

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.

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: GTK+, Motif and Microsoft Windows 2000/XP/2003/Vista.

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

Currently available interface elements can be categorized as follows:

- **Primitives** (effective user interaction): **dialog, label, button, text, multi-line, list, toggle, canvas, frame, image**.
- **Composition** (ways to show the elements): **hbox, vbox, zbox, fill**.
- **Grouping** (definition of a common functionality for a group of elements): **radio**.
- **Menu** (related both to menu bars and to pop-up menus): **menu, submenu, item, separator**.
- 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, and AIX) using Motif 2.x
- UNIX (FreeBSD and Linux) using GTK+ (since 3.0)
- Microsoft Windows 2000/XP/2003/Vista using Win32

Support

The official support mechanism is by e-mail, using iup@tecgraf.puc-rio.br. 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 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>.

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 tgc@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
Marian Trifon
Mark Stroetzel Glasberg
Mauricio Oliveira Carneiro
Milton Jonathan
Neil Armstrong
Renato Borges
Renato Cerqueira
Roberto Beaclair
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 Andr Carregal, being supervised and oriented by Luiz Martins, Luiz Henrique de Figueiredo, Marcelo Gattass and Carlos Jos Pereira de Lucena at Tecgraf, PUC-Rio for the Data Processing Sector (SEPROC) at CENPES/PETROBRAS.

Documentation

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

The full documentation can be downloaded from the [Download Files](#). The documentation is also available in Adobe Acrobat and Windows HTML Help 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:

- Scuri, A. "IUP - Portable User Interface". Software Developer's Journal. Dec/2005. [<http://en.sdjournal.org/products/articleInfo/2>]
- 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](#)]
- 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](#)]
- 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](#)]

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](#).

Copyright © 1994-2007 [Tecgraf](#) / [PUC-Rio](#) and [PETROBRAS S/A](#).

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

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

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

Download

The download files are in the **IUP LuaForge** site available at:

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

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/>

Tecgraf/PUC-Rio Library Download Tips

All the libraries were build using **Tecmake**. Please use it if you intend to recompile the sources. **Tecmake** can be found at <http://www.tecgraf.puc-rio.br/tecmake>.

The **IM** files can be downloaded at http://luaforge.net/project/showfiles.php?group_id=86.

The **CD** files can be downloaded at http://luaforge.net/project/showfiles.php?group_id=88.

The **IUP** files can be downloaded at http://luaforge.net/project/showfiles.php?group_id=89.

The **Lua** files can be downloaded at http://luaforge.net/project/showfiles.php?group_id=110.

Build Configuration

Libraries and executables were built using speed optimization. In UNIX the dynamic libraries were NOT built with the `-fpic` parameter. In MacOS X the dynamic libraries are in bundle format. The source code along with the "config.mak" files for **Tecmake** are also available.

The DLLs were built using the **cdecl** calling convention. This should be a problem for Visual Basic users.

In Visual C++ we use the single thread C Run Time Library for static libraries and the multi thread C RTL for DLLs. Because this is the default in Visual Studio for new projects. In Visual C++ 8 both use the multi thread C RTL.

Packaging

The package files available for download are named according to the platform where they were build.

In UNIX all strings are based in the result of the command "uname -a". The package name is a concatenation of the platform **uname**, the system **major** version number and the system **minor** version number. Some times a suffix must be added to complement the name. The default compiler is gcc, if the native compiler is used the name receive the suffix "cc". Binaries for 64-bits receive the suffix: "_64". In Linux when gcc is changed for the same uname in a new platform the major version number of the compiler is added as a suffix: "g3" for gcc 3 and "g4" for gcc 4.

In Windows the platform name is the **compiler** and its **major** version number.

All library packages contains binaries for the specified platform and includes. Packages with "_bin" suffix contains executables only.

The package name is a general reference for the platform. If you have the same platform it will work fine, but it may also work in similar platforms.

Here are some examples of packages:

iup2_4_AIX43_64_bin.tar.gz = IUP 2.4 64-bits Executables for AIX version 4.3

iup2_4_Linux26g4_lib.tar.gz = IUP 2.4 32-bits Libraries and Includes for Linux with Kernel version 2.6 built with gcc 4.

iup2_4_Win32_vc7_lib.tar.gz = IUP 2.4 32-bits Libraries and Includes for Windows to use with Visual C++ 7.

iup2_4_Docs_html.tar.gz = IUP 2.4 documentation files in HTML format (the web site files can be browsed locally).

iup2_4_Win32_bin.tar.gz = IUP 2.4 32-bits Executables for Windows.

The documentation files are in HTML format. They do not include the CHM and PDF versions. These two files are provided only as a separate download, but they all have the same documentation.

Installation

For any platform we recommend you to create a folder to contain the third party libraries you download. Then just unpack the packages you download in that folder. The packages already contains a directory structure that separates each library or toolkit. For example:

```
\mylibs\  
  iup\  
    bin\  
    html\  
    include\  
    lib\Linux26  
    lib\vc7  
    src  
  cd\  
  im\  
  lua5\  

```

This structure will also made the process of building from sources more simple, since the projects and makefiles will assume this structure .

Usage

For makefiles use:

```
1) "-I/mylibs/iup/include" to find include files  
2) "-L/mylibs/iup/lib/Linux26" to find library files
```

3) "-liup" to specify the library files

For IDEs the configuration involves the same 3 steps above, but each IDE has a different dialog. The IUP toolkit has a Guide for some IDEs:

Open Watcom - <http://www.tecgraf.puc-rio.br/iup/en/guide/owc.html>

Dev-C++ - <http://www.tecgraf.puc-rio.br/iup/en/guide/dev-cpp.html>

Borland C++ BuilderX - <http://www.tecgraf.puc-rio.br/iup/en/guide/cppbx.html>

Microsoft Visual C++ (Visual Studio 2003) - <http://www.tecgraf.puc-rio.br/iup/en/guide/msvc.html>

Microsoft Visual C++ (Visual Studio 2005) - <http://www.tecgraf.puc-rio.br/iup/en/guide/msvc8.html>

Eclipse for C++ - <http://www.tecgraf.puc-rio.br/iup/en/guide/eclipse.html>

Available Platforms

AIX43	IBM AIX 4.3 (ppc) / gcc 2.95 / Motif 2.1
AIX43cc	IBM AIX 4.3 (ppc) / cc 4.4 / Motif 2.1
AIX43_64	IBM AIX 4.3 (ppc) (64 bits libraries) / cc 4.4 / Motif 2.1
IRIX65	SGI IRIX 6.5 (mips) / gcc 3.0 / Motif 2.1
IRIX6465	SGI IRIX 6.5 (mips) / gcc 3.3 / Motif 1.2
IRIX6465cc	SGI IRIX 6.5 (mips) / cc MIPSpro 7.4 / Motif 1.2
IRIX6465_64	SGI IRIX 6.5 (mips) (64 bits libraries) / cc MIPSpro 7.4 / Motif 1.2
Linux24	Red Hat 7.3 (x86) / Kernel 2.4 / gcc 2.95 / Open Motif 2.1
Linux24g3	CentOS 3.7 (x86) / Kernel 2.4 / gcc 3.2 / Open Motif 2.2
Linux24g3_64	Red Hat E.L. WS 3 (x86_64) (64 bits libraries) / Kernel 2.4 / gcc 3.2 / Open Motif 2.2
Linux26	Fedora Core 3 (x86) / Kernel 2.6 / gcc 3.4 / Open Motif 2.2
Linux26g4	Fedora Core 6 (x86) / Kernel 2.6 / gcc 4.1 / Open Motif 2.3
Linux26g4_64	Ubuntu 6.10 (x86_64) / Kernel 2.6 / gcc 4.1 / OpenMotif 2.2
Linux26g4ppc	Ubuntu 6.10 (ppc) / Kernel 2.6 / gcc 4.1 / Open Motif 2.2
Linux26_ia64	Red Hat E.L. AS 4 (ia64) / Kernel 2.6 / gcc 3.4 / Open Motif 2.2
SunOS57	Sun Solaris 7 (sparc) / gcc 2.95 / Motif 2.1
SunOS57cc	Sun Solaris 7 (sparc) / cc 5.2 (Sun WorkShop 6 update 1) / Motif 2.1
SunOS57_64	Sun Solaris 7 (sparc) (64 bits libraries) / cc 5.2 (Sun WorkShop 6 update 1) / Motif 2.1
SunOS58	Sun Solaris 8 (sparc) / gcc 3.4 / Motif 2.1
SunOS58cc	Sun Solaris 8 (sparc) / Sun WorkShop 6 update 2 C++ 5.3 / Motif 2.1
SunOS58_64	Sun Solaris 8 (sparc) / Sun WorkShop 6 update 2 C++ 5.3 / Motif 2.1
SunOS510x86	Sun Solaris 10 (x86) / gcc 3.3 / Motif 2.1
FreeBSD54	Free BSD 5.4 (x86) / gcc 3.4 / Open Motif 2.2
HP-UXB11	HP-UX 11 (9000) / HP ANSI C++ B3910B / Motif 2.1
Darwin87	Mac OS X 10.4.7 (ppc) / Darwin Kernel Version 8.7 / gcc 4.0 / Open Motif 2.2
Win32_vc6	Microsoft Visual C++ 6 (static RTL/single thread)
Win32_vc7	Microsoft Visual C++ 7.1 (.NET 2003) (static RTL/single thread) Also compatible with Microsoft Visual C++ Toolkit 2003 - http://msdn.microsoft.com/visualc/vctoolkit2003/ ^{Â¹}
Win32_vc8	Microsoft Visual C++ 8.0 (2005) (static RTL/multithread) Also compatible with Microsoft Visual C++ 2005 Express Edition - http://msdn.microsoft.com/vstudio/express/visualc/ ^{Â¹}
Win64_vc8	Same as Win32_vc8 but for 64-bits systems using x64 standard.
Win32_dll	built using vc6, creates dependency with MSVCRT.DLL (either other libraries or new applications).
Win32_dll7	built using vc7, creates dependency with MSVCR71.DLL (either other libraries or new applications).
Win32_dll8	built using vc8, creates dependency with MSVCR80.DLL (either other libraries or new applications).
Win64_dll8	Same as Win32_dll8 but for 64-bits systems using x64 standard.
Win32_gcc3	Cygwin gcc 3.4 (Depends on Cygwin DLL 1.5) - http://www.cygwin.com/ ^{Â¹}
Win32_cygw15	Same as Win32_gcc3 , but using the Cygwin Posix system
Win32_mingw3	MingW gcc 3.4 - http://www.mingw.org/ ^{Â¹} Also compatible with Dev-C++ - http://www.bloodshed.net/devcpp.html ^{Â¹}
Win32_mingw4	MingW gcc 4.x (unofficial) - http://www.develer.com/oss/GccWinBinaries ^{Â¹}
Win32_owc1	Open Watcom 1.5 - http://www.openwatcom.org/
Win32_bc56	Borland C++ BuilderX 1.0 / Borland C++ 5.6 Compiler - http://www.borland.com/products/downloads/download_cbuilderx.html ^{Â¹,Â²} (the C++ BuilderX IDE can also be configured to use mingw3 or gcc3 versions.)
Win32_bin	Executables only for Windows NT/2000/XP
Win64_bin	Same as Win32_bin but for 64-bits systems using the x64 standard
Win32_cygw15_bin	Executables only for Windows NT/2000/XP, but using the Cygwin Posix system

^{Â¹} - Notice that all the Windows compilers with links here are free to download and use.

^{Â²} - Recently Borland removed the C++ Builder X from download. But if you bought a book that has the CD of the compiler, then it is still free to use.

History of Changes

Version 3.0 (NOT Released/Committed YET)

- IUP 3.0 is already under development but will be released only about two months after 2.6. See in [Version 3.0 History](#) what has been changed so far.
- Parts of the documentation also already reflect 3.0 changes. **So be aware that a few parts of the documentation are for IUP 3.**

Version 2.x

See [Version 2.x History](#).

Version 1.x

See [Version 1.x History](#).

History of Changes of Version 3.x

Version 3.0 (NOT Committed YET)

