbar position. It can be any non-negative integer value. Default: "0"

NON_SCROLLABLE_COLS: Determines the number of non-scrollable columns (horizontal headers) that should allways be visible despite the horizontal scrollbar position. It can be any non-negative integer value. Default: "0"

ORIGIN: Sets the first visible line and column positions. This attribute is set by a formatted string `"%d:%d"` (C syntax), where each "%d"represent the line and column integer indexes respectely.

REPAINT: When set with any value (write-only), provokes the control full repaint.

FULL_VISIBLE: Tries to show completely a specific cell (considering any vertical or horizontal header or scrollbar position) .This attribute is set by a formatted string `"%d:%d"` (C syntax), where each "%d"represent the line and column integer indexes respectely.

NO_COLOR: Adjusts the default color of cells which the drawing callback does nothing. Default: the BGCOLOR attribute.

LIMITS: (Read Only) Returns the limits of a given cell. Input format is `"lin:col"` or `"%d:%d"` in C. Output format is `"xmin:xmax:ymin:ymax"` or `"%d:%d:%d:%d"` in C.

FIRST_COL: (Read Only) Returns the number of the first visible column.

FIRST_LINE: (Read Only) Returns the number of the first visible line.

BUFFERIZE: When set to "YES", disables the control redrawing. It should be used only to avoid the control blinking effect when several attributes are being changed at sequentially. When REPAINT attribute is set, BUFFERIZE is automatically adjusted to "NO". Default: `"NO"`.

IMAGE_CANVAS: Returns the internal image CD canvas (read-only). This attribute should be used only in specific cases and by experienced CD programmers.

CANVAS: Returns the internal IUP CD canvas (read-only). This attribute should be used only in specific cases and by experienced CD programmers.

Callbacks

MOUSECLICK_CB: called when a color is selected. The primary color is selected with the left mouse button, and if existant the secondary is selected with the right mouse button.

```
int function(Ihandle* self, int button, int pressed, int line, int column,
elem:mouseclickcb(button, pressed, line, column, x, y: number, string: stat
elem:mouseclick_cb(button, pressed, line, column, x, y: number, string: sta
```

self: identifies the control that activated the function's execution.
but: identifies the activated mouse button (just like in canvas control):

`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`**: raster position (relative to the canvas) where the event has occurred, in pixels.
`line`, **`column`**: the grid position in the control where the event has occurred, in grid coordinates.
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)
isdouble(status)
```

They return `1` if the respective key or button is pressed, and `0` otherwise.

`MOUSEMOTION_CB`: called when the mouse moves over the control.

```
int function(Ihandle *self, int line, int column, int x, int y, char *r); [in C]
elem:mousemotion(x, y: number, r: string) -> (ret: number) [in IupLua3]
elem:mousemotion_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.
`line`, **`column`**: the grid position in the control where the event has occurred, in grid coordinates.
`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)
isdouble(r)
```

`DRAW_CB`: called when a specif cell needs to be repainted.

```
int function(Ihandle* self, int line, int column, int xmin, int xmax, int ymin,
elem:drawcb(line, column, xmin, xmax, ymin, ymax: number) -> (ret: number) [in I
elem:draw_cb(line, column, xmin, xmax, ymin, ymax: number) -> (ret: number) [in
```

`line`, **`column`**: the grid position inside the control that is being repainted, in grid coordinates.
`xmin, xmax, ymin, ymax`: the raster bounding box of the repainting cells, where the application can use CD functions to draw anything. If the atributte IUP_CLIPPED is set (the default), all CD graphical primitives is clipped to the bounding region.

The returned value is ignored.

Important note: Inside this callback, the cdActivate() function call is **not** required. Before DRAW_CB is called, the active cdCanvas is properly set; and correctly restored when this function ends. Moreover, all CD attributes are saved and set back for the callback calling, so that the application does not need to deal with the graphical attributes restoration.

WIDTH_CB: called when the controls needs to know a (eventually new) colunm width

```
int function(Ihandle* self, int column);  [in C]
elem:widthcb(column: number) -> (ret: number) [in IupLua3]
elem:width_cb(column: number) -> (ret: number) [in IupLua5]
```

column: the column index

The return value should be an integer that specifies the desired width (in pixels). Negative values will be ignored.

HEIGHT_CB: called when the controls needs to know a (eventually new) line heigth.

```
int function(Ihandle* self, int line);  [in C]
elem:widthcb(line: number) -> (ret: number) [in IupLua3]
elem:width_cb(column: number) -> (ret: number) [in IupLua5]
```

line: the line index

The return value should be an integer that specifies the desired heigth (in pixels). Negative values will be ignored.

NLINES_CB: called when then controls needs to know its number of lines.

```
int function(Ihandle* self);  [in C]
elem:nlinescb() -> (ret: number) [in IupLua3]
elem:nlines_cb() -> (ret: number) [in IupLua5]
```

The return value should be an integer that specifies number of lines. Negative values will be ignored and considered as zero

NCOLS_CB: called when then controls needs to know its number of columns.

```
int function(Ihandle* self);  [in C]
elem:ncolscb() -> (ret: number) [in IupLua3]
elem:ncols_cb() -> (ret: number) [in IupLua5]
```

The return value should be an integer that specifies number of lines. Negative values will be ignored and considered as zero

HSPAN_CB: called when the control needs to know if a cell should be horizontally spanned.

```
int function(Ihandle* self, int line, int column);  [in C]
elem:hspancb(line, column: number) -> (ret: number) [in IupLua3]
elem:hspan_cb(line, column: number) -> (ret: number) [in IupLua5]
```

line, column: the line and colun indexes (in grid coordinates)

The return value should be an integer that specifies the desired span Negative values will be ignored and treated as 1 (no span)

If this callback is not set, all cells will not have any span (default value 1).

VSPAN_CB: called when the control needs to know if a cell should be vertically spanned.

```
int function(Ihandle* self, int line, int column);  [in C]
elem:vspancb(line, column: number) -> (ret: number) [in IupLua3]
elem:vspan_cb(line, column: number) -> (ret: number) [in IupLua5]
```

line, column: the line and colun indexes (in grid coordinates)

The return value should be an integer that specifies the desired span. Negative values will be ignored and treated as 1 (no span)

If this callback is not set, all cells will not have any span (default value 1).

SCROLLING_CB: called when the user right click a cell with the Shift key pressed. It is independent of the SHOW_SECONDARY attribute.