General

- **Changed: INCOMPATIBILITY** : **IupOpen** API now include command line arguments used by X-Windows and GTK.
- **Changed: INCOMPATIBILITY** : "IUP" cursor replaced by "HELP" cursor.
- **Changed: INCOMPATIBILITY** : WIN_SAVEBITS renamed to SAVEUNDER. And now is also available in Motif.
- **Changed: INCOMPATIBILITY** : removed old MOTIF_FONT and WIN_FONT definitions. Use only the FONT attribute.
- **Changed: INCOMPATIBILITY** : **IupGetType** renamed to **IupGetClassName**.
- **Changed:** Now attributes are stored in the element environment only if not processed by the element specific implementation.
- **Changed:** **IupGetAttribute**, **IupSetAttribute** and **IupStoreAttribute** can also be used to access global attributes using NULL as element.
- **Changed:** TIP and ZORDER attributes are now non inheritable.
- **Changed:** Removed the conflicts: K_BS=K_cH, K_TAB=K_cI and K_CR=K_cM.
- **Changed:** The global attributes TRUECOLORCANVAS, SYSTEMLANGUAGE, SHIFTKEY and CONTROLKEY are now available in all drivers.
- **Changed:** DROPPFILES_CB callback is now available for all controls. It is only activated using DRAGDROP attribute. It is active by default only for **IupCanvas** and **IupDialog**.
- **Changed:** MAP_CB is now available for all controls.
- **Changed:** attribute FONT now uses a common a more flexible definition for all drivers.
- **Changed:** child dialogs were automatically destroyed by the system but not by IUP. **IupDestroy** is now automatically called for child dialogs when the parent is destroyed.
- **Changed:** **IupAppend** and **IupDetach** can now be used for dynamic creation of menus, **IupSbox**, **IupCbox** and **IupTabs**, even after the element is mapped.
- **Changed:** **IupDetach** will now automatically unmap the element.
- **Changed:** **IupAppend** will now return the actual parent.
- **Changed:** ALIGNMENT attribute in IupLabel now includes vertical alignment values.
- **Changed:** the default FONT in Motif is now "Fixed, 10".
- **Changed:** **IupImageLibOpen** will now only register names, but will not load the images. New 32bpp images for Windows. GTK aliases are also available. Many new images.
- **Changed:** in the **IupCanvas** if ACTION is defined nothing is painted in the canvas now also in Motif. BORDER is now also supported in Motif. Now scrollbar parameters min, max, page size and line size are updated when DX/DY are updated. POSX and POSY will only update the position of the scrollbar. Automatic hide of the scrollbar now works also in Motif.
- **Changed:** LANGUAGE global attribute default from PORTUGUESE to ENGLISH.
- **New:** GTK driver, available in UNIX and Windows.
- **New:** global attributes APPSHELL, XDISPLAY, XSCREEN, XSERVERVENDOR, XVENDORRELEASE in Motif.
- **New:** MODKEYSTATE global attribute in all drivers.
- **New:** key definitions: K_acute, K_ccedilla, K_Print, K_Menu.
- **New:** key definitions for the system key modifier K_y*. In Windows this is the Windows key and in Mac this is the Apple key.
- **New:** "UPARROW" cursor in Motif. New cursors "RESIZE_NS" and "RESIZE_WE". Updated cursor documentation with pictures of all pre-defined cursors.
- **New:** dialog attributes: MINSIZE and MAXSIZE. InWindows MINSIZE is ignored for systems with multiple monitors. The Windowing system may impose a minimum default limit for the dialog that includes the title bar with all it buttons.
- **New:** common attribute CLIENTSIZE. Returns the size of containers excluding their decoration.
- **New:** ENTERWINDOW_CB, LEAVEWINDOW_CB for all controls.
- **New:** DROPPFILES_CB, RESIZE_CB callbacks for the **IupDialog**.
- **New:** IUP_CURRENT and IUP_CENTERPARENT positions for **IupShowXY** and **IupPopUp**.
- **New:** IUP_HIDE and IUP_MAXIMIZE flags for SHOW_CB callback.
- **New:** MODAL attribute for the **IupDialog**.
- **New:** function **IupGetClassType**.
- **New:** **IupGetIntInt** utility function.
- **New:** IUP_ASSERT compile flag.
- **New:** internal documentation and Guide to create new controls. Now all the controls use the same architecture using the same base class.
- **New:** internal code reorganization. More clear and simple to create controls and drivers. All comments are now in English.
- **New:** TIP additional attributes: TIPDELAY, TIPBGCOLOR, TIPFGCOLOR, TIPBALLON (Windows Only), TIPBALLONTITLE (Windows Only), TIPBALLONTITLEICON (Windows Only), TIPVISIBLE. Not available in GTK.
- **New:** **IupColorDlg**, **IupFontDlg** and **IupMessageDlg** native pre-defined dialog as elements.
- **New:** support for 24 and 32 bpp images using **IupImageRGB** and **IupImageRGBA** constructors.
- **New:** attributes PADDING, ELLIPSIS, MNEMONIC, WORDWRAP and MARKUP for **IupLabel**.
- **New:** functions **IupUnmap** and **IupReparent**.
- **New:** LINEX, LINEY, XAUTOHIDE and YAUTOHIDE attributes for the **IupCanvas** scrollbar.
- **New:** SHOWHIDDEN attribute for **IupFileDLg**. Preview canvas support for the Motif driver.

[History of Version 2.x](#)

History of Changes of Version 2.x

Version 2.6 (26/Nov/2007)

General

- **Changed:** SELECTION attribute in IupText now accept values in reverse order.
- **Changed:** IupView improvements. New functions: "Save All Images"; "Save All Images in One File". Changes: "Import Image" can load multiple images in Windows; "Save Image" allow to save in GIF format.
- **New:** SCROLLBAR attribute for IupMultiline and IupList.
- **New:** WORDWRAP attribute for IupMultiline.

Windows

- **New:** "INCLUDE" parameter for FILE_CB callback in IupFileDlg.
- **Fixed:** FONT creation when system uses a non ANSI charset.

Motif

- **Fixed:** FONT attribute internal storage.
- **Fixed:** IupMapFont interpretation of the size value to use points in X-Windows Logical Font Description format (XLFD).

IupControls

- **New:** new parameter for IupGetParam to specify a file name string that can be changed using a file selection dialog. Thanks to Flavia Anjos. New interval step for real and integer interval.
- **Fixed:** for all additional controls the used font follows strict the FONT attribute. Previously for some of the controls the CD default font were used causing an inconsistency with the control size calculation.
- **Fixed:** ACTIVE update in IupVal.
- **Changed:** in IupPPlot ACTIVE attribute renamed to CURRENT to avoid conflict with the IupCanvas ACTIVE attribute. Fixed DS_MODE and DS_EDIT return values. Fixed DS_EDIT when set to "NO" from a previously set to "YES".
- **Changed:** moved IupSbox, IupCbox and IupSpin to the core library. They do not depend on the CD library.

IupMatrix

- **Changed:** BGCOLORL:C, FGCOLORL:C and FONTL:C are now handled different for title columns and title lines. When you set the color or font of a full line/column it will not affect the title line/column except when that line/column is the title line/column (lin=0 or col=0). Individual cell colors are still handled independently.
- **New:** RASTERWIDTHn and RASTERHEIGHTn attributes.
- **Fixed:** EDITION_CB called with invalid self parameter.
- **Fixed:** DROPSELECT_CB called after dropdown list is hidden.

IupLua

- **Fixed:** missing IupCells and IupColorbar initialization in iupcontrolslua_open.
- **New:** added LuaGL binding to the IupLua console executable. So OpenGL commands can be used in Lua.

Version 2.6 RC2 (10/May/2007)

General

- **New:** function IupUpdate to force a redraw of the element and its children.
- **New:** function IupExitLoop to exit the current message loop. It is equivalent of returning IUP_CLOSE in a callback.
- **Changed:** now for the IupList when DROPDOWN=Yes the size of the dropped list will expand to include the largest text.

Version 2.6 RC1 (15/Apr/2007)

General

- **New:** functions IupGetChild, IupGetAllAttributes.
- **New:** CLIPBOARD attribute with COPY, PASTE and CUT values for IupText and IupMultiline.
- **New:** control IupPPlot that uses the PPlot library to draw 2D plots. Thanks to Marian Trifon.
- **Changed:** LEDC now supports IupCells, IupCbox, IupOleControl and IupSpin.
- **Changed:** IupMultiline and IupText size calculation. When EXPAND is different than NO or NULL, the control will ignore its contents when calculating the control size if SIZE or RASTERSIZE is not set. So now if text is larger than the multiline and EXPAND is set, the multiline will not expand to include its contents when the dialog is expanded. In this case the multiline will be expanded only what the dialog allows it to expand.
- **Changed:** size update when FONT is set. Now to update the control size IupRefresh must be called.
- **Fixed:** Added missing documentation of IupGetParent.
- **Fixed:** caps lock key codes.
- **Fixed:** Added missing IupSetAttributeHandle and IupGetAttributeHandle exports in the DLL.

Windows

- **Changed:** Resource files moved from "iup/lib" to "iup/etc".
- **Changed:** IupFileDialog attributes FILE and DIRECTORY in Windows to accept paths containing also "/".
- **Fixed:** dialog activation after IupPopup.
- **Fixed:** IUP_CLOSE return in K_ANY and K_* callbacks.
- **Fixed:** WHEEL_CB parameters x and y.
- **Fixed:** IupPopup for menus when used in the Tray if there is no visible dialogs.
- **Fixed:** FONT attribute initialization when control is not mapped yet. Affected mainly controls inside other controls.
- **Fixed:** FONT attribute parse when value is invalid.

Motif

- **New:** TOPLEVEL global attribute.
- **Changed:** default IupHelp application in Linux to "firefox".
- **Changed:** some attributes were updating the size of the control in the dialog. Now to update the control size IupRefresh must be called.
- **Fixed:** Idle processing.
- **Fixed:** return value of IupLoopStep.
- **Fixed:** invalid resize of IupList when COMBOBOX=YES and an element is added dynamically.

IupLua

- **New:** conversion to string for an Ihandle. Now returns "IUP(type): address", for example "IUP(dialog): 08C55240".
- **Changed:** IupLua5 executable in Windows to enable GDI+ in CD library.
- **Changed:** IupLua3 libraries names changed to include "3" as a suffix.
- **Fixed:** Added missing IupGLIsCurrent binding.
- **Fixed:** error message management when inside a callback in Lua 5.
- **Fixed:** error handling in iuplua_dofile and iuplua_dostoring.
- **Fixed:** the second Ihandle parameter inside the callbacks in Lua 3: DROP_CB, DROPSELECT_CB and TABCHANGE_CB.
- **Fixed:** conflict in dialog resize attribute with resize callback from canvas in Lua 3.
- **Fixed:** getattribute metamethod when value is not a number or string before calling GetHandle to check if it is a handle.
- **Fixed:** setattribute metamethod when value is stored in C now is also set to nil in Lua to avoid old invalid values in Lua.
- **Fixed:** IupAlarm optional parameters in Lua 3.
- **Fixed:** missing edit_cb callback definition for IupList in Lua 5.
- **Fixed:** Lua object memory management when destroy is called.

IupMatrix

- **New:** RELEASE_CB mouse callback.
- **Changed:** DRAW_CB callback to add the CD canvas as the last parameter. Now the canvas is also available for CDLua.
- **Fixed:** BGCOLOR and FGCOLOR for full lines or full columns in titles (L:* or *:C).
- **Fixed:** BGCOLOR for titles and empty area to use the parent's BGCOLOR instead of the dialog BGCOLOR.
- **Fixed:** BGCOLOR_CB and FGCOLOR_CB in Lua when IUP_IGNORE is returned.
- **Fixed:** setting VALUE attribute when the cell is being edited.
- **Fixed:** redraw when resizing column and the scrollbar is added to the canvas in Windows.
- **Fixed:** redraw in SunOS after editing the cell.

Other IupControls

- **Changed:** in IupTabs, when next or previous tab is selected using the arrow buttons or arrow keys, inactive tabs are skipped.
- **Changed:** CD calls to use the new CD API available only in CD version 5.0. So IUP will not be compatible with old CD versions.
- **Changed:** Because of the new parameter of DRAW_CB callback in IupMatrix, the IupControls Lua binding now depends on the CD Lua binding.
- **Fixed:** F2 key processing to rename a node in IupTree.
- **Fixed:** focus change when changing the active tab in IupTabs.
- **Fixed:** BUTTON_PRESS_CB and BUTTON_RELEASE_CB binding in Lua 3 for IupDial and IupVal.
- **Fixed:** IupTree rename box position when using scrollbars.

IupGLCanvas

- **New:** SHAREDCONTEXT attribute.
- **Fixed:** Added missing DLL export IupGLIsCurrent.

Version 2.5 (31/Mar/2006)

General

- **IMPORTANT:** New functions IupSetCallback and IupGetCallback to register callbacks without using a global name. IupGetFunction and IupSetFunction are still working, but are not used internally anymore. The new functions speed up the performance of callbacks, and reduce to zero name conflicts for callbacks in the global name table. It is recommended that the applications should replace IupSetFunction and IupGetFunction by IupSetCallback and IupGetCallback. IupLua applications are automatically benefited.
- **IMPORTANT:** Applications that overload internal callbacks of the additional controls (like IupMatrix and IupTree) must now use IupSetCallback and IupGetCallback to do the overloading. And as before these callback can not be overloaded in Lua.
- **IMPORTANT:** removed the support for callback inheritance. Now callbacks can only be set in the own element. The only exception is the K_ANY and the K_* callbacks that continues to be propagated to the parent of the element with the keyboard focus. (This was a not very usefull feature, with very few uses. But slows a lot callback management in C and in Lua. With the new IupGetCallback we were able to remove the inheritance mechanism for callbacks.)
- Changed some function declarations of the main API, some now use "const char*" in their declaration.
- Changed global attributes now are stored only if not processed by the driver.
- **IMPORTANT:** Changed the definition of Icallback to a simple one without the variable arguments. Fixed canvas callback parameters, in the documentation is float, but with the old Icallback definition the compiler used double. Now must be float.
- Changed all the internal attributes now start with the prefix "_IUP".
- Changed the default limit for text in IupText and IupMultiline to be 2³¹.
- New canvas callback FOCUS_CB.
- New helper function IupSetAttributeHandle to associate Ihandle* to attributes using automatic names. Instead of using IupSetHandle and IupSetAttribute with a new creative name, this function automatically creates a non conflict name and associates the name with the attribute. Also new function IupGetAttributeHandle.
- New "Pause" button in the IUP Image Library.
- Fixed the MULTISELECT_CB callback of the IupList so it does not need that the ACTION callback is also defined.
- Reviewed the popup dialog management. So we improve the behavior of the IupShow of other dialogs after a IupPopup, and a new possibility to safely cascade popups.

Windows

- **IMPORTANT:** Global attribute WIN_DEFAULTFONT renamed to DEFAULTFONT.
- Fixed attribute PLACEMENT=NORMAL when the dialog in minimized or maximized.
- Fixed IupPopup for menus, when the menu item callback returned IUP_CLOSE, the return value is now processed and the application is closed.
- Change WOM_CB and be set also for the dialog.
- Changed CoInitialize to CoInitializeEx[COINIT_APARTMENTTHREADED] and InitCommonControls to InitCommonControlsEx[ICC_WIN95_CLASSES] in IupOpen.

IupControls

- Changed the GETFOCUS_CB and KILLFOCUS_CB callbacks for the additional controls IupMatrix, IupVal and IupDial, now can be set without affecting their implementation.
- Changed the K_ANY for the additional controls IupTree, IupSpin and IupColorBrowser, now can be set without affecting their implementation.
- New DOUBLEBUFFER attribute for IupTabs. Default is YES. If NO will disable the double buffer. This may solve a slow Tabs redraw in UNIX when the a Tab contains many controls.
- New IupVal attributes HANDLER_IMAGE and HANDLER_IMAGE_INACTIVE that allow the use of images to replace the handler. Thanks to Rodrigo Espinha.
- Reviewed and optimized iupMask code. Added new callback MATCH_CB.

IupMatrix

- **IMPORTANT:** Callbacks ACTION and SCROLL_CB were renamed to ACTION_CB and SCROLLTOP_CB to avoid conflict with the IupCanvas callbacks also inherited by the IupMatrix.
- **IMPORTANT:** You can not automatically override the KEYPRESS_CB callback anymore. You must save the original callback and call it from inside your own.
- **IMPORTANT:** Now when in callback mode much less memory will be allocated. Also the new callbacks MARK_CB and MARKEDIT_CB can be used to control the selected cells in callback mode.
- Fixed some string buffer sizes to handle very large matrices.
- Fixed IupGetAttribute for the VALUE attribute when using callback mode and retrieving column or line title values ("0:C" or "L:0").
- Changed "matrx_img_cur_excel" to "IupMatrixCrossCursor". Old name is still available.

IupTree

- **IMPORTANT:** The IupTree implementation now uses the KEYPRESS_CB callback. The K_ANY override support was removed. The K_ANY callback can be used normally. If the application was using the KEYPRESS_CB, now it must override it manually, you must save the original callback and call it from inside your own.
- Change the appearance in Windows and Motif are now the same. Both systems look like the previous Windows implementation with a white background and some small enhancements.

IupLua

- **IMPORTANT:** IupLua3 now supports IupLua5 names. Old IupLua3 names still work, but now all the samples for IupLua5 also work in IupLua3. The documentation and the examples for the old names were removed from the manual pages. Old applications using IupLua3 can use the old names or the new names. This will make easier to old applications migrate their code to Lua 5. All Lua examples were re-tested and fixed.
- **IMPORTANT:** In IupLua3 the callbacks in C are registered only when the application register the callback in Lua, just like in IupLua5.
- **IMPORTANT:** IupColorBrowser name changed in IupLua3 from "iupcb" to "iupcolorbrowser".
- Fixed documentation of IupGetAllDialogs and IupGetAllNames. Fixed implementation to match the documentation.
- Fixed IupTimer old callback name in IupLua3.
- Fixed DROPFILES_CB canvas callback can be now used in Lua for the controls based in IupCanvas, like IupMatrix and IupTree.
- Fixed parameters of the canvas action and scroll_cb callbacks in Lua 5.
- Fixed missing FILE_CB callback in Lua.
- Changed all the additional controls now can have the K_ANY, GETFOCUS_CB and KILLFOCUS_CB callbacks without affecting their internal implementation.
- Changed Lua 5.1 "require" can now be used for all the IupLua 5.1 libraries, but the full library name must be used. For example: require"iuplua51", require"iupluacontrols51".
- Documented the IupLua 5 architecture.
- Reviewed and reorganized IupLua3 and IupLua5 code, also cleaned and simplified. In IupLua3 callbacks are now set only if they are set by the application.
- Changed IupClose can now be called from Lua in Lua 5.
- Reviewed and improved the interchange of Ihandle between C and Lua. The documentation was updated with all the possibilities.

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 DROPDOWN=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 Andr 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 SORTSIGNC 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 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 retrieve 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 Andr 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 retrieving 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.
- BGOLOR for images were ignored in the IupTree.
- Now some matrix cell attributes are not inherited from parent. Like "L:C", "ALIGNMENT*", "FGCOLOR*", "BGOLOR*", "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 FGOLOR_CB and BGOLOR_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 IupFileDialog ignored the x,y parameters of IupPopup.

Motif

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

IupControls

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

IupMatrix

- The TITLES, BGOLORs, FGOLORs 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 column at once.
- New BGCOLOR_CB and FG_COLOR_CB callbacks.
- Fixed when MARKMODE=LIN/COL/LINCOL if the first cell in the line/column is selected the click in the title area was ignored.

IupLua

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

Version 2.2.1 (25/Aug/2004)

General

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

Windows

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

Motif

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

Controls

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

IupLua

- Wrong implementation of DROPCHECK_CB.

Version 2.2 (11/Aug/2004)

INCOMPATIBILITIES

- Definition of K_parenleft changed to K_parentleft in C and all Lua bindings.
- Major IupLua5 change (see IupLua section below).
- IupLua4 is not supported.
- Motif 1.x is not supported.

General

- Documentation in Portuguese removed from the manual.
- Changed and documented the default palette used in IupImage.
- IupImage can now have up to 256 colors.
- New mouse wheel callback "WHEEL_CB" for Windows and Motif. If not defined the wheel will automatically scroll the canvas vertically.
- Changes on global attributes:
"COMPUTERNAME", "USERNAME" - now implemented also in Motif.
"COPYRIGHT" - not documented
"SCREENDEPTH", "SYSTEMVERSION" - new for Windows and Motif
"SYSTEM" - Implementation were different from the documentation
"CURSORPOS" was documented as if it was only for Windows.
"LOCKLOOP" now implemented also in Motif..
- The definitions IUP_SBDRAV and IUP_SBDRAH were not documented.
- Callback MENUSELECT_CB changed to HIGHLIGHT_CB. Now implemented also in Motif.
- New menu callback MENCLOSE_CB.
- New utility functions IupMessagef and IupGetInt2.
- Improved visual appearance of IupScarf, 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 IupFileDialog when browsing for folder it will use a new interface, with a resizable dialog and other features.
Also in IupFileDialog fixed start position for IupPopup. New file selection callback and preview area. IupFileDialog was not using the IUP_PARENTDIALOG attribute. Default value for IUP_NOOVERWRITEPROMPT was wrong. ALLOW_NEW was inconsistent with the documentation.
- The button callback now is called only when the button is released inside the button area.
- WOM callback renamed to WOM_CB.
- New "HELPPBUTTON" attribute for the dialog.
- The menu item now accepts auxiliary bitmaps.
- When the dialog has a multiline and the user press ESC the window was improperly closed.
- Fixed combobox resize feedback. When resizing the dialog the combobox was temporarily opened.
- IupCanvas was not receiving arrow keys events correctly in keypress_cb.
- IupHide now can close popup dialogs.
- Attribute TABSIZE for IupMultiline in Windows was not documented.
- Default value for attribute BGCOLOR for IupCanvas in Windows was not documented.
- Direction keys now are processed by the ACTION callback for IupText.
- The GETFOCUS_CB and KILLFOCUS_CB management for the controls was reviewed and optimized.
GETFOCUS_CB now works for toggle and button.
- First RESIZE_CB of the canvas received a wrong canvas size.
- Label alignment for images was always center.

Motif

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

Controls

- DEFAULTESC and DEFAULTENTER were missing in IupGetColor.
- New function IupLoadImage that uses the library IM to load an image file (implemented in an additional library).
- New dialog IupGetParam, similar to IupScanf but uses variable controls for fields.
- IupTabs now uses the 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 suffix "_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.colorbarbrowser.
- 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 additional border included in "RASTERIZE").

Windows

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

Motif

- Dropdown were becoming unstable when VALUE attribute is set after IupMap.
- Dropdown were not being positioned accordingly.
- IupList was not selecting the first item.
- IupTimer callback were called only once.
- The value "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 IupFileDialog.

Windows

- New attribute "CURSORPOS", allows programmatically changing the cursor's position on the screen.
- New attribute "NOOVERWRITEPROMPT" for IupFileDialog. It prevents IupFileDialog 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 IupFileDialog. 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](#)

History of Changes of Version 1.x

Version 1.9.1 (17/Oct/2002)

General

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

Windows

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

Motif

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

IupLua

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

IupTree

- Expand and collapse no more alters selection of elements.
- When all nodes were deleted using "DELNODE0", "CHILDREN" inside a tree_selection callback, the program crashed.
- BRANCH_OPEN now passes parameter node.
- IUP_DEPTH now works for folders and leaves. Attention: the depth works only with the appointed element, not with its children.
- Some conditions necessary for a DEPTH change were wrong.
- Redraw optimization.
- When a tree was big, the scrollbar was not working properly.
- When the tree was totally expanded and the scrollbar was all down, collapsing folders made the thumb be wrongly calculated.
- PGDN and PGUP were stopping in any folder that was closed.
- Even when the user did not want a folder or leaf to be selected, sometimes the tree allowed it.
- When the tree's folder does not have children, an empty box is shown next to it (instead of the + and - symbol.)
- Sometimes an error occurred in selection when a double click was done in a tree.
- Callback RENAMENODE_CB now works correctly.
- When the TreeSetValue function was used to define a tree, using a folder with no leaves made the program crash.
- New attribute "COLORid" allows the text color to be changed.

IupTabs

- **IUP_REPAINT was not repainting the elements in its interior.**

IupMatrix

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

IupVal

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

Version 1.9.0 (18 Dec 2001)

General

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

Windows

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

Motif

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

iupMask

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

IupMatrx

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

IupTree

- Attribute IUP_MARKED now also sets.
- IupTree's binding now exports functions to set and get ID.
- Redraw is now done with one attribute. This avoids unnecessary redraw when the user wants to insert a lot of data.
- IupTree now takes leafs and nodes before IupMap.
- Clicking to select a LEAF was not always working in Windows.
- BRANCHOPEN and BRANCHCLOSE callbacks were not testing the return value correctly.
- Double clicking was not working properly. When the user clicked a node, while the timer was still waiting for the second click, it was impossible to click a nother node.
- Hitting the space button with CTRL pressed now marks the element immediately.
- SELECTION_CB callback was created. This callback is called when any type of mark is made on the Tree. The return value blocks this action.
- Removed callbacks BEFORESELECTION_CB and AFTERSELECTION_CB.
- Setting IUP_VALUE though programming does not activate callbacks anymore.
- Keyboard control, including arrow keys, PGUP, PGDOWN, HOME e END were not working properly.
- Clicking + or - was not activating the SELECTION_CB callback.
- SELECTION_CB is now in the binding. BEFORESELECTION_CB and AFTERSELECTION_CB are not.
- The IUP_MARKEDid attribute now returns IUP_YES or IUP_NO depending on the state of the node's mark. If the node does not exist, the returned value is NULL.
- IupTree was breaking when it tried to erase a marked node inside BRANCHCLOSE_CB.
- The BRANCHCLOSE_CB callback was not being called for the correct node.
- SELECTION_CB was included in the binding.
- Including a new leaf now does not alter selection.

IupGL

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

IupCB

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

IupLua

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

Version 1.8.9 (07 May 2001)

IupMatrx Control

- If the user defined FG_COLOR while the matrix was in edition mode, the application crashed.

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

IupTree Control

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

IupCB Control

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

Windows

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

Version 1.8.8 (15 Mar 2001)

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

CPI

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

Motif

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

IupTabs Control

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

IupMatrx Control

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

Windows

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

IupLua.exe

- Now works properly with all controls.

IUP Manual

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

Version 1.8.7 (23 Nov 2000)

- The alignment of composition elements can now be changed on-the-fly.
- Current language treatment has been changed. ATTENTION: previous putenv no longer works! Use new functions IupSetLanguage and IupGetLanguage. Default language: Portuguese.
- IupAlarm's design was reformulated. Now all buttons have the same size.
- Functions IupUnMapFont and IupMapFont were created to make the use of the drivers fonts easier.

- Attribute IUP_FONT now accepts a string either with the native font or the IUP font, and always returns the native font (attributes WINFONT and MOTFONT are now obsolete).

Motif

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

Windows

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

IupMatrix Control

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

Other Extended Controls

- The element from IupGL was not getting the focus when it was the only element in the dialog.
- In IupGL, OpenGL now synchronizes its functioning with Motif (glXWaitX) at resize.
- IupGC now works with IUP_ENGLISHs variable set (cancel/cancela, red/Verm, etc.)
- IupGauge now accepts changing text or percentage values on-the-fly.
- Tabs font now has a differentiated color when it is inactive.

IupLua

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

IupLua Program

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

Version 1.8.6 (21 Jun 2000)

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

User Manual

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

Windows

- There is no longer any restriction for the number of dialogs created using IUP (the only limitation now is Windows capacity to create native elements).
- Events of IupButton and IupToggle were being improperly called when a IupHide or a IupShow was made on the dialog.

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

Motif

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

IupLua

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

Extended Controls

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

Version 1.8.5 (18 Apr 2000)

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

Windows

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

Motif

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

IupMatrix Element

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

Version 1.8.4 (09 Dec 1999)

Windows

- A problem, which called the dropdown callback even for an already-deleted element, was fixed.
- Function `IupHelp` is now available.
- A bug was fixed; it caused excessive system resource usage when dialogs with several elements were used.
- The size of the version dialog was corrected.
- A bug was fixed; it made IUP crash depending on the use of `MessageBox`. Same for `IupFileDialog`.
- Callback `IUP_BUTTON_CB` was added for the `IupButton` element.
- A bug was fixed; it made `IupGetInt(d,IUP_X)` return a wrong value when the dialog was maximized.

CPI Controls

- The color inheritance problem was fixed.
- Corrections were made to the Dial size.
- Attributes of colors FG_COLOR, BG_COLOR, and fonts FONT, WIN_FONT, MOTIF_FONT.