```
int function(Ihandle* self, int line, int column);  [in C]
elem:scrollingcb(line, column: number) -> (ret: number) [in IupLua3]
elem:scrolling_cb(line, column: number) -> (ret: number) [in IupLua5]
```

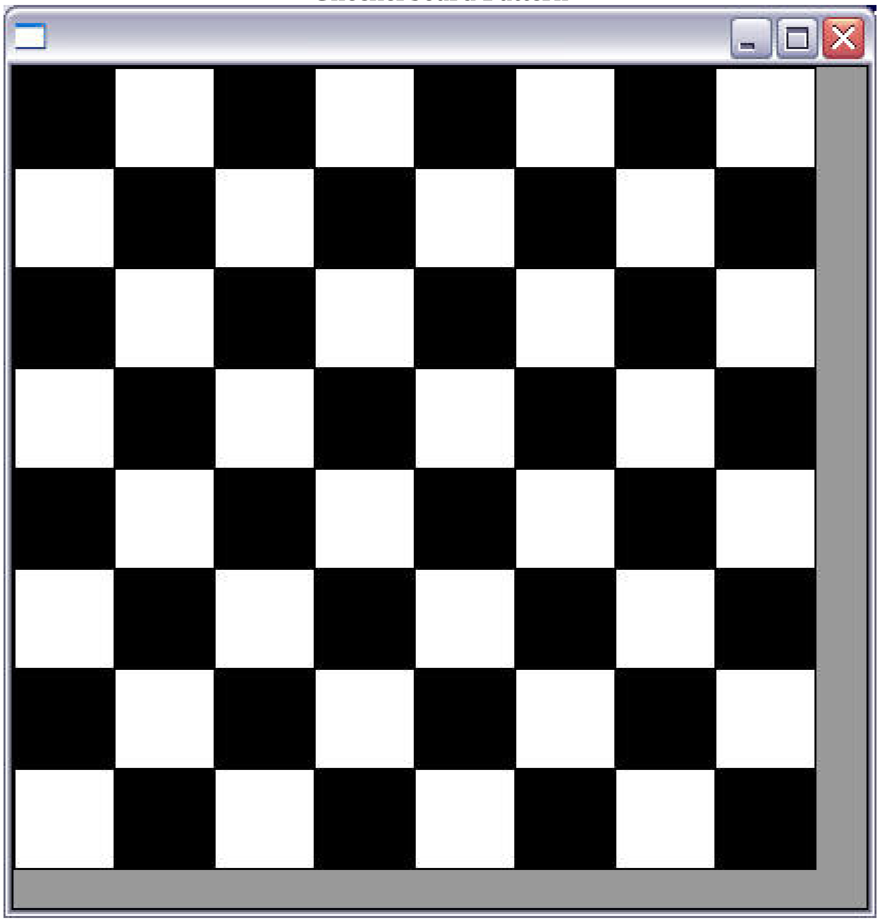
line, column: the first visible line and colunm indexes (in grid coordinates)

The return value should be IUP_DEFAULT is the application wants the grid to be repainted.

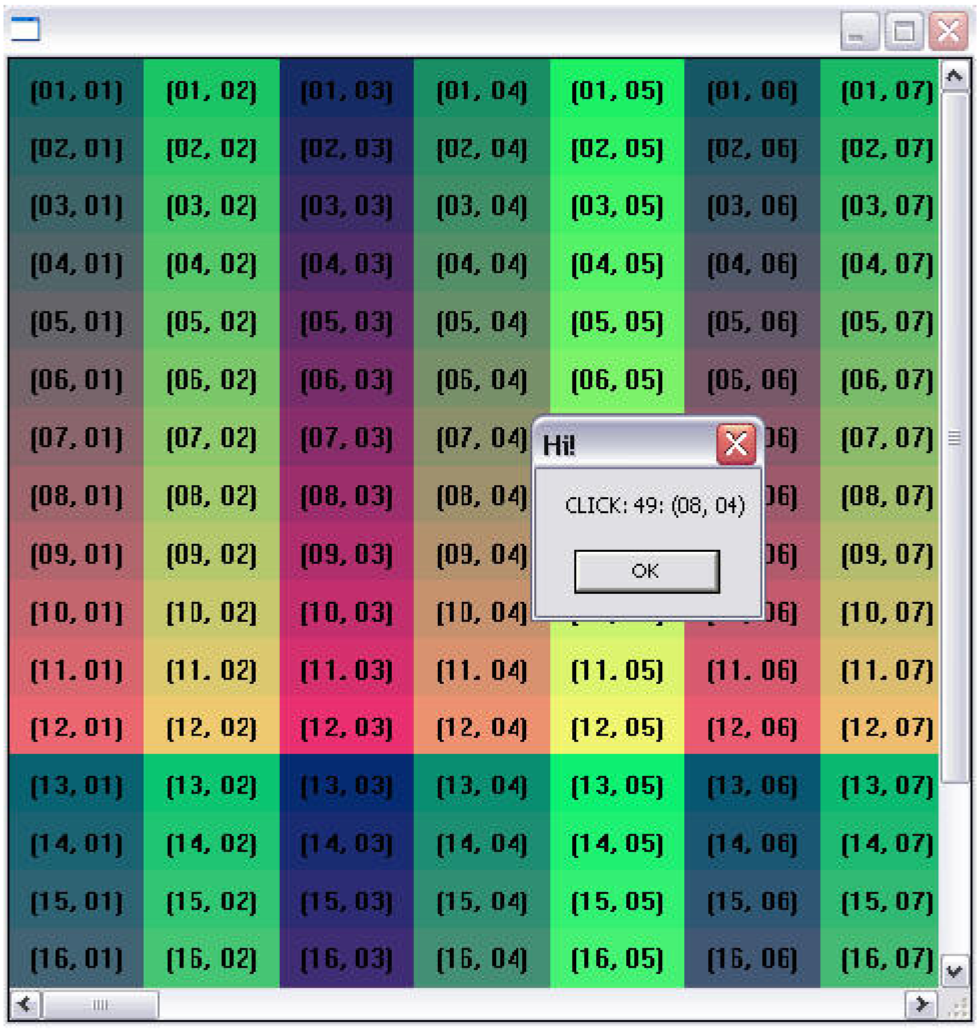
If this callback is not set, all visible cells are redrawn after the scrollbar adjustments.

Examples

Checkerboard Pattern



Numbering Cells



[01, 01]	[01, 02]	[01, 03]	[01, 04]	[01, 05]	[01, 06]	[01, 07]
[02, 01]	[02, 02]	[02, 03]	[02, 04]	[02, 05]	[02, 06]	[02, 07]
[03, 01]	[03, 02]	[03, 03]	[03, 04]	[03, 05]	[03, 06]	[03, 07]
[04, 01]	[04, 02]	[04, 03]	[04, 04]	[04, 05]	[04, 06]	[04, 07]
[05, 01]	[05, 02]	[05, 03]	[05, 04]	[05, 05]	[05, 06]	[05, 07]
[06, 01]	[06, 02]	[06, 03]	[06, 04]	[06, 05]	[06, 06]	[06, 07]
[07, 01]	[07, 02]	[07, 03]	[07, 04]		06]	[07, 07]
[08, 01]	[08, 02]	[08, 03]	[08, 04]		06]	[08, 07]
[09, 01]	[09, 02]	[09, 03]	[09, 04]		06]	[09, 07]
[10, 01]	[10, 02]	[10, 03]	[10, 04]		06]	[10, 07]
[11, 01]	[11, 02]	[11, 03]	[11, 04]	[11, 05]	[11, 06]	[11, 07]
[12, 01]	[12, 02]	[12, 03]	[12, 04]	[12, 05]	[12, 06]	[12, 07]
[13, 01]	[13, 02]	[13, 03]	[13, 04]	[13, 05]	[13, 06]	[13, 07]
[14, 01]	[14, 02]	[14, 03]	[14, 04]	[14, 05]	[14, 06]	[14, 07]
[15, 01]	[15, 02]	[15, 03]	[15, 04]	[15, 05]	[15, 06]	[15, 07]
[16, 01]	[16, 02]	[16, 03]	[16, 04]	[16, 05]	[16, 06]	[16, 07]

Hi!

CLICK: 49: (08, 04)

See Also

[IupCanvas]()

IupColorbar

Creates a color palette to enable a color selection from several samples. It can select one or two colors. The primary color is selected with the left mouse button, and the secondary color is selected with the right mouse button. You can double click a cell to change its color and you can double click the preview area to switch between primary and secondary colors. It inherits from IupCanvas. Originally implemented by André Clinio.

Creation

```
Ihandle* IupColorbar(void); [in C]
iupcolorbar{} -> (elem: ihandle) [in IupLua3]
iup.colorbar{} -> (elem: ihandle) [in IupLua5]
colorbar() [in LED]
```

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

Attributes

`ORIENTATION`: Controls the orientation. It can be `"VERTICAL"` or `"HORIZONTAL"`. Default: `"VERTICAL"`.

NUM_CELLS: Contains the number of color cells. Default: "16". The maximum number of colors is 256. The default colors use the same set of IupImage.

NUM_PARTS: Contains the number of lines or columns. Default: "1".

CELLn: Contains the color of the "n" cell. "n" can be from 0 to NUM_CELLS-1.

PREVIEW_SIZE: Fixes the size of the preview area in pixels. The default size is dynamically calculated from the size of the control. The size is reset to the default when SHOW_PREVIEW=NO.

SHOW_PREVIEW: Controls the display of the preview area. Default: "YES".

SHOW_SECONDARY: Controls the existence of a secondary color selection. Default: "NO".

PRIMARY_CELL: Contains the index of the primary color. Default "0" (black).

SECONDARY_CELL: Contains the index of the secondary color. Default "15" (white).

SQUARED: Controls the aspect ratio of the color cells. Non square cells expand equally to occupy all of the control area. Default: "YES".

SHADOWED: Controls the 3D effect of the color cells. Default: "YES".

BUFFERIZE: Disables the redrawing of the control, so many attributes can be changed without many redraws. Default: "NO".

TRANSPARENCY: Contains a color that will be not rendered in the color pallete. The color cell will have a white and gray chess pattern. It can be used to create a pallete with less colors than the number of cells. Default is not defined.

Callbacks

SELECT_CB: called when a color is selected. The primary color is selected with the left mouse button, and if existant the secondary is selected with the right mouse button.

```
int function(Ihandle* self, int cell, int type);  [in C]
elem:selectcb(cell, type: number) -> (ret: number) [in IupLua3]
elem:select_cb(cell, type: number) -> (ret: number) [in IupLua5]
```

cell: index of the selected cell.
type: indicates if the user selected a primary or secondary color. In can be: IUP_PRIMARY (-1) or IUP_SECONDARY(-2).

If IUP_DEFAULT is returned the selection is accepted. If the callback does not exist the selection is always accepted.

CELL_CB: called when the user double clicks a color cell to change its value.

```
char* function(Ihandle* self, int cell); [in C]
elem:cellcb(cell: number) -> (ret: string) [in IupLua3]
elem:cell_cb(cell: number) -> (ret: string) [in IupLua5]
```

cell: index of the selected cell. If the user double click a preview cell, the respective index is returned.

The callback should return a new color or NULL (nil in Lua) to ignore the change. If the callback does not exist nothing is changed.

SWITCH_CB: called when the user double clicks the preview area outside the preview cells to switch the primary and secondary selections. It is only called if `SHOW_SECONDARY=YES`.

```
int function(Ihandle* self, int prim_cell, int sec_cell);  [in C]
elem:switchcb(prim_cell, sec_cell: number) -> (ret: number) [in IupLua3]
elem:switch_cb(prim_cell, sec_cell: number) -> (ret: number) [in IupLua5]
```

prim_cell: index of the actual primary cell.
sec_cell: index of the actual secondary cell.

If `IUP_DEFAULT` is returned the switch is accepted. If the callback does not exist the switch is always accepted.

`EXTENDED_CB:` called when the user right click a cell with the Shift key pressed. It is independent of the `SHOW_SECONDARY` attribute.

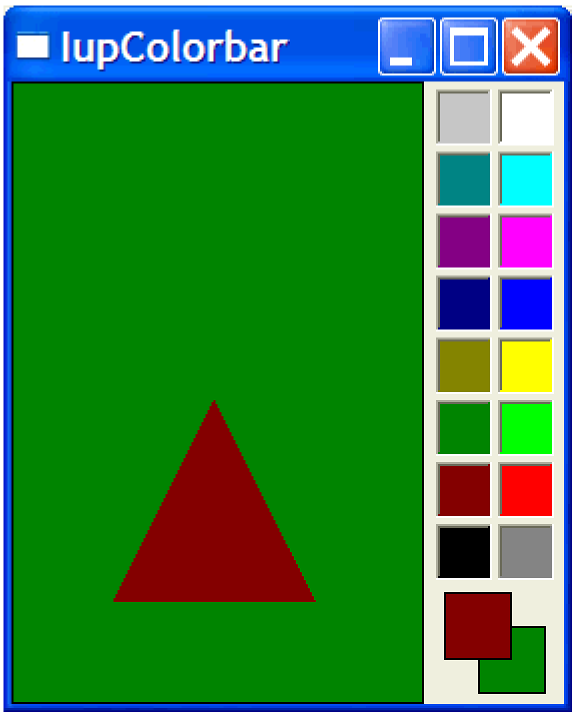
```
int function(Ihandle* self, int cell);  [in C]
elem:extendedcb(cell: number) -> (ret: number) [in IupLua3]
elem:extended_cb(cell: number) -> (ret: number) [in IupLua5]
```

cell: index of the selected cell.

If `IUP_DEFAULT` is returned the control is redrawn.

Examples

Creates a Colorbar for selection of two colors.



See Also

[IupCanvas](), [IupImage]()

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`.

Creation

```
Ihandle* IupColorBrowser(void); [in C]
iupcb{} (elem: ihandle) [in IupLua3]
iup.colorbrowser{} (elem: ihandle) [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); [in (
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); [i
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.