Version 1.8.3 (15 Jun 1999)

Windows

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

Motif

- Color management for 8bpp displays (256 colors) was re-implemented. Basic colors used by IUP (black, white and the grays used for highlight and shadow) are now reserved, and the search for colors in the palette was optimized.
- Elements such as IupCanvas now have their own visual, independent from their parents. If allowed by the display, the default visual of a canvas will be TrueColor (24bpp); if not, it will be the same as the default display visual.
- The IupToggle element now processes the IMAGE attribute differently: it now shows the toggle with the same appearance as the IupButton element, but maintaining its functionality the button remains pressed until the user clicks it again. The IMPRESS attribute can be used to define the image used for the pressed button. In this case, the user is in charge of giving it a 3D appearance.
- IMPORTANT: The size of the dialog can be adjusted after being mapped, by means of the SIZE and RASTERSIZE attributes
 - The size of the dialog has now precedence over the smallest size required by its children (either having been specified in its creation or in run-time).
 - Attributing a NULL value to the SIZE or RASTERSIZE (in C) of a dialog will re-compute its size according to the size of its children.
 - Partial dimensions (##x and x##) are now treated correctly.
 - Therefore, applications that define sizes for dialogs (either in LED or in C) smaller than the minimum size required by their children will show truncated dialogs. To force a computation based on the size of the children, set any of these attributes to NULL (in C) or simply do not define them in LED. As a general rule, avoid specifying a dialog size unless there is a real need for such in this case, be careful to specify a sufficient size.
- IupFileDialog:
 - The default value for the DIALOGTYPE attribute was not being recognized (the program aborted when there was no defined value).
 - When ALLOWNEW = NO, the dialog informs if the user is specifying a non-existing file (instead of simply returning, as was happening).
 - When the dialog type was OPEN, the returned value was 1 (Cancel) even when the user confirmed the operation.
 - If DIALOGTYPE is SAVE, a confirmation is required if the file already exists.
 - A new dialog was created for each popup without destroying the previous dialog.
 - The NOCHANGEDIR attribute was implemented.
 - The dialog does not return if the user specifies a new file when attribute ALLOWNEW = NO. The same happens when attribute ALLOWDIR = NO and a directory is specified. In these cases, alerts are shown.
- The IupGetColor function for CPI controls was replaced in functionality by the IupGetRGB function (IupGetColor is maintained for compatibility purposes, but it should no longer be used).
- TRUECOLORCANVAS was created. It indicates if the display allows the creation of TrueColor windows (> 8bpp), even if the default is PseudoColor.
- Tabs: a problem was fixed concerning the use of the VISIBLE attribute for elements belonging to a non-selected tab.
- IupHelp: allows using a browser (default = Netscape) for viewing HTML pages.
- The ACTION_CB callback, from IupText, now receives, apart from Ihandle* and int, a char* pointing to the new text value in case the key is confirmed.
- Dropdown lists were not correctly processing the VISIBLE attribute.
- A problem with the initialization of multiple-selection lists was solved: the VALUE attribute was not being respected in some cases.
- Attributes FG_COLOR and BG_COLOR from the dropdown list were not being correctly updated.
- IupLoopStep was re-implemented: now it no longer blocks when there are no events to be processed (it simply returns DEFAULT).
- The dropdown list is closed when the associated textbox is totally or partially darkened.
- The dropdown list was not being closed when the dialog lost the focus if IupIdle was registered.
- A problem in the exhibition of CPI controls was fixed.
- New return code (CONTINUE) was created, specific for key callbacks, to be used when the event is to be propagated to the parent of the element receiving it.
- In some situations, elements destroyed by means of IupDestroy were receiving events, making the application abort.
- The redefinition of items in the main menu was making the dialog return to its original size.
- Consulting attribute BG_COLOR in a dropdown list was aborting the application.
- Consulting attributes BG_COLOR and FG_COLOR of a canvas with a different visual from the default was generating an X-Windows error message.
- The problem with IupFileDialog was fixed (the application was aborting).
- IupDestroy in a bar menu was inducing an infinite loop to the application.
- The list now matches the documentation: it calls the action callback for the de-selected element (with the v = 0 parameter).
- Bug correction: The use of a Motif attribute instead of a function was making Motif lost control of memory management (memory already liberated was liberated again, which aborted the application).
- ACTION in IupText caused SIGSEV when the user pressed ENTER.
- New IupMapFont for mapping IUP fonts -> Motif.

Version 1.8.2

Windows (12 Jan 99)

- Function char* IupMapFont(char* font) converts a IUP font describer (used by the IUP_FONT attribute) into a native font describer (used by IUP_WIN_FONT).
- File Drag & Drop was implemented in dialogs and canvases, via the IUP_DROPFILES_CB callback.
- Attribute IUP_EXTFILTER was implemented for the IupFileDialog control, allowing the use of more than one filter.
- Changes were made to allow the creation of CPI elements other than CANVASes or dialogs.
- The IUP_ACTIVE attribute of a dialog can now be changed after it was mapped.
- List callback correction: the callback is now called both for selected and not selected items.
- New function void IupHelp(char *url) shows a URL in a Netscape window.
- The treatment of the new return value for keyboard callbacks, IUP_CONTINUE, was implemented.

- IUP_CURSOR attribute was implemented.
- A code was added to treat the case of toggle de-selection via IupSetAttribute.
- IUP_CARET now uses , as a separator instead of old :.
- A restriction was eliminated that prevented the function iupGetTextSize from being called passing a dialog or frame as a parameter.
- New text callback was implemented; it receives the text both before and after the change, and receives the code of the typed key.
- It was possible to set two activated radio toggles by selecting VALUE for one of them on the radio and VALUE = ON on the other toggle.
- Attributes IUP_STARTFOCUS, IUP_DEFAULTENTER and IUP_DEFAULTESC were implemented.
- The IUP_VALUE of a IupRadio was not allowing to be changed if it was not visible.
- A problem was corrected for the lists, which were being reset between a IupShow/IupPopup and another.
- Attribute IUP_SELECTEDTEXT was implemented. It returns the selected text (if there is any), with the \r already filtered.
- A bug was corrected; it caused an Assertion Failed when the mouse was moved after a window was destroyed.
- The value of IUP_VALUE of a IupText and a IupMultiline now does not contain \r.

Motif v1.8.2 (14 Aug 98)

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

Version 1.8.1

Windows v1.8.1 (17 Jul 98)

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

Motif v1.8.1 (16 Jun 98)

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

Version 1.8 (29 May 98)

General (also includes changes to both drivers)

- BUG: Valuator, Dial and Gauge could cause an invalid memory access on resize or destroy.
- BUG: The parse of CPI elements described in LED was corrected.
- BUG: Valuator was removing the applications idle action.
- NEW: FILEDLG control.
- NEW: IupStoreAttribute function.
- NEW: IupSetAttribute function.
- NEW: IupSetGlobal, IupGetGlobal and IupStoreGlobal functions for global attributes.
- NEW: K_sCR key; shift-enter combination is now treated by IUP (callback: IUP_K_sCR, code: K_sCR).
- NEW: IUP_TYPENAME attribute returns the name of the element type.
- NEW: CPI popup method.
- NEW: Definition of global attributes (verification only) IUP_VERSION, IUP_DRIVER, IUP_SYSTEM and IUP_SCREENSIZE.
- NEW: Attributes IUP_X and IUP_Y were implemented, for dialogs only. They provide the dialogs upper left corner coordinates in relation to the upper left corner of the screen.
- NEW: IUP_SHRINK attribute to change the computation of the position and size of elements.
- NEW: CPI control for an OpenGL canvas.
- CHANGE: The IUP_TYPE attribute of the IupFileDialog control was changed into IUP_DIALOGTYPE, which must contain OPEN, SAVE or NULL.
- CHANGE: The IupSetAttributes function now returns the Ihandle*.
- CHANGE: The IupSetAttribute function no longer returns the old value.
- CHANGE: CPIs create method now creates the handle.
- CHANGE: New function for CPI class creation.
- CHANGE: Some obsolete definitions of iup.h are now only available when the IUP_COMPAT macro is set.
- CHANGE: The ICTL_TYPE attribute of the IupTabs control was changed to ICTL_TABTYPE.

Lua Binding

- NEW: iupkey_open function allows using IUPs key definitions in Lua.

Windows

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

Motif

- BUG: The dropdown list no longer remains on the screen.
- BUG: The computation of scrollbar attributes POSX and POSY was fixed.
- BUG: Double-click was only being generated for the first button.
- BUG: FRAME layout was corrected.
- BUG: The color of the menu item was corrected.
- BUG: The management of the nested elements of a ZBOX and/or with the VISIBLE attribute defined for its children was fixed.
- BUG: The color remained undefined when the value of attribute FGCOLOR or BGCOLOR was not valid.
- BUG: General cleaning was made to remove memory leaks from the driver.

- NEW: Attributes IUP_X and IUP_Y to provide the pixel position of any element.
- NEW: Attribute IUP_RASTERSIZE can be consulted.
- NEW: Menu item now accepts \t to align the text to the right Windows already allowed it.
- NEW: Version number was added; can be retrieved with tecver.
- CHANGE: Multilines scrollbar is no longer deactivated with ACTIVE=NO.
- CHANGE: Multilines and lists BG_COLOR no longer affects the scrollbars.

Version 1.7

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

To Do

General

- A MacOS X native driver using Carbon.
- A wxWidgets driver? A Qt driver?
- A tutorial section in the documentation. Add more complete samples.
- 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.
- Drag&Drop between controls or dialogs in the same application.
- Buttons with image and text simultaneously.
- IupLua executables in SunOS using the Sun WorkShop 6 C++ and in AIX using the C++ for AIX compilers do not include the IMLua binaries.
- BUG: In Windows when using a callback to show a dialog during a key press in Multiline, if the key contains a character even with a Ctrl modifier, the key is inserted in the Multiline after the dialog is hidden.

Motif

- Support for MDI.
- Callback SHOW_CB is not called when the dialog is hidden because of PARENTDIALOG.
- 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

- Fix column resize when line is moved outside the matrix canvas.
- 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.
- Should allows to start the selection on cell 0:0.

IupTree

- Define minimum size based on tree nodes. New callback mode. Change internal list to real tree?
- Images with variable sizes for nodes.

New Controls

- A detachable toolbar?
- Image Listbox
- Grid Container (to distribute elements in a grid)
- RTF editor in Windows (Using Scintilla?)
- 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
FOX	LGPL*	1997-2006/08	1.7.4	C++	Win, X	own	3	great look, license restrictions
FLTK	LGPL*	1998-2006/01	1.1.7	C++	Win, X, Mac	own	4	was from Digital Domain. Easy to learn.
GTK+	LGPL*	1997-2006/07	2.10.1	C	Win, X, Mac	own	9	target for X-Windows, basis of GNOME, Windows is apart, Mac using X
Qt	GPL	1994-2006/06	4.1.4	C++	Win, X, Mac	own	(many)	Is free for Non Commercial, basis of KDE, Emulates the native look and feel
wxWidgets	LGPL*	1992-2006/08	2.7.0	C++	Win, X, Mac	native	6	X can use Motif or GTK
IUP	MIT*	1994-2006/09	2.5.1	C	Win, X, Mac	native	2	X can use Motif or GTK, Mac using X

Table Last Update: August 2006

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 fully "portable" toolkit since it is targeted 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, specially in Windows. The Windows port works ok but it is not Windows behavior neither look and feel.

Qt has several license limitations, although it 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 platforms that include several secondary libraries not related to User Interface. In IUP we focus only in Graphical User Interface.

Developing IUP

IUP has a C API, only has functions for Graphical User Interface, and uses "Native Controls" in Windows, Motif and GTK+. These are the major differences between IUP and other toolkits. Because of that IUP is small, fast and very 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.

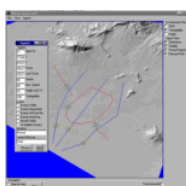
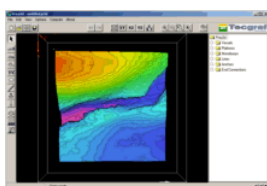
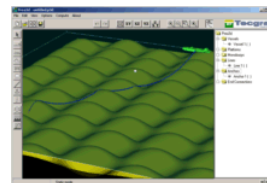
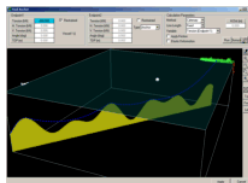
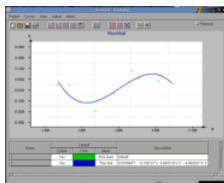
Why Not Mac? The first Mac driver was developed for MacOS 9 and had several memory limitations so it was abandoned. With Mac OS X we have the opportunity to do something better. Today IUP runs on Mac OS X using X11 and Motif or GTK. We plan for the future to build a native Carbon driver.

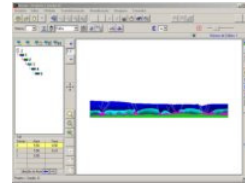
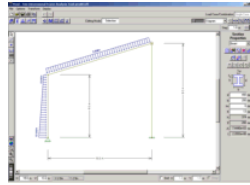
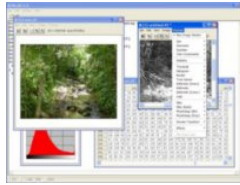
Why Motif? Motif is still very important for non Linux systems, some Tecgraf applications run on AIX, SGI and Sun systems, that only have Motif installed and we can not force the installation of other toolkits like GTK.

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

Screenshots

Click on the picture to enlarge image.





Guide

Getting Started

IUP has four 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 associate a C function with a name using **IupSetFunction**, and then associate the callback attribute with that name using **IupSetAttribute**. But applications now should use the **IupSetCallback** function to directly associate a callback for a control.

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 **lua.lib/liblua.a/liblua.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 some icons and cursors can be used when not using the DLLs. **iup.rc** is located in "etc" folder of the distribution.

There is also guides for using some IDEs: [C++ Builder X](#), [Dev-C++](#), [OpenWatcom C++](#), [Visual C++ 7 \(Visual Studio 2003\)](#), [Visual C++ 8 \(Visual Studio 2005\)](#) and [Eclipse for C++](#).

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.

GTK+ (since 3.0)

In UNIX it will need the X-Windows libraries just like the Motif driver. And in UNIX or in Windows it will need the "iupgtk" library and the following GTK+ libraries:

```
gtk-win32-2.0 gdk-win32-2.0 gdk_pixbuf-2.0 pango-1.0 pangowin32-1.0 gobject-2.0 gmodule-2.0 glib-2.0
or
gtk-x11-2.0 gdk-x11-2.0 gdk_pixbuf-2.0 pango-1.0 pangox-1.0 gobject-2.0 gmodule-2.0 glib-2.0
```

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 in Motif. This is because it depends on Motif and X11, you will have to load these libraries first. So it is easier to build a base application that already includes X11, Motif and the main IUP library than trying to load them all. In Windows this is not a problem. The IUP secondary libraries can be easily dynamic loaded regardless of the system.

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 (already installed in the system) and on the Motif 2.x+X11-R6. In Linux you should use Open Motif 2.x. In Windows the defines `_WIN32_WINNT=0x0500` and `_WIN32_IE=0x0500` must be defined. IUP_ASSERT can be defined to enable some runtime checks for the main API.

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 a 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);
        IupSetCallback(hDlg, "RESIZE_CB", (Icallback)ResizeCB);
        IupSetCallback(hDlg, "SHOW_CB", (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

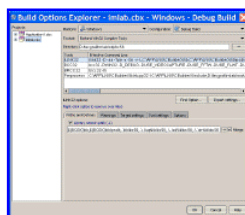
Borland C++ Builder X is an Integrated Development Environment (IDE) for Java and C/C++ languages. It can use several sets of compilers, including the Borland command line compilers version 5.6.

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 computer, 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:

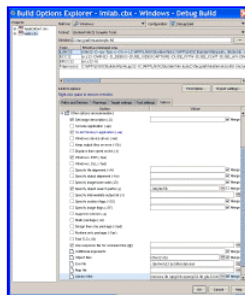
- In the Tools list, click on ILINK32. Then bellow select the Path and Defines tab - there you are going to add the path of the libraries you use, for example:

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



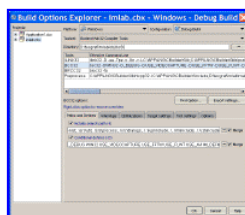
- In the same ILINK32 options, in the tab Options, select Other Options and Parameters, then Library files - there you are going to list the libraries, for example:

```
cw32.lib import32.lib vfw32.lib comctl32.lib iup.lib iupcontrols.lib cd.lib cdiup.lib im.lib im_capture.lib im_avi.lib im_process.lib i
```



- 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

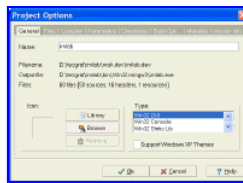
<http://www.bloodshed.net/devcpp.html>

"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 "mingw32" binaries in the download page.