Creation

```
Ihandle* IupDial(char *type); [in C]
iupdial{type: string} -> (elem: ihandle) [in IupLua3]
iup.dial{type: string} -> (elem: ihandle) [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:button_press_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:button_release_cb(angle: number) -> (ret: number) [in IupLua5]
```

Notes

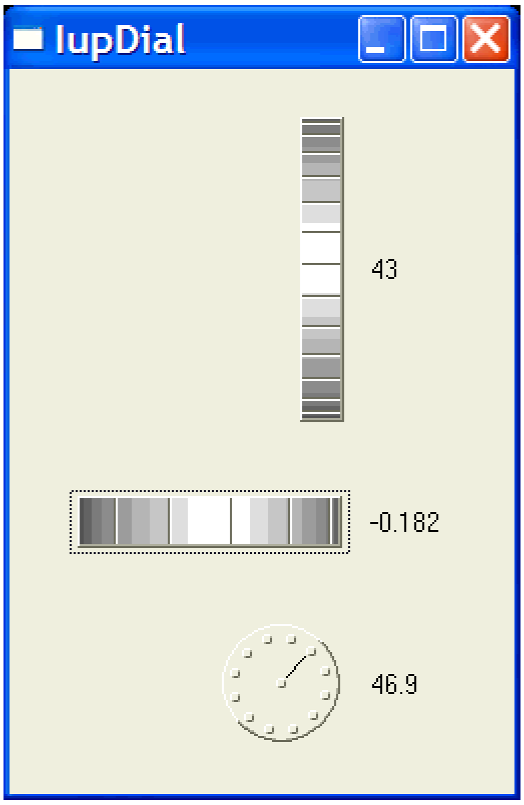
When the keyboard arrows are pressed and released the mouse press and the mouse release callbacks are called in this order. If you hold the key down a mouse move callback is also called.

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].

Creation

```
Ihandle* IupGauge(void); [in C]
iupgauge{} -> (elem: ihandle) [in IupLua3]
iup.gauge{} -> (elem: ihandle) [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".

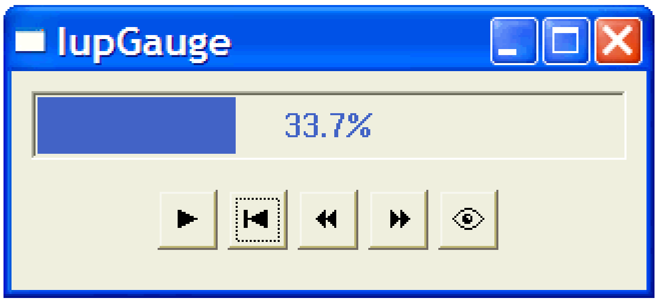
FONT: Character font of the text.

SIZE: The default is "170x17".

EXPAND: The default is "NO".

Examples

Creates a Gauge with a control bar.



See Also

IupCanvas

IupSbox

Creates a split panel control. Allows the provided control to be enclosed in a box that allows resizing. The IupSbox inherits from the IupCanvas, so all the canvas attributes and callbacks are valid. The IupSbox contains a IupZbox where all the children are inserted, and contains another IupCanvas to implement the split handler.

Creation

```
Ihandle* IupSbox(Ihandle* elem); [in C]
iupsbox{elem: ihandle} -> (elem: ihandle) [in IupLua3]
iup.sbox{elem: ihandle} -> (elem: ihandle) [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:

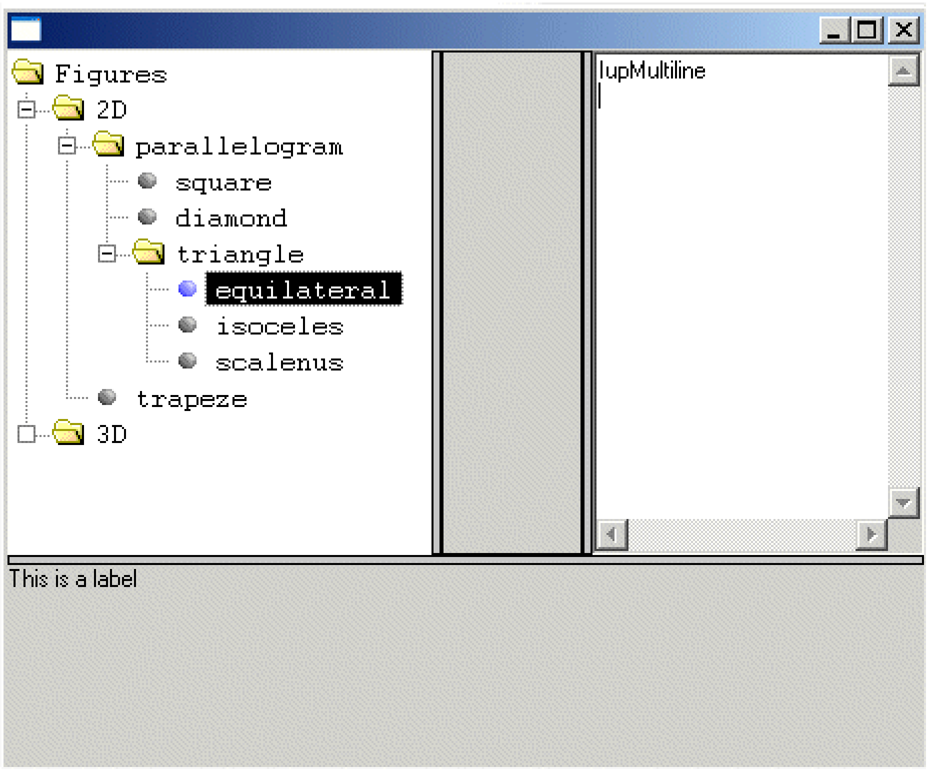
"NORTH", "SOUTH", "EAST", "WEST".

COLOR: Changes the color of the Sbox's thumb. The value should be given in "R G B" color style.

Notes

The controls that you want to be resized must have the EXPAND=YES attribute set.

Examples



Example 2 image

IupSpin and IupSpinBox

This functions will create a control set with a vertical box containing two buttons, one with an up arrow and the other with a down arrow, to be used to increment and decrement values.

Creation

```
Ihandle* IupSpin(void); [in C]
iupspin{} -> (elem: ihandle) [in IupLua3]
iup.spin{} -> (elem: ihandle) [in IupLua5]
```

This function returns the identifier of the created box. It is just a IupVbox with two IupButton.

```
Ihandle* IupSpinbox(Ihandle* ctrl); [in C]
iupspinbox{ctrl: ihandle} -> (elem: ihandle) [in IupLua3]
iup.spinbox{ctrl: ihandle} -> (elem: ihandle) [in IupLua5]
```

This function returns the identifier of the created box. This will create a IupHbox with the additional control and a IupSpin set.

Callbacks

SPIN_CB: Called each time the user clicks in the buttons. It will increment 1 and decrement -1 by default. Holding Shit will set a factor of 2, holding Ctrl a factor of 10, and both a factor of 100.

```
int function(Ihandle *self, int inc); [in C]
elem:spincb(inc: number) -> (ret: number) [in IupLua3]
elem:spin_cb(inc: number) -> (ret: number) [in IupLua5]
```

See Also

IupGetParam

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. The IupZbox is a child of the IupCanvas, so all the attributes set in the Tabs will affect its child by attribute inheritance.

Creation

```
Ihandle* IupTabs(Ihandle* elem1, Ihandle* elem2, ...); [in C]
Ihandle* IupTabsv(Ihandle** elems); [in C]
iuptabs{elem1, elem2, ...: ihandle} -> (elem: ihandle) [in IupLua3]
iup.tabs{elem1, elem2, ...: ihandle} -> (elem: ihandle) [in IupLua5]
tabs(elem1, elem2, ...) [in LED]
```

elem1, elem2, ...: List of the elements that will be placed into Tabs.

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: In this case it is propagated to the Zbox when changed (ALIGMENT is one of the attributes that are not inherited from the parent). See the IupZbox documentation.

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".

TABORIENTATION: Indicates the orientation of tab text, which can be one of the following:

"HORIZONTAL" or "VERTICAL". Default is "HORIZONTAL".

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 child. If both are defined, the size of the tab child will have priority over the global `IupTabs` size.

`VALUE`: Changes the active tab. The value passed must be the name of one of the elements contained in the `tabs`. Default: the first element. To set the name of an element, use the `IupSetHandle` function. In Lua you can also use the element reference directly.

`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: ihandle) -> (ret: number) [in IupLua3]
elem:tabchange_cb(new_tab, old_tab: ihandle) -> (ret: number) [in IupLua5]
```

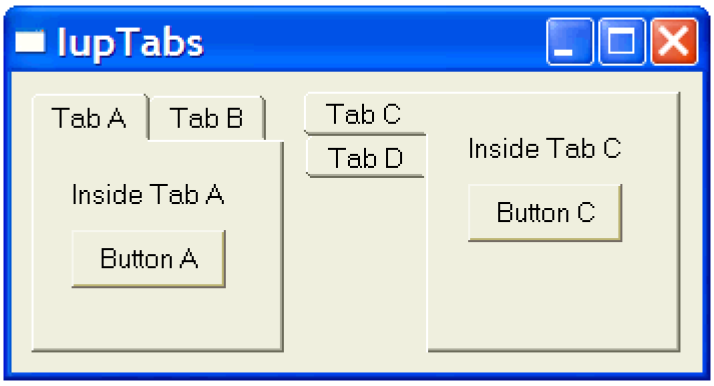
`self`: `Ihandle*` of the control
`new_tab`: `Ihandle*` of the tab selected by the user
`old_tab`: `Ihandle*` of the previously selected tab

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.

When you change the active tab the focus is not changed. If you want to control the focus behavior call IupSetFocus in the TABCHANGE_CB callback.

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.

Creation

```
Ihandle* IupVal(char *type); [in C]
iupval{type: string} -> (elem: ihandle) [in IupLua3]
iup.val{type: string} -> (elem: ihandle) [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

TYPE: Informs whether the valuator is "VERTICAL" or "HORIZONTAL". Vertical valuators are bottom to up, and horizontal valuators are left to right variations of min to max.

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.

STEP: Controls the increment for keyboard control and the mouse wheel.

PAGESTEP: Controls the increment for pagedown and pageup keys.

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:button_press_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.

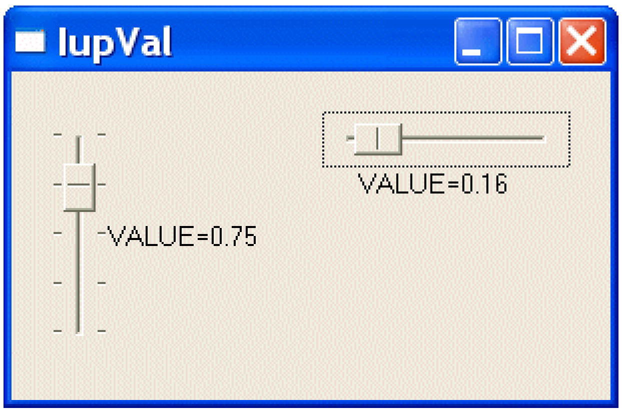
```
int function(Ihandle *self, double val); [in C]
elem:buttonrelease(val: number) -> (ret: number) [in IupLua3]
elem:button_release_cb(val: number) -> (ret: number) [in IupLua5]
```

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 existance of this callback defines the mode the matrix will operate.

Creation

```
Ihandle* IupMatrix(char *action); [in C]
iupmatrix{} -> (elem: ihandle) [in IupLua3]
iup.matrix{} -> (elem: ihandle) [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
SORTSIGN

Size Attributes

NUMCOL
NUMCOL_VISIBLE
NUMLIN
NUMLIN_VISIBLE
WIDTHDEF
WIDTHn
HEIGHTn
RESIZEMATRIX

Mark Attributes

AREA
MARK_MODE
MARKED
MULTIPLE

Action Attributes

ADDCOL
ADDLIN
DELCOL
DELLIN
EDIT_MODE
ORIGIN
REDRAW

General Attributes

CURSOR
FRAMECOLOR
SCROLLBAR
SIZE
CARET
SELECTION
HIDEFOCUS

Callbacks

Interaction

ACTION - Action generated when a keyboard event occurs.
CLICK_CB - Action generated when any mouse button is pressed over a cell.
MOUSEMOVE_CB - Action generated to notify the application that the mouse has moved over the matrix.
ENTERITEM_CB - Action generated when a matrix cell is selected, becoming the current cell.
LEAVEITEM_CB - Action generated when a cell is no longer the current cell.
SCROLL_CB - Action generated when the matrix is scrolled with the scrollbars or with the keyboard.

Drawing

BGCOLOR_CB - Action generated to retrieve the background color of a cell when it needs to be redrawn.
FGCOLOR_CB - Action generated to retrieve the foreground color of a cell when it needs to be redrawn.
DRAW_CB - Action generated before the cell is drawn. Allow a custom cell draw.
DROPCHECK_CB - Action generated to determine if a dropdown feedback should be shown.

Editing

DROP_CB - Action generated to determine if a text field or a dropdown will 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.

Callback Mode

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.

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 tradicional 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, "BGCOLOR30:10
IupMatSetAttribute (n, "ALIGNMENT" , 10, 0, v) = IupSetAttribute(n, "ALIGNMENT10

(*) noticed that in this case the second value will be ignored.
```

Notes

The `IupMask` control can be used to create a mask and filter the text entered by the user in each cell.

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.

When allowed the width of a column can be changed holding and dragging its title right border.

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.)

Navigation

Navigating through the matrix cells outside the edition mode is done by using the following keys:

- **Arrows**: Moves the focus to the next cell, according to the arrow's direction.
- **Page Up** and **Page Down**: Moves a visible page up or down.
- **Home**: Moves the focus to the fist column in the line.

- **Home Home**: Moves the focus to the upper left corner of the visible page.
- **Home Home Home**: Moves the focus to the upper left corner of the first page of the matrix.
- **End**: Moves the focus to the last column in the line.
- **End End**: Moves the focus to the lower right corner of the visible page.
- **End End End**: Moves the focus to the lower right corner of the last page in the matrix.

Inside the **edition mode**, the following keys are used for a text field:

- **Up and down arrows**: Leave the edition mode and moves the focus accordingly.
- **Left and right arrows**: If the caret is at the extremes of the text being edited then leave the edition mode and moves the focus accordingly.
- **Ctrl + Arrows**: Leave the edition mode and moves the focus accordingly.

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 with the current text of the cell. If **Del** is pressed, the whole contents of the cell will be deleted. Double-clicking a cell also enters the edition mode. In Motif, when start editing using a double click, the user must click again to the edit control get the focus.

When the matrix is in the edition mode, to confirm the entered value, press **Enter**. By pressing **Esc**, the previous value is restored. The cell will also leave the edition mode if the user clicked in another cell or in another control, then the value will be confirmed. When pressing **Enter** to confirm the value the focus goes to the cell bellow the current cell, if at the last line then the focus goes to the cell on the left. The value confirmation depends on the `EDITION_CB` callback return code.

Marks

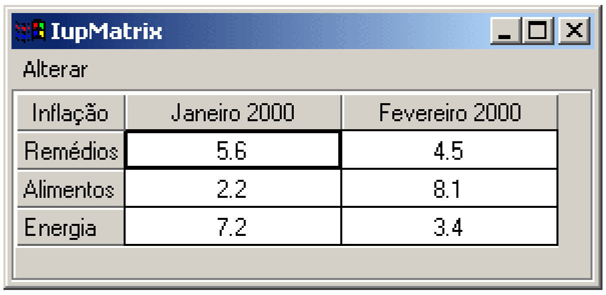
When mark mode is active the cells can be marked using mouse, if the keyboard is used all marks are cleared.

A marked cell will have its background attenuated to indicate that it is marked.

Cells can be selected individually or the marks can be restricted to lines and/or columns. Also multiple cells can be marked simultaneously in continuous or in segmented areas. Lines and columns are marked only when the user clicks in their respective titles. Continuous areas are marked holding and dragging the mouse or holding the **Shift** key. Segmented areas are marked holding the **Ctrl** key.

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.
***L*:0**: Title of line *L*.
0:*C*: Title of column *C*.
0:0: Title of the area between the line and column titles.

These are valid only in normal mode. No redraw is done for all cases, the application must explicity set the REDRAW attribute.

ALIGNMENT*N*: Alignment of the cells in column *N*, where n must be replaced by the wished column number (n >= 0). No redraw is done. Possible values:

> "ALEFT", "ACENTER" or "ARIGHT". Default: "ALEFT".

BGCOLOR: Background color of the matrix.
BGCOLOR*L*:*: Background color of the cells in line *L* (no redraw is done for this case).
BGCOLOR*:*C*: Background color of the cells in column *C* (no redraw is done for this case).
BGCOLOR*L*:*C*: Background color of the cell in line *L* and column *C* (no redraw is done for this case).

> 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.
FGCOLOR*L*:*: Text color of the cells in line *L* (no redraw is done for this case).
FGCOLOR*:*C*: Text color of the cells in column *C* (no redraw is done for this case).
FGCOLOR*L*:*C*: Text color of the cell in line *L* and column *C* (no redraw is done for this case).

> 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.
FONT*L*:*: Text font of the cells in line L (no redraw is done for this case).
FONT*:*C*: Text font of the cells in column C (no redraw is done for this case).
FONT*L*:*C*: Text font of the cell in line L and column C (no redraw is done for this case).

> This attribute must be set before the control is showed. It affects the calculation of the size of all the matrix cells. 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). When retrieved inside the EDITION_CB callback when mode is 0, then the return value is the new value that will be updated in the cell.

SORTSIGN*C*: Shows a sort sign (up or down arrow) in the column *C* title. Possible values: "UP", "DOWN" and "NO". Default: NO.

Size Attributes

NUMCOL: Defines the number of columns in the matrix.

Must be an integer number. Default: "0".

NUMCOL_VISIBLE: When set defines the minimum number of visible columns in the matrix. When retrieved returns the current number of visible lines. 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: When set defines the minimum number of visible lines in the matrix. When retrieved returns the current number of visible lines. The remaining lines will be accessible only by using the scrollbar.

Must be an integer number. Default: "3".

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 right border. 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 x 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".

ORIGIN: Scroll the visible area to the given cell. Returns the cell at the upper left corner. To move only a line or a column, use a value such as "L:*"or "*:C" (L and C>=1). Possible values: two numbers in the "L:C" format.

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)

No redraw is done when the application sets cell, line or column graphics attributes attributes: **0:0**, **0:C**, **L:0**, **L:C**, **ALIGNMENTn**, **BGCOLORL:***, **BGCOLOR*:C**, **BGCOLORL:C**, **FGCOLORL:***, **FGCOLOR*:C**, **FGCOLORL:C**, **FONTL:***, **FONT*:C**, **FONTL:C**. Global and size attributes always automatically redraw the matrix.

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.

CARET: Allows specifying and verifying the caret's position of the text box in edition mode.

SELECTION: Allows specifying and verifying selection interval of the text box in edition mode.

HIDEFOCUS: do not show the focus mark when drawing the matrix. Default is NO.

IupMatrix Callbacks

Interaction

ACTION: Action generated when a keyboard event occurs.

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

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.

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), isdouble(r)`. They return `1` if the respective key or button is pressed, or `0` otherwise.

To interrupt further internal processing return IUP_IGNORE.

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.

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`. This callback is also called when matrix is getting focus.

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`. This callback is also called when the matrix is losing focus. Returning `IUP_IGNORE` prevents the current cell from changing, but this will not work when the matrix is losing focus. If you try to move to beyond matrix borders the cell will lose focus and then get it again, so leaveitem and enteritem will be called.

`SCROLL_CB`: Action generated when the matrix is scrolled with the scrollbars or with the keyboard. 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`.

Drawing

`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 int *g
elem:bgcolorcb(lin, col: number) -> (red, green, blue, ret: number) [in IupLua3]
elem:bgcolor_cb(lin, col: number) -> (red, green, blue, ret: number) [in IupLua5
```

`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.

`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 *g
elem:fgcolorcb(lin, col: number) -> (red, green, blue, ret: number) [in IupLua3]
elem:fgcolor_cb(lin, col: number) -> (red, green, blue, ret: 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.

`DRAW_CB`: Action generated before a cell is drawn. Allows to draw a custom cell contents. You must use the CD library primitives.