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

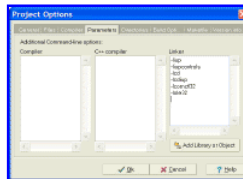
- General / Type - you can configure Win32 GUI or Win32 Console, but if you set to console it will always create a console screen behind your window when the program starts. Do not select "Support Windows XP Themes".



- 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.

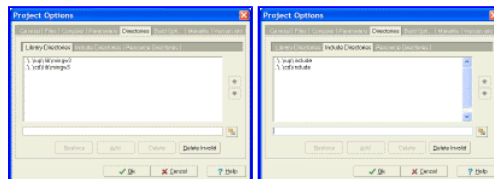


- Directories / Library Directories and Include Directories - where you are going to list the include path, for example:

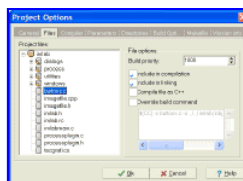
```
..\..\iup\lib\mingw3
..\..\cd\lib\mingw3
or
c:\tecgraf\iup\lib\mingw3
c:\tecgraf\cd\lib\mingw3
```

And:

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



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



OpenWatcom C++ IDE Project Options Guide

<http://www.openwatcom.org/>

Open Watcom is an Integrated Development Environment (IDE) for Fortran and C/C++ languages using the Watcom compilers.

"It is a joint effort between SciTech Software Inc, Sybase and the Open Source development community to maintain and enhance the Watcom C/C++ and Fortran cross compilers and tools. An Open Source license from Sybase allows free commercial and non-commercial use of the Open Watcom tools."

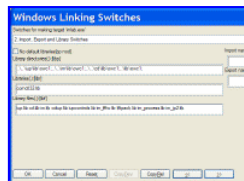
To use IUP with Open Watcom you will need to download the "owc1" binaries in the download page.

After unpacking the file in your computer, you must create a new Project for a "Windowed Executable" and configure your Project Options. In the Project Options there are 2 important places:

- In the Windows Linking Switches dialog, select option 2. Import, Export and Library Switches. Then enter the Library directories and Library files. For example:

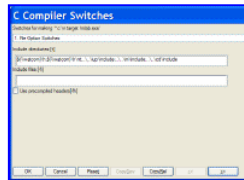
```
..\lib\owc1;..\..\iup\lib\owc1;..\..\cd\lib\owc1;..\..\im\lib\owc1
```

```
comctl32.lib iup.lib iupcontrols.lib cd.lib cdiup.lib im.lib im_process.lib iupgl.lib opengl32.lib glu32.lib
```



- In the C Compiler Switches dialog, select 1. File Option Switches. Then enter the include path, for example:

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

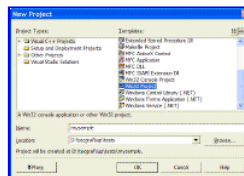


Visual C++ 7 IDE Project Properties Guide

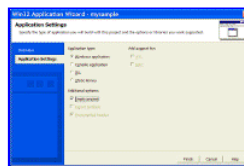
<http://msdn2.microsoft.com/en-us/vstudio/aa700867.aspx>

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

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



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

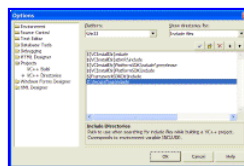


You can also create a "Console application", and whenever you execute your application a text console will also be displayed. But this is a very useful situation so you can use the standard C printf function to display textual information for debugging purposes.

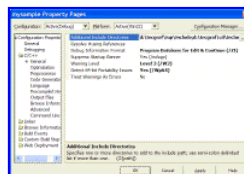
Then add your files in the menu "Project / Add New Item" or "Project / Add Existing Item".

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

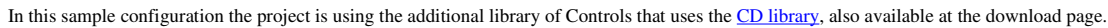
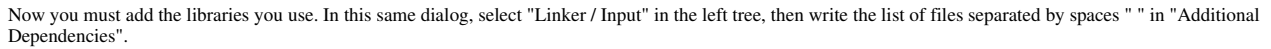
One is in the menu "Tools / Options", then select "Project / Visual C++ Directories". Select "Include Files" or "Library Files" in "Show directories for:". In this dialog you will configure parameters that will affect all the projects you open.



Or you can configure the parameters only for the project you created. In this case go 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".



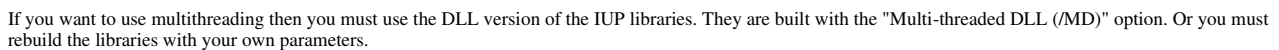
To configure the library files location select "Linker / General" in the left tree, then write the list of folders separated by ";" in "Additional Library Directories".



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 /NODEFAULTLIB:library
```

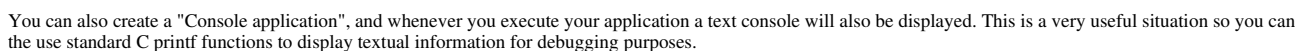
The default configuration use the C run time library with debug information, and IUP uses the C run time library without debug information. You can simply ignore this warning or change your project properties in "C/C++ / Code Generation" in the left tree, then change "Run Time Library" to "Single Threaded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



<http://msdn2.microsoft.com/en-us/vstudio/default.aspx>
<http://msdn.microsoft.com/vstudio/express/downloads/> (free version)

This guide was built using Microsoft Visual Studio 2005, which includes Visual C++ 8. Also works for Visual Studio Express Edition.

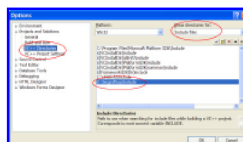
To create a new project go to the menu "File / New / Project":



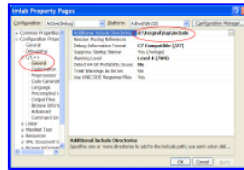
Then add your files in the menu "Project / Add New Item" or "Project / Add Existing Item".

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.

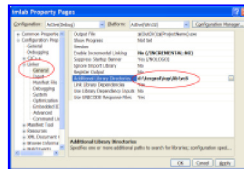
One is in the menu "Tools / Options", then select "Project and Solutions / Visual C++ Directories". Select "Include Files" or "Library Files" in "Show directories for:". In this dialog you will configure parameters that will affect all the projects you open.



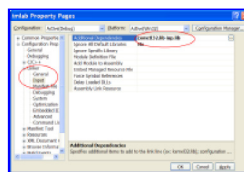
Or you can configure the parameters only for the project you created. In this case go the menu "Project / Properties". To configure the include files location select "Configuration Properties / C/C++ / General" in the left tree, then write the list of folders separated by ";" in "Additional Include Directories".



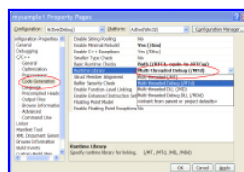
To configure the library files location select "Configuration Properties / Linker / General" in the left tree, then write the list of folders separated by ";" in "Additional Library Directories".



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



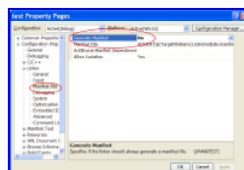
The default configuration use the C run time library with debug information and in a DLL. The standard IUP binary distribution has two packages for Visual Studio 2005 (or Visual C++ 8). Both do not have debug information, but this could be ignored even if a warning appears in the Output log. To change your project properties go to "Configuration Properties / C/C++ / Code Generation" in the left tree, then change "Run Time Library" to match the IUP binary package you are using.



The "vc8" package includes static libraries without debug information. So to match this package configuration you should select "Multi-threaded (/MT)".

The "dll8" package includes dynamic libraries without debug information. So to match this package configuration you should select "Multi-threaded DLL (/MD)".

When using the "iup.manifest" from "iup.rc", configure the linker properties of your project to do NOT generate a manifest file or the Windows XP Visual Styles won't work.

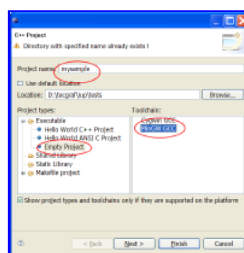


Eclipse for C++ Project Properties Guide

<http://www.eclipse.org/>

This guide was built using Eclipse 3.3 IDE for C/C++ Developers in Windows (but similar configuration can be applied for Linux).

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

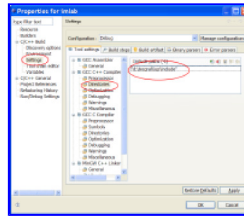


You can use the MingW3 or Cygwin compiler. Just use the respective IUP binaries package: "mingw3" or "gcc3".

Then add your files to the projet folder if they are not already there.

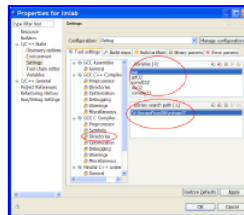
After creating the project you must configure it to find the IUP includes and libraries.

Go the menu "Project / Properties", then to configure the include files location select "GCC C Compiler / Directories" in the left tree, then add the list of folders in "Include Paths".



Be aware that you will have to repeat the configuration for the C++ compiler.

To configure the library files location select "MinGW C++ Linker / Libraries" in the left tree, then add the list of folders in "Library Search Path" and add the add the list of folders in "Libraries".



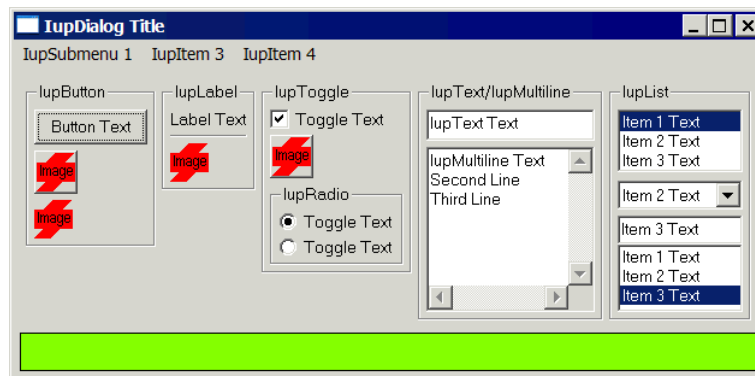
Complete Samples

Standard Controls

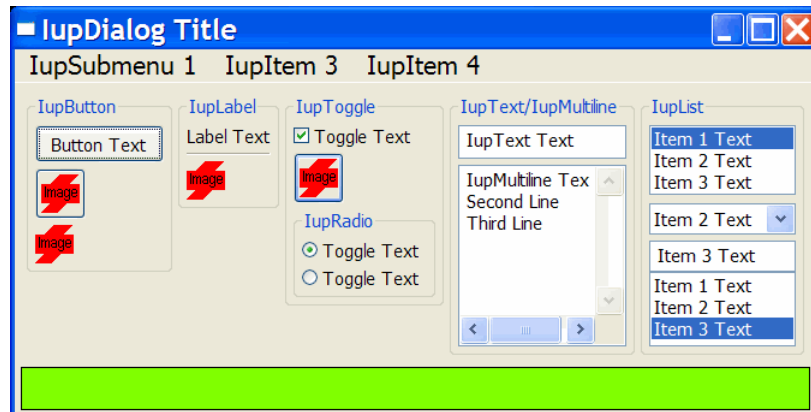
The following example creates a dialog with virtually all of IUP's elements as well as some variations of them, with some attributes changed. The same example is implemented in C, LED and Lua. Both screens presented are from the same example, one in Windows 95 and the other in IRIX. The C code is ready 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 IupLua
[sample.c](#) [sample.led](#) [sample.lua](#)

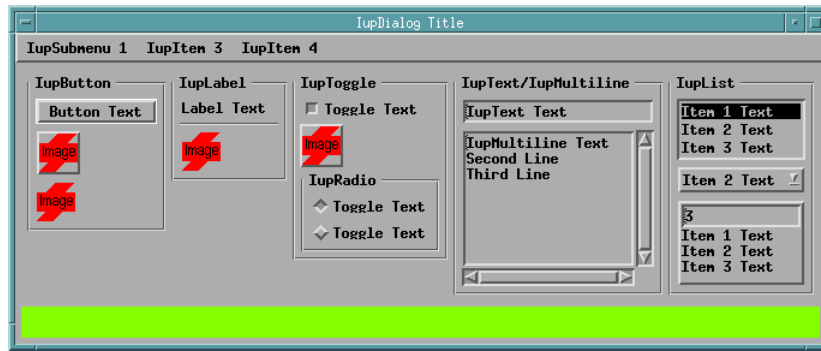
The Result in Windows 2000



The Result in Windows XP

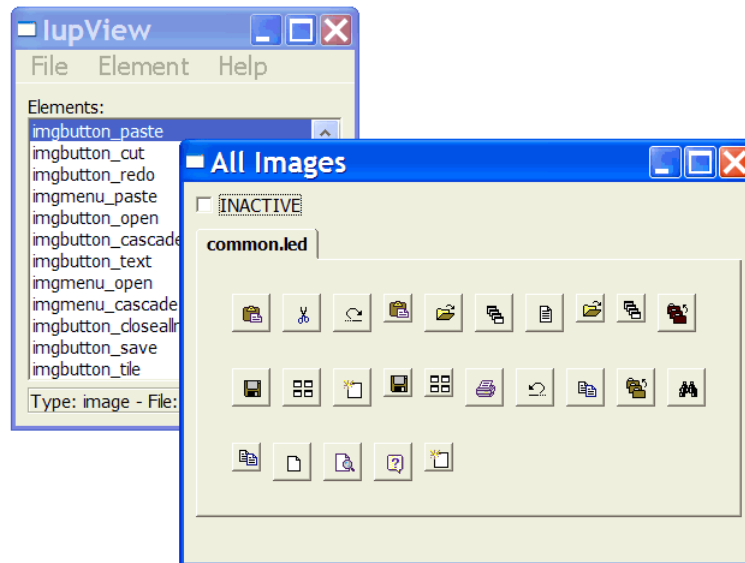


The Result in Motif



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.

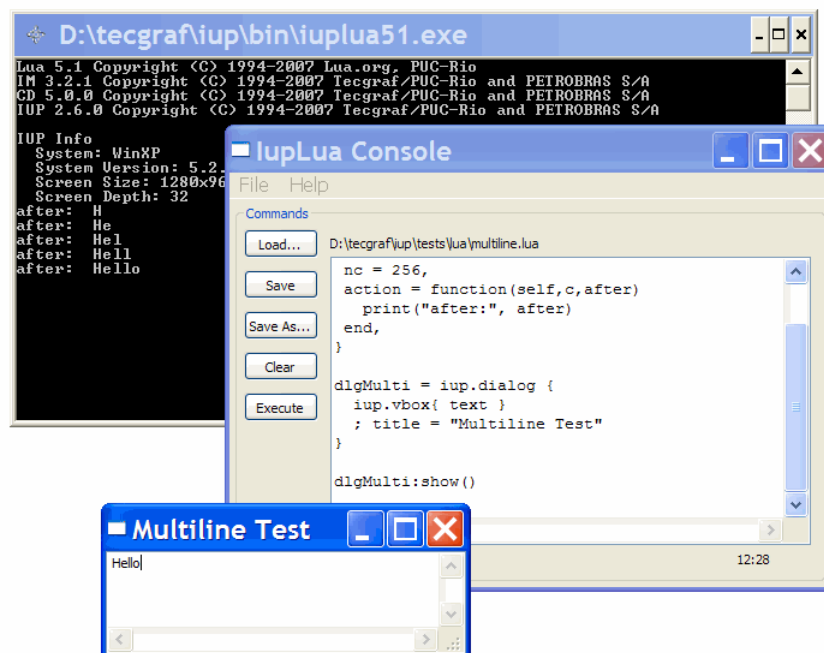


The **IupLua** console can load and execute Lua scripts using the IupLua binding. Lua print calls are output in the console. The executable also include the [LuaGL](#), CDLua and IMLua bindings.

Since the libraries are statically linked the Lua "require" does not work. To build the IupLua executables with dynamic loading enabled in Windows we use:

```
iup/srclua5> tecmake dll "MF=iuplua" "USE_DLL=Yes"
```

These are available in a separate folder together with all the necessary DLLs.



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

[Ed Patt](#) - Pattern Editor

[Ftool](#) - Two-dimensional Frame Analysis Tool

System

IUP has several global tables as together 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:

```
int main(int argc, char* argv[])
{
    if (IupOpen(&argc, &argv) == IUP_ERROR)
    {
        fprintf(stderr, "Error Opening IUP.")
        return;
    }

    ...
    IupMainLoop();
    IupClose();

    return 0;
}
```

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 IupLua, 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 IupLua we follow the same organization of the Lua libraries using a namespace (“iup”) before all the definitions.

- All exported functions are accessed only through **iup.FunctionName**, including control initialization like **iup.label**.
- All callbacks in are accessed through their exact name in the C API.
- Numeric definitions were kept in upper case without the IUP_ prefix, such as **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 use a define to interchange 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 (**IupLua3**) and one for Lua 5.1 (**IupLua51**), that you can use to test your Lua code. Both applications have support for all the additional controls and are available at the [Download](#).

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(int argc, char* argv[])
{
    IupOpen(&argc, &argv);
    IupControlsOpen();

    /* Lua 5 initialization */
    L = lua_open();
    luaL_openlibs(L);

    iuplua_open(L); /* Initialize Binding Lua */
    iupcontrolslua_open(L); /* Initialize additional controls binding Lua */

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

    IupMainLoop(); /* could be here or inside "myfile.lua" */

    lua_close(L);

    IupControlsClose();
    IupClose();
}
```

```

    return 0;
}

```

See the example [iuplua51_init.c](#) for Lua 5.1.

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 "loadlib". 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**.

Here is an example on how to dynamically load IUP in Lua 5.1:

```

local iuplua_open = package.loadlib("iuplua51.dll", "iuplua_open")
iuplua_open()
local iupcontrolslua_open = package.loadlib("iupluacontrols51.dll", "iupcontrolslua_open")
iupcontrolslua_open()

```

Lua 5.1 "require" can be used for all the **IupLua** libraries, but the full library name should be used. For example: **require"iuplua51"**, **require"iupluacontrols51"**, and so on. Additionally the **LUA_CPATH** in UNIX must add the prefix "lib" to the search path, for example:

```

LUA_CPATH = ./\?.so\;./lib\?.so\;$LIBPATH_ARCH/\?.so\;$LIBPATH_ARCH/lib\?.so

```

You can also use **require"iuplua"** and so on, but the **LUA_CPATH** must contains the "51" pattern. For example:

```

LUA_CPATH = ./\?.so\;./lib\?51.so\;$LIBPATH_ARCH/\?.so\;$LIBPATH_ARCH/lib\?51.so

```

The simplest form **require"iup"** and so on, can not be used because there are IUP dynamic libraries with names that will conflict with the names used by **require** during search.

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 "Ihandle*" 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.

In C, to push an Ihandle in Lua's stack, use the function:

```
iuplua_pushihandle(Ihandle *ih);           // for Lua3
iuplua_pushihandle(lua_State *L, Ihandle *ih); // for Lua5
```

In C, to receive an Ihandle in a C function called from Lua, just use one of the following code:

```
Ihandle* ih = (Ihandle*)lua_getuserdata(object); // for Lua 3
Ihandle* ih = *(Ihandle**)lua_touserdata(L, pos); // for Lua 5
```

or using parameter checking:

```
Ihandle* iuplua_checkihandle(int pos);           // for Lua 3
Ihandle* iuplua_checkihandle(lua_State *L, int pos); // for Lua 5
```

In Lua, if the handle is a user data create with the above structure, but not mapped to a Lua object, use the function:

```
iup.RegisterHandle(handle, typename)
```

where "typename" is the string returned in IupGetType(handle)

In Lua, to access a handle created in C as a Lua object, alternatively use the function:

```
handle = iup.GetFromC(name)
```

where "name" is the name of the element previously defined with IupSetHandle.

Error Handling

In Lua 3 the `_ALERT` function is redefined to show the message in a dialog.

In Lua 5 there is no such function, so 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);
```

These functions mimics the implementation in the standalone interpreter for Lua 5, that displays the error message followed by the stack.

If the these functions are used the errors will be reported through the "iup._ERRORMESSAGE(msg)" function. By default `_ERRORMESSAGE` is defined to show a dialog with the error message.

When printing an Ihandle reference the returned string is "IUP(*type*): *address*", for example "IUP(dialog): 08C55240".

The Architecture Behind IupLua 3

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 3 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 for the standard controls 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

```

The Architecture Behind IupLua 5

There are two important names in IupLua5: "iup handle" and "iup widget".

When you create an IUP element in Lua 5 it is created a table with a metatable called "iup widget". This metatable has its "__index" method redefined so when an index is not defined it looks for it in the "parent" table. The table it self represents the class of the control. And all the classes inherit the implementation of the base class WIDGET. Each control class must implement the "createElement" method of the class. The WIDGET class also a member called "handle" that contains the Ihandle* in Lua. The constructor of the WIDGET class returns the handle.

The Ihandle* is represented in Lua as a table with a metatable called "iup handle". This metatable has its "__index", "__newindex" and "__eq" methods redefined. The index methods are used to implement the set and get attribute facility. The handle knows its class because it is stored in its "parent" member.

Since the controls creation is done by the "iup.<control>" function, the application does not use the WIDGET class directly. All the time the application only uses the handle.

So for example the IupLabel:

```

iup.label calls iup.LABEL:constructor
since iup.LABEL.parent = iup.WIDGET and iup.LABEL:constructor is not implemented
it calls iup.WIDGET:constructor
then iup.WIDGET:constructor calls iup.LABEL:createElement
and finally returns the created handle

```

The complete class hierarchy for the standard controls can be represented as follows:

```

WIDGET
  BUTTON
  ITEM
  TOGGLE
  CANVAS
  BOX
    HBOX
    VBOX
    ZBOX
  MENU
  DIALOG
  FILL
  FRAME
  IMAGE
  LABEL
  LIST
  RADIO
  SEPARATOR
  SUBMENU
  TEXT
  MULTILINE

```

IupOpen

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

Parameters/Return

```

int IupOpen(int *argc, char ***argv); [in C]
[There is no equivalent in Lua]

```

argc and **argv**: are the same as the application "main" function. Some parameters processed by the driver can be removed so the address is necessary. They can be NULL. (Since 3.0)

This function returns IUP_OPENED (already opened), IUP_ERROR or IUP_NOERROR.

Only in UNIX can fail to open, because X-Windows may be not initialized.

Notes

In Windows, CoInitializeEx(COINIT_APARTMENTTHREADED) and InitCommonControlsEx(ICC_WIN95_CLASSES) functions are called.

In Motif, XtOpenApplication function is called.

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

Environment Variables

The toolkit's initialization depends also on platform-dependent environment variables, see each driver documentation.

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 to YES, IUP generates a message dialog indicating the driver's version when initializing. Default: NO.

Lua Binding

This function should be called by the host program and before the IupLua initialization function **iuplua_open**. If not the IupLua initialization function will call it.

See Also

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

IupClose

Ends the IUP toolkit and releases internal memory.

Parameters/Return

```
void IupClose(void); [in C]
iup.Close() [in Lua]
```

Notes

In Windows, the CoUninitialize function is called.

In Motif, the XtDestroyApplicationContext function is called.

Lua Binding

This function usually will be called by the host program. But if IUP is dynamically loaded in Lua 5 then you should call **iup.Close** from Lua.

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]
```

Returns: 0 (the number of elements in the stack).

Notes

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

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]
iup.Version() -> (version: string) [in Lua]
```

Definitions

```
[in C]
IUP_NAME          "IUP - Portable User Interface"
IUP_COPYRIGHT     "Copyright (C) 1994-2007 Tecgraf/PUC-Rio and PETROBRAS S/A"
IUP_DESCRIPTION   "Portable toolkit for building graphical user interfaces."
IUP_VERSION       "2.6.0"
IUP_VERSION_DATE  "2007/05/10"
IUP_VERSION_NUMBER 206000

[in Lua]
iup._NAME
iup._DESCRIPTION
```

```
iup._COPYRIGHT
iup._VERSION
iup._VERSION_DATE
iup._VERSION_NUMBER
```

IupLoad

Compiles a LED specification.

Parameters/Return

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

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.

Notes

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 some pre-defined dialogs. This is an old function, it just sets the global attribute LANGUAGE.

Parameters/Return

```
void IupSetLanguage(const char *lng); [in C]
iup.SetLanguage(lng: string) [in Lua]
```

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

- "ENGLISH"
- "PORTUGUESE"

default: "ENGLISH".

Affects

All elements that have predefined texts.

Examples

```
#include "iup.h"

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

See Also

[IupGetLanguage](#), [LANGUAGE](#)

IupGetLanguage

Returns the global attribute LANGUAGE.

Parameters/Return

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

See Also

[IupSetLanguage](#), [LANGUAGE](#).

Motif System Driver

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

Environment Variables

DEBUG

This variables 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 users home or in a directory pointed to by the XAPPLRESDIR environment variable. Below you can see an example of this files 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.

Also a resource file named ".Xdefaults" will also affect the visual appearance of all applications that use Motif and Intrinsics.

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 SunSolves <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 servers 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 or 64 bits (2000/XP/2003/Vista).

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

COM Initialization

IupOpen calls "CoInitializeEx(NULL, COINIT_APARTMENTTHREADED);". if you need another concurrency model call CoInitializeEx with other options before calling IupOpen. Be aware that some features in some controls require single-thread apartment, and they will stop working, this includes: IupFileDLg when selecting a folder, and IupOle.

InitCommonCtrlEx Linker Error

On Windows a common error occurs: "Cannot find function **InitCommonCtrlEx()**" 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

Windows 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".

When using Visual C++ 8 with "iup.manifest", configure the linker properties of your project to do NOT generate a manifest file or the Windows XP Visual Styles won't work.

Black Canvas

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

GTK System Driver (since 3.0)

This driver was designed for the GTK+ version 2. It can be compiled in Windows or UNIX.

Tips

Although the GTK driver can be compiled and used in Windows, it is not recommended since it is slower and much more memory consuming than the IUP native Windows driver.

Although GTK has layout elements they are not used. IUP fill, vbox, hbox and zbox containers are implemented independent from the native system.

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.

Attribute names are always upper case, lower case names will not work. But attribute values like "YES", "NO", "TOP", are case insensitive, so "Yes", "no", "top", and other variations will work.

If not defined their value can be inherited from the parent container.

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.
- **IupStoreAttribute** 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 inside **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 updated (**Set**) it is stored internally at the hash table of the control only if the driver implementation of the control class allows the storage. If the value is NULL, the attribute will also be removed from the hash table. Finally the attribute is updated for the children of the control if they do not have the attribute defined in their own hash table. Here is a pseudo-code:

```
IupSetAttribute(ih, name, value)
{
    if ih.SetAttribute(name, value) then
        ih.hash_table.Set(name, value)
    endif

    if (ih.IsInheritable(name))
        for each child of ih do
            if not ih.hash_table.IsDefined(name) then
                child.SetAttribute(name, value)
                child.NotifyChildren(name, value)
            endif
        endfor
    endif
}
```

When an attribute is retrieved (**Get**) it will first be checked at the driver implementation of the control class. If not defined then it checks in the hash table. If not defined it checks its parent hash table and so forth, until it reaches the dialog. And finally if still not defined then a default value is returned (the default value can also be NULL).

```
value = IupGetAttribute(ih, name)
{
    value = ih.GetAttribute(name)
```

```

if not value then
    value = ih.hash_table.Get(name)
endif

if not value then
    value = ih.GetParentHashTableAttribute(name)
endif

if not value then
    value = ih.GetDefaultAttribute(name)
endif
}

```

The driver implementation of the control class can update or retrieve a value even if the control is not mapped. This is enabled for only a few attributes.

Attribute names are always upper case, lower case names will not work. But attribute values like "YES", "NO", "TOP", are case insensitive, so "Yes", "no", "top", and other variations will work. Boolean attributes accept the values "1", "YES", "ON" or "TRUE" for **true**, and NULL (not defined), "0", "NO", "OFF" or "FALSE" for **false**, even if in the documentation is only one of these combinations.

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 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 vboxes, 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. Some examples: "TIP", "ZORDER", "TITLE", "VALUE", "ALIGNMENT", "X", "Y", "RASTERSIZE", "SIZE", "IMAGE", "IMINACTIVE", "IMPRESS", "SEPARATOR".

Id numbered attributes are NON inheritable always, like "1" or "MARK1:1". Attributes that handle names like "IMAGE" and "MENU" are NON inheritable always. Attributes that are read-only or write-only are NON inheritable.

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 transferred. 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 = iup.button(title = "Ok", bgcolor = "0 255 0")
```

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 parameters 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 = iup.hbox(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 a handle, 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.