```
int function(Ihandle *self, int lin, int col, int x1, int x2, int y1, int y2); [
elem:draw(lin, col, x1, x2, y1, y2: number) -> (ret: number) [in IupLua3]
elem:draw_cb(lin, col, x1, x2, y1, y2: number) -> (ret: number) [in IupLua5]
```

self: Identifier of the matrix interacting with the user.
lin, **col**: Coordinates of the current cell.
x1, **x2**, **y1**, **y2**: Bounding rectangle of the current cell in pixels.

If this function return `IUP_IGNORE` the normal text drawing will take place. The clipping is set for the bounding rectangle. The callback is called after the cell background has been filled with the background color. If `HIDEFOCUS=NO` (the default) the drawing area will not include the focus area, if `HIDEFOCUS=YES` the complete cell is available.

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.

Editing

DROP_CB: Action generated before the current cell enters edition mode to determine if a text field or a dropdown will be shown. It is called after `EDITION_CB`. 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: ihandle, lin, col: number) -> (ret: number) [in IupLua3]
elem:drop_cb(drop: ihandle, 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. If returns `IUP_CONTINUE` the value is accepted as a new value and the matrix leaves edition mode.

```
int function(Ihandle *self, int lin, int col, Ihandle *drop, char *t, int i, int
elem:dropselect(lin, col: number, drop: ihandle, t: string, i, v: number) -> (re
elem:dropselect_cb(lin, col: number, drop: ihandle, t: string, i, v: number) ->
```

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 mode);  [in C]
elem:edition(lin, col, mode: number) -> (ret: number) [in IupLua3]
elem:edition_cb(lin, col, mode: number) -> (ret: number) [in IupLua5]
```

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 new value is accepted only if the callback returns `IUP_DEFAULT`. If the callback does not exists the new value is always accepted. If the user pressed **Enter** and the callback returns `IUP_IGNORE` the editing continues. If the callback returns `IUP_CONTINUE` when mode is 0, the edit mode is ended but the value is not updated, so the application can update a different value (usefull to format the new value). If the control loses its focus the edition mode will be ended always even if the callback return `IUP_IGNORE.`

This callback is also called if the user cancel the editing with **Esc** and when the user press **Del** to validate the operation for each cell been cleared (in this case is called only with mode=1).

Callback Mode

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 existance 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). This callback is usually set in callback mode, but also works in normal mode.

```
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 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.

Creation

```
Ihandle* IupTree(void); [in C]
iuptree{} -> (elem: ihandle) [in IupLua3]
iup.tree{} -> (elem: ihandle) [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
SHOWDRAGDROP
SHOWRENAME
RENAMECARET
RENAMESELECTION

Marks

CTRL
SHIFT
STARTING
VALUE
MARKED

Images

IMAGELEAF
IMAGEBRANCHCOLLAPSED
IMAGEBRANCHEXPANDED

IMAGEid
IMAGEEXPANDEDid

Nodes

NAME
STATE
DEPTH
KIND
PARENT
COLOR

Action

ADDLEAF
ADDBRANCH
DELNODE
REDRAW
RENAMENODE

Callbacks

SELECTION_CB: Action generated when an node is selected or deselected.
MULTISELECTION_CB: Action generated when multiple nodes are selected with the mouse and the shift key pressed.
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.
RENAME_CB: Action generated when a node new name was entered.
SHOWRENAME_CB: Action generated when a new name is about to be entered.
DRAGDROP_CB: Action generated when a drag & drop is executed.
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.

Colors

The IupTree colors are fixed by definition. The tree background color is "255 255 255" in Windows and "156 156 165" in Motif. The tree selected node color is "8 33 107" in Windows and "0 0 156" in Motif.

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 brach 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.
- **Ctrl+Space**: Marks or unmarks a node.
- **F2**: Calls the rename callback or invoke the inplace rename.

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 a collapsed branch, it is expanded; if it is a leaf, it is executed.
- **Double-clicking a node text**: Calls the rename callback or invoke the inplace rename.

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: ihandle, id: number, userid: userdata) [in IupLua3]
iup.TreeSetUserId(self: ihandle, id: number, userid: userdata) [in IupLua5]
```

`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: ihandle, id: number) -> (ret: userdata) [in IupLua3]
iup.TreeGetUserId(self: ihandle, id: number) -> (ret: userdata) [in IupLua5]
```

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: ihandle, userid: userdata) -> (ret: number) [in IupLua3]
iup.TreeGetId(self: ihandle, userid: userdata) -> (ret: number) [in IupLua5]
```

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: ihandle, id: number, table: table) [in IupLua3]
iup.TreeSetTableId(self: ihandle, 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: ihandle, table: table) -> (ret: number) [in IupLua3]
iup.TreeGetTableId(self: ihandle, 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: ihandle, id: number) -> (ret: table) [in IupLua3]
iup.TreeGetTable(self: ihandle, 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.

Utility Functions

These functions can be used to help set and get attributes from the matrix:

```
void  IupTreeSetAttribute (Ihandle *n, char* a, int id, char* v);
void  IupTreeStoreAttribute(Ihandle *n, char* a, int id, char* v);
char* IupTreeGetAttribute (Ihandle *n, char* a, int id);
int   IupTreeGetInt (Ihandle *n, char* a, int id);
float IupTreeGetFloat (Ihandle *n, char* a, int id);
void  IupTreeSetfAttribute (Ihandle *n, char* a, int id, char* f, ...);
```

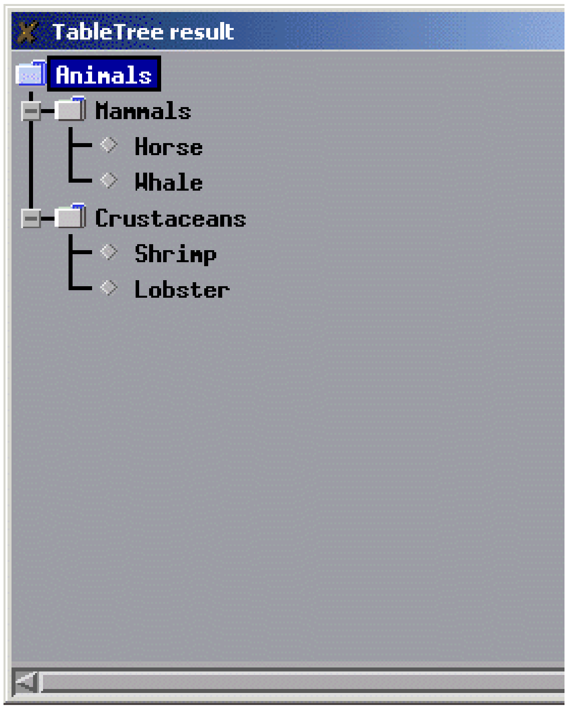
They work just like the respective tradicional set and get functions. But the attribute string is complemented with the L and C values. For ex:

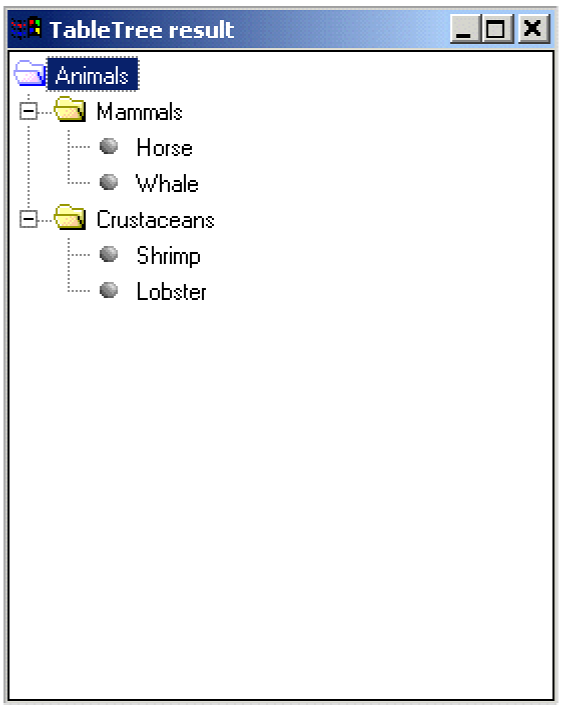
```
IupTreeSetAttribute(n, "KIND" , 30, v) = IupSetAttribute(n, "KIND30", v)
IupTreeSetAttribute(n, "ADDLEAF" , 10, v) = IupSetAttribute(n, "ADDLEAF10", v)
```

See also the IupTreeUtil by Sergio Maffra and Frederico Abraham. It is an utility wrapper in C++ for the IupTree.

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"`.

SHOWDRAGDROP: Shows a drag cursor if the user drags a node or branch and enables the **DRAGDROP_CB** callback. Default: "NO".

SHOWRENAME: Allows the in place rename of a node. The **RENAME_CB** is called after the user entered a new name, instead of calling **RENAME_NODE_CB** to notify the user that it should rename a node. Default: "NO".

RENAMECARET: Allows specifying and verifying the caret's position of the text box when in-place renaming.

RENAMESELECTION: Allows specifying and verifying selection interval of the text box when in-place renaming.

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 is 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 unmaked. 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 unmaked. 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.

RENAME: Forces a rename action to take place. If SHOWRENAME=YES then does in-place rename, else just calls the RENAMENODE_CB. 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.

MULTISELECTION_CB: Action generated when multiple nodes are selected with the mouse and the shift key pressed.

```
int function(Ihandle *self, int* ids, int n) [in C]
elem:multiselection(ids: table, n: number) -> (ret: number) [in IupLua3]
elem:multiselection_cb(ids: table, n: number) -> (ret: number) [in IupLua5]
```

self: Identifier of the IupTree interacting with the user.
ids: Array of node identifiers.
n: Number of nodes in the array.

This function must return IUP_IGNORE for the selected nodes 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) or press **F2**, and **SHOWRENAME**=NO.

```
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 IupLua5]
```

self: Identifier of the IupTree interacting with the user.
id: Identifier of the clicked node.
name: Current name of the clicked node.

SHOWRENAME_CB: Action generated when a node is to be renamed in place and **SHOWRENAME**=**YES**. It occurs only when the user double clicks the **text** associated to a node (leaf or branch) or press **F2**, and **SHOWRENAME**=**YES**.

```
int function(Ihandle *self, int id); [in C]
elem:showrenamecb(id: number: string) -> (ret: number) [in IupLua3]
elem:showrename_cb(id: number: string) -> (ret: number) [in IupLua5]
```

`self`: Identifier of the `IupTree` interacting with the user.
`id`: Identifier of the clicked node.

`RENAME_CB`: Action generated after a node was renamed in place. It occurs when the user press **Enter** after editing the name, or when the text box looses it focus.

```
int function(Ihandle *self, int id, char *name); [in C]
elem:renamecb(id: number, name: string) -> (ret: number) [in IupLua3]
elem:rename_cb(id: number, name: string) -> (ret: number) [in IupLua5]
```

`self`: Identifier of the `IupTree` interacting with the user.
`id`: Identifier of the clicked node.
`name`: New name of the clicked node.

The new name is accepted only if the callback returns IUP_DEFAULT. If the callback does not exists the new name is always accepted. If the user pressed **Enter** and the callback returns IUP_IGNORE the editing continues. If the text box looses its focus the editing stops always.

`DRAGDROP_CB`: Action generated when a drag & drop is executed. Only active if **`SHOWDRAGDROP=YES.`**

```
int function(Ihandle *self, int drag_id, int drop_id, int isshift, int iscontrol
elem:dragdrop(drag_id, drop_id, isshift, iscontrol: number) -> (ret: number) [in
elem:dragdrop_cb(drag_id, drop_id, isshift, iscontrol: number) -> (ret: number)
```

`self`: Identifier of the `IupTree` interacting with the user.
`drag_id`: Identifier of the clicked node where the drag start.
`drop_id`: Identifier of the clicked node where the drop were executed.
`isshift`: Boolean flag indicatinf the shift key state.
`iscontrol`: Boolean flag indicatinf the control key state.

`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`.

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, **`iupgllua_open`**, after calling **`iuplua_open`**. The `iupluagl.h` file must also be included in the source code. The program must be linked to the control's libraries (`iupluagl[5].lib` on Windows and `libiupluagl[5].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).

Creation

```
Ihandle* IupGLCanvas(char* action); [in C]
iupglcanvas{} -> (elem: ihandle) [in IupLua3]
iup.glcanvas{} -> (elem: ihandle) [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`.

Addicionally:

`RESIZE_CB`: By default the resize callback sets:

```
glViewport(0,0,width,height);
```

Auxiliary Functions

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

Activates the given canvas as the current OpenGL context. All subsequent OpenGL commands are directed to such canvas.

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

Returns a non zero value if the given canvas is the current OpenGL context.

`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

IMPORTANT: An OpenGL canvas when put inside an IupFrame will not work.

Examples

See Also

[IupCanvas]

IupOleControl

The IupOleControl hosts an windows OLE control (also named ActiveX control), allowing it to be used inside IUP dialogs. There are many OLE controls available, like calendars, internet browsers, PDF readers etc.

Notice that IupOleControl just takes care of the visualization of the control (size and positioning). It does not deal with properties, methods and events. The application must deal with them using the the COM interfaces offered by the control. Nevertheless, using IupLua together with LuaCOM makes it possible to use OLE controls very easily in Lua, accessing their methods, properties and events similarly to the other IUP elements.

Notice that this control works only on Windows, using Visual C++ or Borland C++.

Initialization and usage

The **`IupOleControlOpen`** function must be called after a **`IupOpen`**, so that the control can be used. The `iupole.h` file must also be included in the source code. The program must be linked to the control’s library `iupole.lib`.

To make the control available in Lua, use the initialization function in C, **`iupolelua_open`**, after calling **`iuplua_open`**. The `iupluaole.h` file must also be included in the source code. The program must be linked to the control’s library: `iupluaole[5].lib`.

Creation

```
Ihandle* IupOleControl(char* ProgID); [in C]
iup.olecontrol{ProgID: string} -> (elem: ihandle) [in IupLua5]
```

`ProgID`: the programmatic identifier of the OLE control. This can be found in the documentation of the OLE control or by browsing the list of registered controls, using tools like OleView.

The function returns the OLE control created or `NULL` if an error occurs.

Attributes

`DESIGNMODE:` Returns if the control is in design mode. Some controls behave differently when in design mode. See this article for more information about design mode. Default value: `"NO".`

`IUNKNOWN:` Returns the IUnknown pointer for the control. This pointer is necessary to access methods and properties of the control in C/C++ code. This is a read-only attribute.

The control's specific attributes shall be accessed using the COM mechanism (see section below for more information).

Callbacks

In C/C++, the OLE control's callbacks (events, in ActiveX terms) shall be set using the control's interface and the COM mechanism. When using IupLua, it's possible to call methods, access properties and receive events from the OLE control using the LuaCOM library. When the LuaCOM library is loaded, for each OLE control created a LuaCOM object is also created and stored in the `com` field of the object returned by **`iup.olecontrol`**. This LuaCOM object can be used to access properties, methods and events in a way very similar to VB. See the examples for more information.

Notes

To learn more about OLE and ActiveX:

http://www.microsoft.com/com
http://www.webopedia.com/TERM/A/ActiveX_control.html
http://msdn.microsoft.com/workshop/components/activex/activex_node_entry.asp
http://activex.microsoft.com/activex/activex/

Examples

See Also

`IupCanvas`

IupSpeech

Creates a speech engine that allows speech recognition and speech.

[Windows only]

Initialization and Usage

The **`IupSpeechOpen`** function must be called after a **`IupOpen`**, so that the `Speech` control can be used (no binding available yet.)

To generate an application that uses this control, the program must be linked to the control's library (`iupspeech.lib` on Windows). The **`iupspeech.h`** file must also be included in the source code.

The Microsoft Speech SDK 5.1 must be installed in the system.

Creation

```
Ihandle* IupCreate("speech"); [in C]
[Not available in IupLua]
```

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

Attributes

`GRAMMAR`: Accepts a full directory path to a .xml file defining the grammar that the speech engine will be considering. Only one grammar allowed.

`SAY`: Speaks the given text.

Callbacks

`ACTION_CB`: Called when the engine recognizes a word/sentence.

int function(Ihandle ***self**, char *text); [in C]

self: Ihandle.

text: Full recognized text based on given grammar.

Comments

Check Speech SDK help file for more information on how to create an input xml file.

IUP's speech interface will create a "shared-recognizer", i.e., a unique recognizer that will work for the entire system. That means that the input focus can be anywhere in the system and still the recognition will be triggered.

Only one process in the system can initialize IupSpeech.

The system is greatly improved by training. Look in the control-panel in the Speech tab for more details.

Available only in Microsoft Visual Studio .NET.

Examples

Keyboard

Keyboard navigation is the dialog uses the "Tab" key to change the keyboard focus from one control to another. All IUP interactive controls have Tab stops, but the navigation order is related to the order the controls are placed in the dialog and can not be changed. When the focus is at a Multiline control to change focus the combination "Ctrl+Tab" must be used, because "Tab" is a valid character for the Multiline. The application can also control the focus using the functions: `IupGetFocus`, `IupSetFocus`, `IupNextField` and `IupPreviousField`. And when the focus is changed the application is notified trough the callbacks `GETFOCUS_CB` and `KILLFOCUS_CB`.

Two keys are also important in keyboard navigation: "Enter" and "Esc". But they are only active is the application register the attributes `DEFAULTENTER` and `DEFAULTESC` of the `IupDialog`. These attributes configure buttons to be activated when the respective key is pressed. Again "Enter" is a valid key for the Multiline so the combination "Ctrl+Enter" must be used instead. If the focus is at a button then the Enter key will activate this button independent from the `DEFAULTENTER` attribute.

Usualy the application will process keyboard input in the canvas using the `KEYPRESS_CB` callback. But there is also the `K_ANY` callback that can be used for all the controls, but it does not have control of the press state. Both callbacks use the key codification explained in Keyboard Codes. These codes are also used in the `ACTION` callbacks of `IupText` and `IupMultiline`, and in shortcuts for menu items and submenus using the `KEY` attribute of `IupItem` and `IupSubmenu`. Finally all the keyboard codes can be used as callback names to implement application hot keys.

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. Normally these keys are used in K_ANY and KEYPRESS_CB callbacks to inform the key that was pressed in the keyboard.

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. Except for the Brazilian ABNT keyboard which works in Windows and Linux. This does not affect the IupText and IupMultiline text input.

Notice that somes 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. This macro is also available in Lua as a function with the same name.

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
SPACE	K_SP
!	K_exclam
"	K_quotedbl
#	K_numbersign
$	K_dollar
%	K_percent
&	K_ampersand
'	K_apostrophe
(	K_parentleft
)	K_parentright
*	K_asterisk
+	K_plus
,	K_comma
-	K_minus
.	K_period
/	K_slash
0	K_0
1	K_1
2	K_2
3	K_3
4	K_4
5	K_5
6	K_6
7	K_7
8	K_8
9	K_9
:	K_colon

Key	Code / Attribute
Alt-A	K_mA
Alt-B	K_mB
Alt-C	K_mC
Alt-D	K_mD
Alt-E	K_mE
Alt-F	K_mF
Alt-G	K_mG
Alt-H	K_mH
Alt-I	K_mI
Alt-J	K_mJ
Alt-K	K_mK
Alt-L	K_mL
Alt-M	K_mM
Alt-N	K_mN
Alt-O	K_mO
Alt-P	K_mP
Alt-Q	K_mQ
Alt-R	K_mR
Alt-S	K_mS
Alt-T	K_mT
Alt-U	K_mU
Alt-V	K_mV
Alt-W	K_mW
Alt-X	K_mX
Alt-Y	K_mY
Alt-Z	K_mZ
Alt-1	K_m1

Key	Code / Attribute
Ctrl-SPACE	K_cSP
Ctrl-A	K_cA
Ctrl-B	K_cB
Ctrl-C	K_cC
Ctrl-D	K_cD
Ctrl-E	K_cE
Ctrl-F	K_cF
Ctrl-G	K_cG
Ctrl-H	K_cH
Ctrl-I	K_cI
Ctrl-J	K_cJ
Ctrl-K	K_cK
Ctrl-L	K_cL
Ctrl-M	K_cM
Ctrl-N	K_cN
Ctrl-O	K_cO
Ctrl-P	K_cP
Ctrl-Q	K_cQ
Ctrl-R	K_cR
Ctrl-S	K_cS
Ctrl-T	K_cT
Ctrl-U	K_cU
Ctrl-V	K_cV
Ctrl-W	K_cW
Ctrl-X	K_cX
Ctrl-Y	K_cY
Ctrl-Z	K_cZ

K
Shift- SPAC
Shift-
Shift-
Shift-
Shift-
Shift-
Shift-
Shift-
Shift-
Shift-
Shift-
Shift-
Shift-
Shift-
Shift-
Shift-
Shift-
Shift-
Shift-
Shift-
Shift-
Shift-
Shift-
Shift-
Shift-

;	K_semicolon
<	K_less
=	K_equal
>	K_greater
?	K_question
@	K_at
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

Alt-2	K_m2
Alt-3	K_m3
Alt-4	K_m4
Alt-5	K_m5
Alt-6	K_m6
Alt-7	K_m7
Alt-8	K_m8
Alt-9	K_m9
Alt-0	K_m0
Alt-Tab	K_mTAB
Alt-Home	K_mHOME
Alt-UP	K_mUP
Alt-PgUp	K_mPGUP
Alt-LEFT	K_mLEFT
Alt-RIGHT	K_mRIGHT
Alt-END	K_mEND
Alt-DOWN	K_mDOWN
Alt-PgDn	K_mPGDN
Alt-Insert	K_mINS
Alt-Del	K_mDEL
Alt-F1	K_mF1
Alt-F2	K_mF2
Alt-F3	K_mF3
Alt-F4	K_mF4
Alt-F5	K_mF5
Alt-F6	K_mF6
Alt-F7	K_mF7
Alt-F8	K_mF8
Alt-F9	K_mF9
Alt-F10	K_mF10

Ctrl-Tab	K_cTAB
Ctrl-Home	K_cHOME
Ctrl-UP	K_cUP
Ctrl-PgUp	K_cPGUP
Ctrl-LEFT	K_cLEFT
Ctrl-MIDDLE	K_cMIDDLE
Ctrl-RIGHT	K_cRIGHT
Ctrl-END	K_cEND
Ctrl-DOWN	K_cDOWN
Ctrl-PgDn	K_cPGDN
Ctrl-Insert	K_cINS
Ctrl-Del	K_cDEL
Ctrl-F1	K_cF1
Ctrl-F2	K_cF2
Ctrl-F3	K_cF3
Ctrl-F4	K_cF4
Ctrl-F5	K_cF5
Ctrl-F6	K_cF6
Ctrl-F7	K_cF7
Ctrl-F8	K_cF8
Ctrl-F9	K_cF9
Ctrl-F10	K_cF10
Ctrl-F11	K_cF11
Ctrl-F12	K_cF12

Y	K_Y
Z	K_Z
[	K_bracketleft
\	K_backslash
]	K_bracketright
^	K_circum
_	K_underscore
`	K_grave
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

Alt-F11	K_mF11
Alt-F12	K_mF12

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`

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: ihandle) -> (elem: ihandle) [in IupLua3]
iup.NextField(element: ihandle) -> (elem: ihandle) [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: ihandle) -> (elem: ihandle) [in IupLua3]
iup.PreviousField(element: ihandle) -> (elem: ihandle) [in IupLua5]
```

`element`: An element.

This function returns the element that received the focus.

See Also

`IupNextField`.

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: ihandle [in IupLua3]
iup.GetFocus() -> elem: ihandle [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: ihandle) -> elem: ihandle [in IupLua3]
iup.SetFocus(element: ihandle) -> elem: ihandle [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.

Resources

Resources are several auxiliary tools including menus, images, fonts and global names.

Some objects like menus and images, that are not inserted in a dialog children tree, are in fact "associated" with dialogs or controls.

Menus can be associated with dialogs only. Images can be associated with labels, buttons, toggles and menu items (this last in Windows only).

Both images and menus to be associated use a global table of names. This exist because of the LED scripts. First you associate the image or menu Ihandle to a name, then you associated the MENU or IMAGE attribute to the respective name.

For example, in C:

```
Ihandle* img = IupImage (11, 11, pixmap) ;
IupSetHandle("myImg", img);
IupSetAttribute(myButton, "IMAGE", "myImg") ;
```

or in LED:

```
myImg = image[...] (
...
)
myButton = button[IMAGE = myImg]("")
```

or in Lua:

```
myImg = iupimage {
...
}
myButton = iupbutton { title = "", image = myImg }
```

Only dialogs, timers, popup menus and images can be destroyed. Menu bars associated with dialogs are automatically destroyed.

IupItem

Creates an item of the `menu` interface element. When selected, it generates an action.

Creation

```
Ihandle* IupItem(char *title, char *action); [in C]
iupitem(title = title: string) -> elem: ihandle [in IupLua3]
iup.item(title = title: string) -> elem: ihandle [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. The size should be are 16x16.

`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.

Creation

```
Ihandle* IupMenu(Ihandle *elem1, ...); [in C]
Ihandle* IupMenuv(Ihandle **elems); [in C]
iupmenu{elem1, elem2, ...: ihandle} -> (elem: ihandle) [in IupLua3]
iup.menu{elem1, elem2, ...: ihandle} -> (elem: ihandle) [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.

Callbacks

`OPEN_CB`: Called just before a submenu is opened.

`MENUCLOSE_CB`: Called right before the submenu is closed.

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

Notes

A menu can be a menu bar of a dialog, defined by the dialog's MENU attribute, or a popup menu.

A popup menu is displayed for the user using the `IupPopup` function (usually on the mouse position) and disappears when an item is selected.

`IupDestroy` should be called only for popup menus. Menu bars associated with dialogs are automatically destroyed when the dialog is destroyed. But if you change the menu of a dialog for another menu, the previous one should be destroyed using `IupDestroy`.

In Motif the menu does not inherit attributes from the dialog, like in the Windows driver. This should be solved in future versions.

Examples

See Also

`IupDialog`, `IupPopup`, `IupItem`, `IupSeparator`, `IupSubmenu`

IupSeparator

Creates the `separator` interface element. It shows a line between two menu items.

Creation

```
Ihandle* IupSeparator(void); [in C]
iupseparator{} -> (elem: ihandle) [in IupLua3]
iup.separator{} -> (elem: ihandle) [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.

Creation

```
Ihandle* IupSubmenu(char *title, Ihandle *menu); [in C]
iupsubmenu{menu: ihandle; title = title: string} -> (elem: ihandle) [in IupLua3]
iup.submenu{menu: ihandle; title = title: string} -> (elem: ihandle) [in IupLua5]
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

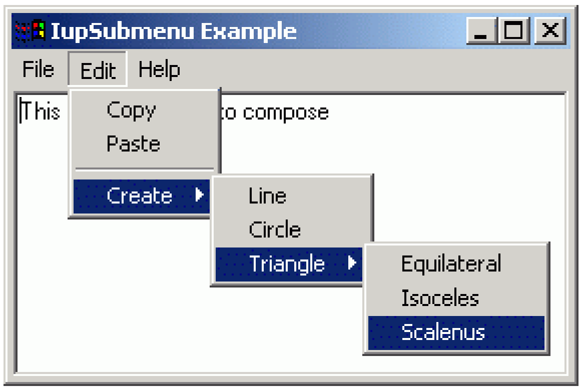
`KEY`: Associates a key to the submenu. In Windows, when used will also set an underscore on the respective letter of the submenu title.

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`.

IupImage

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

Creation

```
Ihandle* IupImage(int width, int height, char *pixels); [in C]
iupimage{pixels: table of numbers, colors: table of colors} -> (elem: ihandle) [in Iup
iup.image{pixels: table of numbers, colors: table of colors} -> (elem: ihandle) [in Iu
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: 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. Its value has the format "x:y", where x and y are integers defining the coordinates in pixels.

HEIGHT: Image height in pixels.

WIDTH: Image width in pixels.

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 discart 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 cian)
 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 (cian)
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, available at the Download. **IupView** is also capable of converting several image formats into an IupImage, and save IUP images as LED, Lua or ICO. Some of these images are already available in the pre-defined image library.

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

Examples

See Also

`IupLabel`, `IupButton`, `IupToggle`, `IupDestroy`.

IupImageLib

A library of pre-defined images for buttons and labels.

Initialization

To generate an application that uses this function, the program must be linked to the function’s library (`iupimglib.lib` on Windows and `libiupimglib.a` on Unix). The `iupcontrols.h` file must also be included in the source code.

The library is quite large because of the images. To avoid using all the images get the source code and extract only the image you need.

Reference

```
void IupImageLibOpen(void); [in C]
```

This function loads all the images in the library.

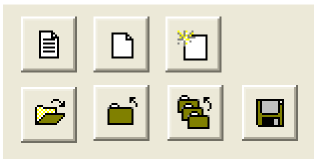
```
void IupImageLibClose(void); [in C]
```

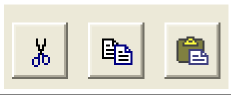
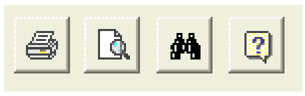
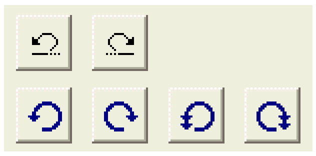
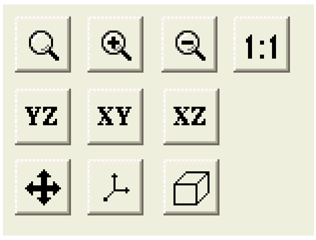
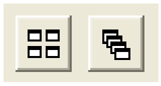
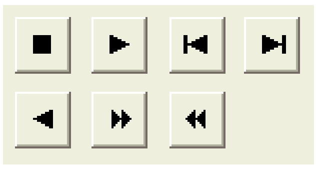
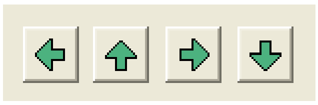
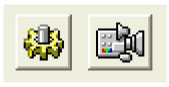
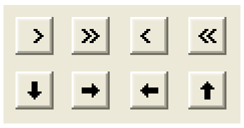
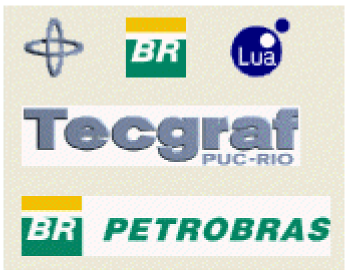
This function releases all the images of the library.

Usage

The following names are defined after the library initialization. The images do NOT include the button borders, this is just a preview for buttons!

The "BGCOLOR" color value is set and the colors are distributed so that the automatic disable color algorithm works fine. Images for buttons have size 20x20, small images are 11x11, and label images have height=30.

Names	Images
"IUP_IMGBUT_TEXT" "IUP_IMGBUT_NEW" "IUP_IMGBUT_NEWSPRITE" "IUP_IMGBUT_OPEN" "IUP_IMGBUT_CLOSE" "IUP_IMGBUT_CLOSEALL" "IUP_IMGBUT_SAVE"	
"IUP_IMGBUT_CUT" "IUP_IMGBUT_COPY"	

"IUP_IMGBUT_PASTE"	
"IUP_IMGBUT_PRINT" "IUP_IMGBUT_PREVIEW" "IUP_IMGBUT_SEARCH" "IUP_IMGBUT_HELP"	
"IUP_IMGBUT_REDO" "IUP_IMGBUT_UNDO" "IUP_IMGBUT_ONELEFT" "IUP_IMGBUT_ONERIGHT" "IUP_IMGBUT_TENLEFT" "IUP_IMGBUT_TENRIGHT"	
"IUP_IMGBUT_ZOOM" "IUP_IMGBUT_ZOOMIN" "IUP_IMGBUT_ZOOMOUT" "IUP_IMGBUT_NOZOOM" "IUP_IMGBUT_YZ" "IUP_IMGBUT_XY" "IUP_IMGBUT_XZ" "IUP_IMGBUT_FIT" "IUP_IMGBUT_AXIS" "IUP_IMGBUT_CUBE"	
"IUP_IMGBUT_TILE" "IUP_IMGBUT_CASCADE"	
"IUP_IMGBUT_STOP" "IUP_IMGBUT_PLAY" "IUP_IMGBUT_PREVIOUS" "IUP_IMGBUT_NEXT" "IUP_IMGBUT_PLAYBACKWARD" "IUP_IMGBUT_FOWARD" "IUP_IMGBUT_REWIND"	
"IUP_IMGBUT_GREENLEFT" "IUP_IMGBUT_GREENUP" "IUP_IMGBUT_GREENRIGHT" "IUP_IMGBUT_GREENDOWN"	
"IUP_IMGBUT_CONFIGURE" "IUP_IMGBUT_VIDEO"	
"IUP_IMGSML_SINGLELEFT" "IUP_IMGSML_DOUBLELEFT" "IUP_IMGSML_SINGLERIGHT" "IUP_IMGSML_DOUBLERIGHT" "IUP_IMGSML_DOWN" "IUP_IMGSML_LEFT" "IUP_IMGSML_RIGHT" "IUP_IMGSML_UP"	
"IUP_IMGLBL_TECGRAF" "IUP_IMGLBL_BR" "IUP_IMGLBL_LUA" "IUP_IMGLBL_TECGRAFPUCRIO" "IUP_IMGLBL_PETROBRAS"	

See Also

IupImage

IUP-IM Functions

Functions to load and save an IupImage from file using the IM library. The function can load or save 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, **`iupimlua_open`**, after calling **`iuplua_open`**. The `iupluaim.h` file must also be included in the source code. The program must be linked to the functions's libraries (`iupluaim.lib` on Windows and `libiupluaim.a` on Unix).

Load

```
Ihandle* IupLoadImage(const char* file_name); [in C]
IupLoadImage{file_name: string} -> (elem: ihandle) [in IupLua3]
iup.LoadImage{file_name: string} -> (elem: ihandle) [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. When failed a message box describing the error is displayed.

Save

```
int IupSaveImage(Ihandle* elem, const char* file_name, const char* format); [in C]
IupSaveImage{elem: ihandle, file_name, format: string} -> (ret: number) [in IupLua3]
iup.SaveImage{elem: ihandle, file_name, format: string} -> (ret: number) [in IupLua5]
```

`elem:` handle of the IupImage.
`file_name`: Name of the file to be loaded.
`format`: format descriptor for IM.

This function returns zero if failed. When failed a message box describing the error is displayed.

See Also

IupImage

IupSetHandle

Defines a name for an interface element.

Parameters/Return

```
Ihandle *IupSetHandle(char *name, Ihandle *element); [in C]
IupSetHandle(name: string, element: ihandle) -> handle: ihandle [in IupLua3]
iup.SetHandle(name: string, element: ihandle) -> handle: ihandle [in IupLua5]
```

`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: ihandle [in IupLua3]
iup.GetHandle(name: string) -> handle: ihandle [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: ihandle) -> (name: string) [in IupLua3]
```

```
iup.GetName(elem: ihandle) -> (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: table, n: number) -> (num: number) [in IupLua3]
iup.GetAllNames(names: 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`.

IupGetAllDialogs

Verifies the names of all defined dialogs.

Parameters/Return

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

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`

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"

IupTimer

Creates a timer which periodicaly invokes a callback when the time is up. Each timer should be destroyed using IupDestroy.

Creation

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

`RUN`: Starts and stops 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

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 usually done by CPI elements, but you can use it to create an `Ihadle*` to store private attributes.

Creation

```
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.

IupGetType

Verifies the name of the type of an interface element.

Parameters/Return

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

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.

iupMask

Functions to associate an input mask to a `IupText` or a `IupMatrix` element.

These functions are included in the Controls Library.

See the Pattern Specification for information on patterns.

Functions

```
int iupMaskSet(Ihandle *h, char *mask, int autofill, int casei) [in C or in IupLua]
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 IupLua]
int iupMaskSetFloat(Ihandle *h, int autofill, float min, float max); [in C or in IupLu
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 co
```

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 iupMaskRemove(Ihandle *h) [in C or in IupLua]
int iupMaskMatRemove(Ihandle *h, int lin, int col); [in C or in IupLua]
```

These functions are responsible for removing the mask from the control.

`h`: `Ihandle` of `IupText` or `IupMatrix`
`lin, col`: Line and column numbers in the matrix

```
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` before setting the mask.

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

iupMask - Pattern Specification

The pattern to be searched in the text can be defined by the rules given below. Note that such rules are very similar to the ones used by Lua, even though they are not the same. For further information on these patterns, please refer to the Lua Manual.

Notes

- "Function" codes (such as `/l`, `/D`, `/w`) cannot be used inside a class (`[...]`).
- If the character following a `/` does not mean a special case (such as `/w` or `/n`), it is matched with no `/` - that means that `/x` will match only `x`, and not `/x`. If you want to match `/x`, use `//x`.
- The caret (`^`) character has different meanings when used inside or outside a class - inside a class it means negative, and outside a class it is an anchor to the beginning of a line.
- The boundary function (`/b`) anchors the pattern to a word boundary - it does not match anything. A word boundary

is a point between a `/w` and a `/W` character.

- Capture operators (`f` and `g`) group patterns and are also used to keep matched sections of texts.
- A word on precedence: concatenation has precedence over the alternation (`j`) operator - that is, `faj fej fi` will match `fa` OR `fe` OR `fi`.
- The `@` character is used to determine that, instead of searching the text until the first match is made, the function should try to match the pattern only with the first character. If present, it must be the first character of the pattern.
- The `%` character is used to determine that the text should be searched to its end, independently of the number of matches found. If present, it must be the first character of the pattern. This is only useful when combined with the capture feature.

Allowed pattern characters

`c`	Matches a `c` (non-special) character
`.`	Matches any single character
`[abc]`	Matches an `a`, `b` or `c`
`[a-d]`	Matches any character between `a` and `d`, including them (just like `[abcd]`)
`[^a-dg]`	Matches any character which is neither between `a` and `d` nor a `g`
`/d`	Matches any digit (just like `[0-9]`)
`/D`	Matches any non-digit (just like `[^0-9]`)
`/l`	Matches any letter (just like `[a-zA-Z]`)
`/L`	Matches any non-letter (just like `[^a-zA-Z]`)
`/w`	Matches any alphanumeric character (just like `[0-9a-zA-Z ]`)
`/W`	Matches any non-alphanumeric character (just like `[^0-9a-zA-Z ]`)
`/s`	Matches any "blank" character (TAB, SPACE, CR)
`/S`	Matches ant non-blank character
`/n`	Matches a newline character
`/t`	Matches a tabulation character
`/nnn`	Matches an ASCII character with a `nnn` value (decimal)
`/xnn`	Matches an ASCII character with a `nn` value (hexadecimal)
`/special`	Matches the special character literally (`/[`, `//`, `/.`)
`abc`	Matches a sequence of `a`, `b` and `c` patterns in order
`aj bj c`	Matches a pattern `a`, `b` or `c`
`a*`	Matches 0 or more characters `a`
`a+`	Matches 1 or more characters `a`
`a?`	Matches 1 or no characters `a`
`(pattern)`	Considers pattern as one character for the above
`fpatterng`	Captures pattern for later reference
`/b`	Anchors to a word boundary
`/B`	Anchors to a non-boundary
`^pattern`	Anchors pattern to the beginning of a line
`pattern$`	Anchors pattern to the end of a line
`@pattern`	Returns the match found only in the beginning of the text
`%pattern`	Returns the firstmatch found, but searches all the text

Examples

`(my|his)`	Matches both `my` pattern and `his` pattern.

`/d/d:/d/d (:/d/d)?`	Matches time with seconds (01:25:32) or without seconds (02:30).
`[A-D]/l+`	Matches names such as Australia, Bolivia, Canada or Denmark, but not England, Spain or single letters such as A.
`/l/w*`	my variable = 23 * width;
`^Subject:[^/n] */n`	Subject: How to match a subject line.1
`/b[ABab]/w*`	Matches any word that begins with A or B
`from:/s*/w+`	Captures "sender" in a message from `sender`