IupStoreAttribute

Defines an attribute for an interface element.

Parameters/Return

```
void IupStoreAttribute(Ihandle *ih, char *name, char *value); [in C]
iup.StoreAttribute(ih: iulua_tag, name: string, value: string) [in Lua]
```

ih: identifier of the interface element. If NULL will set in the global environment.

name: name of the attribute.

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

Notes

The value stored is duplicated if not processed by the element specific implementation. 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 *ih, char *name, char *value); [in C]
Iup.SetAttribute(ih: iulua_tag, name: string, value: string) [in Lua]
```

ih: Identifier of the interface element. If NULL will set in the global environment.

name: name of the attribute.

value: value of the attribute. If 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 internally stored,
                                                // but its memory will be released at the end of this scope)
}
// Use IupStoreAttribute in this case

{
    char *value = malloc(30);
    sprintf(value, "%d", i);
    IupSetAttribute(dlg, "EXAMPLE", value) // correct (but to avoid memory leaks you should free the pointer
                                                after the dialog has been destroyed)
}

IupSetAttribute(dlg, "VISIBLE", "YES") // correct (static values still exists after this scope)

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

struct myData* mydata = malloc(sizeof(struct myData));
IupSetAttribute(dlg, "MYDATA", (char*)mydata) // correct (unknown attributes will be stored as pointers)
```

Internal Attribute Environment

When an attribute is set it is stored in the internal attribute environment if not processed by the element specific implementation. If the value is NULL, the attribute will be removed from the environment.

Inheritance

After the attribute is set because of the attribute inheritance its children will be notified that an attribute has changed.

Examples

Defines a radio's initial value.

```
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 radio */
IupSetAttribute(modo, "VALUE", "landscape"); /* defines the radio's initial value */
```

Some usages:

```
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](#)

IupSelfAttribute

Defines an attribute for an interface element.

Parameters/Return

```
void IupSetfAttribute(Ihandle *ih, char *name, char *format, ...); [in C]
[There is no equivalent in Lua]
```

ih: identifier of the interface element.

name: name of the attribute.

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

...: values of the attribute.

Notes

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](#)

IupSetAttributes

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

Parameters/Return

```
Ihandle *IupSetAttributes(Ihandle *ih, char *str); [in C]
iup.SetAttributes(ih: iuplua_tag, str: string) -> ih: iulua_tag [in Lua]
```

ih: Identifier of the interface element.

str: string with the attributes in the format "v1=a1, v2=a2,..." where vi is the name of an attribute and ai is its value.

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

Examples

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

```
dialog = IupSetAttributes(
  IupDialog(
    IupVBox(
      IupSetAttributes(IupFill(), "SIZE = 5"),
      IupHBox(
        IupSetAttributes(IupFill(), "SIZE = 5"),
        canvas = IupSetAttributes(IupCanvas("repaint_cb"), "BORDER=NO, RASTERSIZE=100x100"),
        IupSetAttributes(IupFill(), "SIZE = 5"),
        NULL),
      IupSetAttributes(IupFill(), "SIZE = 5"),
      NULL)),
  "TITLE = Teste")
```

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

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

See Also

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

IupSetAttributeHandle

Instead of using IupSetHandle and IupSetAttribute with a new creative name, this function automatically creates a non conflict name and associates the name with the attribute.

It is very usefull for associating images and menus.

Parameters/Return

```
void IupSetAttributeHandle(Ihandle *ih, char *name, Ihandle *ih_named); [in C]
[There is no equivalent in Lua]
```

ih: identifier of the interface element.

name: name of the attribute.

ih_named: element to associate using a name

The function will not check for inheritance since all the attributes that associate handles are not inheritable.

See Also

[IupGetAttributeHandle](#), [IupSetAttribute](#), [IupSetAttributes](#), [IupStoreAttribute](#), [IupSetHandle](#)

IupGetAttributeHandle

Instead of using IupGetAttribute and IupGetHandle, this function directly returns the associated handle.

Parameters/Return

```
Ihandle* IupGetAttributeHandle(Ihandle *ih, char *name); [in C]
[There is no equivalent in Lua]
```

ih: identifier of the interface element.
name: name of the attribute.

Returns the element with the associated name. The function will not check for inheritance since all the attributes that associate handles are not inheritable.

See Also

[IupSetAttributeHandle](#), [IupSetAttribute](#), [IupSetAttributes](#), [IupStoreAttribute](#), [IupSetHandle](#)

IupGetAttribute

Returns the name of an interface element attribute.

Parameters/Return

```
char *IupGetAttribute(Ihandle *ih, char *name); [in C]
iup.GetAttribute(ih: ihandle, name: string) -> value: string [in Lua]
```

ih: Identifier of the interface element. If NULL will retrieve from the global environment.
name: 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.

Inheritance

The attribute is first checked at the element specific implementation. If not defined then it checks in the attribute environment. If not defined in its attribute environment, the attribute will be inherited from its parent attribute environment 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

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).

Examples

[Example File](#)

See Also

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

IupGetAllAttributes

Returns the names of all attributes of a element. The internal attributes are also returned (attributes prefixed with "_IUP").

Parameters/Return

```
int IupGetAllAttributes(char *names[], int max_n); [in C]
iup.GetAllAttributes(max_n: number) -> (names: table, n: number) [in Lua]
```

names: table receiving the names
max_n: maximum number of names the table can receive.

Returns the number of names loaded to the table.

See Also

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

IupGetAttributes

Returns all attributes of a given element that are in the internal attribute environment. The known attributes that are pointers (not strings) are returned as integers.

The internal attributes are not returned (attributes prefixed with "_IUP").

Before calling this function the application must ensure that there is no pointer attributes set for that element.

This function should be avoided. Use **IupGetAllAttributes** instead.

Parameters/Return

```
char* IupGetAttributes (Ihandle *ih); [in C]
iup.GetAttributes(ih: iulua_tag) -> (str: string) [in Lua]
```

ih: Identifier of the interface element.

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

Returns a string with all attributes in the format: "NAME=VALUE, NAME="VALUE", ...".

See Also

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

IupGetFloat

Returns the value of an interface element attribute as a floating point number.

Parameters/Return

```
float IupGetFloat(Ihandle *ih, char *name) [in C]
[There is no equivalent in IupLua]
```

ih: Identifier of the interface element.

name: name of the attribute.

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

See Also

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

IupGetInt

Returns the value of an interface element attribute as an integer.

Parameters/Return

```
int IupGetInt(Ihandle *ih, char *name); [in C] - first integer found on string
int IupGetInt2(Ihandle *ih, char *name); [in C] - second integer found on string
int IupGetIntInt(Ihandle *ih, char *name, int *i1, int *i2); [in C] - first and second integers found on string, returns the number of .
[There is no equivalent in IupLua]
```

ih: Identifier of the interface element.

name: name of the attribute.

Notes

In the first form, if the attribute value is "YES"/"NO" or "ON"/"OFF", the function returns 1 / 0, respectively.

See Also

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

IupStoreGlobal

Defines an attribute for the global environment. If the driver process the attribute then it will not be stored internally.

Parameters/Return

```
void IupStoreGlobal(const char *name, const char *value); [in C]
iup.StoreGlobal(name: string, value: string) [in Lua]
```

name: name of the attribute.

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

Notes

The difference between IupSetGlobal and IupStoreGlobal is the same of IupSetAttribute and IupStoreAttribute. In the Store functions the value is duplicated internally. In the **IupStoreGlobal** is duplicated only if not processed by the driver.

The application can also store any private attribute in the global environment.

IupStoreAttribute can also be used to set global attributes, just set the element to NULL.

See Also

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

IupSetGlobal

Defines an attribute for the global environment. If the driver process the attribute then it will not be stored internally.

Parameters/Return

```
void IupSetGlobal(const char *name, const char *value); [in C]
iup.SetGlobal(name: string, value: string) [in Lua]
```

name: name of the attribute.

value: 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**.

IupSetAttribute can also be used to set global attributes, just set the element to NULL.

See Also

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

IupGetGlobal

Returns an attribute value from the global environment. The value can be returned from the driver or from the internal storage.

Parameters/Return

```
char *IupGetGlobal(const char *name); [in C]
iup.GetGlobal(name: string) -> value: string [in Lua]
```

name: name of the attribute.

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

IupGetAttribute can also be used to get global attributes, just set the element to NULL.

See Also

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

ACTIVE

Activates or inhibits user interaction.

In GTK and Motif the inactive dialogs will still be able to move, resize and change their Z-order. Although their contents will be inactive, keyboard will be disabled, and they can not be closed from the close box.

Value

"YES" (active), "NO" (inactive).

Default: "YES".

Affects

All.

BGCOLOR

Element's background color.

Value

The RGB components. Values should be between 0 and 255, separated by a blank space. For example "255 0 128", red=255 blue=0 green=128.

Default: It is the value of the DLGBGCOLOR global attribute.

Affects

All.

See Also

[FGCOLOR](#), [DLGBGCOLOR](#)

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: "0 0 0".

Affects

All.

See Also

[BGCOLOR](#)

FONT (up to 2.x)

Character font of the text shown in the element. See [FONT](#) definition since IUP 3.0.

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 greater 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.

See the [Layout Guide](#) for more details on sizes.

Value

"YES" (both directions), "HORIZONTAL", "VERTICAL" or "NO".

Default: Depends on the element. When not specified otherwise, the default value is "NO".

Affects

All, except menus.

RASTERSIZE (non inheritable)

Specifies the element **User** size, and returns the **Current** size, in pixels.

Value

"widthxheight", where *width* and *height* are integer values corresponding to the horizontal and vertical size, respectively, in pixels.

Affects

All, except menus.

Notes

Please refer to the notes on the SIZE attribute for further detail.

When this attribute is changed, the [SIZE](#) attribute is automatically updated.

See Also

[SIZE](#)

SIZE (non inheritable)

Specifies the element **User** size, and returns the **Current** size, in units proportional to the size of a character.

See the [Layout Guide](#) for more details on sizes.

Value

"*widthxheight*", where *width* and *height* are integer values corresponding to the horizontal and vertical size, respectively, in characters fraction unit (see Notes below).

You can also set only one of the parameters by removing the other one and maintaining the separator "x", but this is equivalent of setting the other value to 0. For example: "x40" (height only = "0x40") or "40x" (width only = "40x0").

When this attribute is consulted the **Current** size of the control is returned.

Notes

The size units observes the following heuristics:

- Width in 1/4's of the average width of a character for the current FONT of each control.
- Height in 1/8's of the average height of a character for the current FONT of each control.

SIZE depends on FONT, so FONT must be set before SIZE for correct size calculation.

When this attribute is changed, the [RASTERSIZE](#) attribute is automatically updated.

A **User** size of "0x0" can be set, it can also be set using NULL.

If you wish to use the **User** 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.

The element is NOT immediately repositioned. Call **IupRefresh** to update the dialog layout.

IupMap also updates the dialog layout even if it is already mapped, so calling it or calling **IupShow**, **IupShowXY** or **IupPopup** (they all call **IupMap**) will also update the dialog layout.

See the [Layout Guide](#) for mode details on sizes.

Affects

All, except menus.

See Also

[EXPAND](#), [RASTERSIZE](#), [IupRefresh](#)

CLIENTSIZE (read-only) (non inheritable)

Returns the container **Current** size, in pixels, excluding the decorations. Usefull for IupFrame, IupTabs and IupDialog. Can also be consulted in IupVbox and IupHbox, it will excludes the margins.

Value

"*widthxheight*", where *width* and *height* are integer values corresponding to the horizontal and vertical size, respectively, in pixels.

Affects

All, except menus.

See Also

[SIZE](#), [RASTERSIZE](#)

POSITION (read-only) (non inheritable)

Returns the position of the element relative to the native parent.

Value

"x,y", where *x* and *y* are integer values corresponding to the horizontal and vertical position, respectively, in pixels.

Affects

All, except menus.

See Also

[SIZE](#), [RASTERSIZE](#)

TIP

Text to be shown when the mouse lies over the element.

Value

Text.

Notes

The FONT used for the text is the same as the FONT selected for the element.

In Windows, the mouse must remain stationary to show the TIP. In Motif, the TIP is always shown after the mouse enter the window.

In Windows, if the mouse remains stationary until the TIP is hidden, the TIP is not shown again until a TIP of another control is shown.

In GTK the TIP will be shown outside the element near the border. Also all additional attributes, including FONT, are predetermined and depend on the native system.

Additional Tip Attributes (Windows and Motif Only) (since 3.0)

These attributes affect the TIP display. Set the additional attributes before setting the TIP attribute.

TIPDELAY: Time the tip will remain visible. Default: "5000".

TIPBGCOLOR: The tip background color. Default: "255 255 225" (Light Yellow)

TIPFGCOLOR: The tip text color. Default: "0 0 0" (Black)

TIPBALLON [Windows Only]: The tip window will have the appearance of a cartoon "balloon" with rounded corners and a stem pointing to the item.. Default: NO.

TIPBALLONTITLE [Windows Only]: When using the ballon format, the tip can also has a title in a separate area.

TIPBALLONTITLEICON [Windows Only]: When using the ballon format, the tip can also has a pre-defined icon in the title area. Values can be:

"0" - No icon
"1" - Info icon
"2" - Warning icon
"3" - Error Icon

TIPVISIBLE: Action attribute used to show or hide the tip under the mouse cursor. Use values "YES" or "NO". This will work in GTK but it will trigger the tip state, the given value will be ignored.

TIPMARKUP [GTK only]: allows the tip string to contains pango markup commands. Can be "YES" or "NO". Default: "NO".

TIPICON [GTK only]: name of an image to be displayed in the TIP. See the [IMAGE](#) attribute.

Affects

All except label, menu item and submenu item.

TITLE (non inheritable)

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 [LupText](#).

For the [LupRadio](#) and [LupZbox](#), 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 [LupSetHandle](#) 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.

VISIBLE

Shows or hides the element.

Value

"YES" (visible), "NO" (hidden).

Default: "YES"

Notes

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

Affects

All, except menus.

WID (read-only) (non inheritable)

Element identifier in the native interface system.

Value

In Motif, returns the **Widget** handle.

In Windows, returns the **HWND** handle.

In GTK, return the **GtkWidget*** handle.

Notes

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

For elements that do not have a native representation, NULL is returned.

Affects

All.

XWINDOW [Motif Only] (read-only) (non inheritable)

Returns the X-Windows Window.

Affects

All that have a native representation.

For elements that do not have a native representation, the native parent XWINDOW is returned.

X (read-only) (non inheritable)

Returns the absolute horizontal position in pixels relative to the upper left corner of the screen.

Value

Integer number.

Affects

All controls that have visual representation.

Y (read-only) (non inheritable)

Returns the absolute vertical position in pixels relative to the upper left corner of the screen.

Value

Integer number.

Affects

All controls that have visual representation.

ZORDER (write-only) (non inheritable)

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.

Global Attributes

LANGUAGE

The language used by some pre-defined dialogs.

Can have the values ENGLISH and PORTUGUESE. Default: ENGLISH. Can also be set by **IupSetLanguage**.

SYSTEMLANGUAGE (read-only)

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

VERSION (read-only)

Returns the name of IUP's version.

The value follows the "major.minor.micro" format, major referring to broader changes, minor referring to smaller changes, and micro referring to corrections only. Ex.: "1.7.2".

COPYRIGHT (read-only)

Returns the IUP's copyright.

Ex: "Copyright (C) 1994-2004 Tecgraf/PUC-Rio and PETROBRAS S/A".

DRIVER (read-only)

Informs the current driver being used.

Two drivers are available now, one for each platform: "GTK", "Motif" and "Win32".

SYSTEM (read-only)

Informs the current operating system.

On UNIX, it is equivalent to the command "uname -s" (sysname). On Windows, it identifies if you are on Windows 2000, Windows XP, Windows Server 2003 or Windows Vista.

Examples:

- "Linux"
- "SunOS"
- "Solaris"
- "IRIX"
- "AIX"
- "HP-UX"
- "Win2K"
- "WinXP"
- "Vista"

SYSTEMVERSION (read-only)

Informs the current operating system version number.

On UNIX, it is equivalent to the command "uname -r" (release). On Windows, it identifies the system version number and service pack version.

FULLSIZE (read-only)

Returns the full screen size in pixels.

String in the "*widthxheight*" format.

SCREENSIZE (read-only)

Returns the screen size in pixels available for dialogs, i.e. not including menu bars, task bars, etc. In Motif has the same value as the FULLSIZE attribute.

String in the "*widthxheight*" format.

SCREENDEPTH (read-only)

Returns the screen depth in bits per pixel.

LOCKLOOP

When the last visible dialog is closed the **IupExitLoop** function is called. To avoid that set LOCKLOOP=YES before hiding the last dialog. Possible values: "YES" or "NO". Default: "NO".

CURSORPOS

The cursor position in absolute coordinates relative to the upper left corner of the screen. Accept values in the format "XxY", example "200x200",

COMPUTERNAME (read-only)

Returns the hostname.

USERNAME (read-only)

Returns the user logged in.

DLGBGCOLOR

Returns the default dialog background color.

DEFAULTFONT

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

SHIFTKEY (read-only)

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

CONTROLKEY (read-only)

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

MODKEYSTATE (read-only) (since 3.0)

Returns the state of the keyboard modifier keys: Shift, Ctrl, Alt and sYs(Win/Apple). In the format of 4 characters: "SCAY". When not pressed the respective letter is replaced by a space " ".

TRUECOLORCANVAS (read-only)

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

Win32 Global Attributes

HINSTANCE (read-only)

This attribute returns a handle (HINSTANCE) that identifies the application in the native system.

GTK Global Attributes

GTKVERSION (read-only)

Returns the version of the run time GTK.

Motif Global Attributes

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.

APPSHELL (read-only) (since 3.0)

Returns the shell Widget created by XtOpenApplication.

MOTIFVERSION (read-only)

Returns the version of the run time Motif.

MOTIFNUMBER (read-only)

Returns the number of the Motif Version if full form, e.x: 2.2.3 = "2203".

XDISPLAY (read-only) (since 3.0)

Returns the X-Windows Display.

XSCREEN (read-only) (since 3.0)

Returns the X-Windows Screen.

XSERVERVENDOR (read-only) (since 3.0)

X-Windows Server Vendor string.

XVENDORRELEASE (read-only) (since 3.0)

X-Windows Server Vendor release number.

Events and Callbacks

IUP is a graphics 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 an event has taken place. Hence events are handled through callbacks, which 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 creates and shows 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. These dialogs will have their 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.

The OLD method 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:

```
# In LED, is equivalent to the IupSetAttribute command in the previous example.
bt = button("Title", my_button_action)
```

In the NEW method, the application does not needs a global name, it directly sets the callback using the attribute name using **IupSetCallback**. For example:

```
int myButton_action(Ihandle* self);
...
IupSetCallback(myButton, "ACTION", (Icallback)myButton_action);
```

The new method is more efficient and more secure, because there is no risk of a name conflict. If the application uses LED, just ignore the name in the LED, and replace **IupSetFunction** by **IupSetCallback**.

Although enabled in old versions, callbacks do NOT have **inheritance** like attributes.

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: Call **IupExitLoop** after return. Depending on the state of the application it will close all windows and exit the application. Applies only to some actions.
- IUP_IGNORE: Makes the native system ignore that callback action. Applies only to some actions.
- IUP_CONTINUE: Makes the element to ignore the callback and pass the treatment of the execution to the parent element. Applies only to some actions.

Only some callbacks support the last 3 return values. Check each callback documentation. When nothing is documented then only IUP_DEFAULT is supported.

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.

The order of callback calling is system dependent. For instance, the RESIZE_CB and the SHOW_CB are called in different order in Win32 and in X-Windows when the dialog is shown for the first time.

To help the definition of callbacks in C, the header "iupcbs.h" can be used, there are typedefs for all the callbacks.

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, calling **IupExitLoop** or by hiding the last visible dialog, the function **IupMainLoop** will return.

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

IupLua

Callbacks in Lua have the same names and receive the same parameters as callbacks in C, in the same order. In Lua the callbacks they can either return a value or not, the IupLua binding will automatically return IUP_DEFAULT if no value is returned. In Lua callbacks can be Lua functions or strings with Lua code.

The callbacks can also be implemented as methods, using the language’s resources for object orientation. Thus, the element is implicitly passed as the **self** parameter.

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
```

Or also

```
myButton.action = function (self)
    ...
end
```

Or, as a string

```
myButton.action = "local aux = self.fgcolor;
    self.fgcolor = self.bgcolor;
    self.bgcolor = aux"
```

Although some callbacks exists only in specific controls, all the callbacks can be set for all the controls. This is usefull to set a callback for a box, so it will be inherited by all the elements inside that box which implements that callback.

IupMainLoop

Executes the user interaction until a callback returns IUP_CLOSE, **IupExitLoop** is called, or hiding the last visible dialog.

Parameters/Return

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

Returns IUP_NOERROR always.

Notes

When this function is called, it will interrupt the program execution until a callback returns IUP_CLOSE, **IupExitLoop** is called, or there are no visible dialogs.

A second call to **IupMainLoop** will have a platform-dependent behavior. In the Motif driver can be called several times but a IUP_CLOSE must occur for each call. In the Win32 driver one IUP_CLOSE will end all calls. If there are no visible dialogs all calls will end in both drivers.

If **IupMainLoop** is called without any visible dialogs and no active timers, the application will hang and will not be possible to close the main loop. The process will have to be interrupted by the system.

When the last visible dialog is hidden the **IupExitLoop** function is called. To avoid that set LOCKLOOP=YES before hiding the last dialog.

See Also

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

IupLoopStep

Runs one iteration of the message loop.

Parameters/Return

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

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 replaces 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](#)

IupExitLoop

Terminates the current message loop. It has the same effect of a callback returning IUP_CLOSE.

Parameters/Return

```
void IupExitLoop(void); [in C]
iup.ExitLoop() [in Lua]
```

IupFlush

Processes all pending messages in the message queue.

Parameters/Return

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

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, call **IupFlush** after the attribute is changed.

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

IupGetCallback

Returns the callback associated to an event.

Parameters/Return

```
Icallback IupGetCallback(Ihandle* ih, const char *name); [in C]
[There is no equivalent in IupLua]
```

ih: identifier of the interface element.

name: attribute name of the callback.

Notes

This function replaces the combination:

```
IupGetFunction(IupGetAttribute(ih, name))
```

If an event is associated using **IupSetFunction** and **IupSetAttribute**, the **IupGetCallback** also returns the correct callback. So old applications work normally.

See Also

[IupSetCallback](#), [IupGetFunction](#)

IupSetCallback

Associates a callback to an event.

Parameters/Return

```
Icallback IupSetCallback(Ihandle* ih, const char *name, Icallback func); [in C]  
[There is no equivalent in Lua]
```

ih: identifier of the interface element.

name: attribute name of the callback.

func: address of a C function. If NULL removes the association.

Returns the address of the previous function associated to the action.

Notes

This function replaces the combination:

```
IupSetFunction(global_name, func);  
IupSetAttribute(ih, name, global_name);
```

So it eliminates the need for a global name.

Callbacks set using **IupSetCallback** can not be retrieved using **IupGetFunction**.

See Also

[IupGetCallback](#), [IupSetFunction](#)

IupGetActionName

Should return the name of the action being executed by the application. In fact returns only the name of the action last retrieved in **IupGetFunction**.

Parameters/Return

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

Returns the name of the action.

See Also

[DEFAULT_ACTION](#)

IupGetFunction

Returns the function associated to an action.

This function is now deprecated. The applications should use **IupGetCallback** instead.

Parameters/Return

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

name: name of the action.

See Also

[IupSetFunction](#), [IupGetCallback](#)

IupSetFunction

Associates a function to an action.

This function is now deprecated. The applications should use **IupSetCallback** instead.

Parameters/Return


```
Icallback IupSetFunction(const char *name, Icallback func); [in C]
[There is no equivalent in Lua]
```

name: name of an action.

func: address of a C function. If NULL removes the association.

Returns the address of the previous function associated to the action.

See Also

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

DEFAULT_ACTION

Predefined IUP action, generated every time an action has no associated function (except for the IDLE_ACTION).

Callback

```
int function(Ihandle *ih); [in C]
[There is no Lua equivalent]
```

ih: identifier of the element that activated the function.

Notes

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. This predefined 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. For example:

```
IupSetFunction("myFunctionName", (Icallback)myFunction);
IupSetAttribute(myButton, "ACTION", "myFunctionNamr"); /* notice the typo error here */
```

In this case the incorrect name "myFunctionNamr" (typo error here) will not be found, so if the DEFAULT_ACTION is defined it will be called when "ACTION" is invoked for the button. In fact it will be called for all the actions that do not have an action associated.

Affects

All callbacks when **IupSetFunction** is used. If **IupSetCallback** is used DEFAULT_ACTION is ignored.

See Also

[IupSetFunction](#), [IupGetActionName](#).

IDLE_ACTION

Predefined IUP action, generated when there are no events.

Callback

```
int function(void); [in C]
```

Return: IUP_CLOSE will be processed.

Notes

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

In Motif it register a background processing procedure.

In GTK it sets an idle function.

It can only be set using **IupSetFunction**.

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. But it is safer to use a timer for this purpose.

Lua Binding

To modify this action use the function **iup.SetIdle(myfunction)** in Lua. Using nil as a parameter to remove the association.

Examples

[Browse for Example Files](#)

See Also

[IupSetFunction](#), [IupTimer](#).

GETFOCUS_CB

Action generated when an element is given keyboard focus. This callback is called after the KILLFOCUS_CB.

Callback

```
int function(Ihandle *ih); [in C]
elem:getfocus_cb() -> (ret: number) [in Lua]
```

ih: identifier of the element that received keyboard focus.

Affects

All elements with user interaction, except menus.

See Also

[KILLFOCUS_CB](#)

KILLFOCUS_CB

Action generated when an element loses keyboard focus. This callback is called before the GETFOCUS_CB.

Callback

```
int function(Ihandle *ih); [in C]
elem:killfocus_cb() -> (ret: number) [in Lua]
```

ih: identifier of the element that activated the event.

Affects

All elements with user interaction, except menus.

See Also

[GETFOCUS_CB](#)

ENTERWINDOW_CB

Action generated when the mouse enters the canvas or button.

Callback

```
int function(Ihandle *ih); [in C]
elem:enterwindow_cb() -> (ret: number) [in Lua]
```

ih: identifier of the element that activated the event.

Affects

All controls with user interaction.

LEAVEWINDOW_CB

Action generated when the mouse leaves a canvas or button.

Callback

```
int function(Ihandle *ih); [in C]
elem:leavewindow_cb() -> (ret: number) [in Lua]
```

ih: identifier of the element that activated the event.

Affects

All controls with user interaction.

K_ANY

Action generated when a keyboard event occurs.

Callback

```
int function(Ihandle *ih, int c); [in C]
elem:k_any() -> (ret: number) [in Lua]
```

ih: identifier of the element that activated the event.

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

Return: If IUP_IGNORE is returned the key is ignored by the system. IUP_CLOSE will be processed. If returns IUP_CONTINUE, the event will be propagated to the parent of the element receiving it. Dialog "K_*" callbacks depend on this return value on child control "K_*" callbacks. If returns IUP_DEFAULT the key is processed but it is not propagated.

Notes

Keyboard callbacks depend on the keyboard usage of the control with the focus.

If the callback does not exists it is automatically propagated to the parent of the element.

K_* callbacks

All defined keys are also callbacks of any element, called when the respective key is activated. For example: "K_cC" is also a callback activated when the user press Ctrl+C, when the focus is at the element. This is the way an application can create shortcut keys, also called hot keys. These callbacks are not available in IupLua.

Affects

All elements with keyboard interaction.

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 *ih); [in C]
elem:help_cb() -> (ret: number) [in Lua]
```

ih: identifier of the element that activated the event.

Return: IUP_CLOSE will be processed.

Affects

All elements with user interaction.

ACTION

Action generated when the element is activated. Affects each element differently.

Callback

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

ih: identifier of the element that activated the event.

In some elements, this callback may receive more parameters, apart from **ih**. Please refer to each element's documentation.

Affects

[IupButton](#), [IupItem](#), [IupList](#), [IupText](#), [IupCanvas](#), [IupMultiline](#), [IupToggle](#)

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.

The composition elements includes 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	in IupLua
<pre>dialogo DIALOG 0, 0, 108, 34 STYLE WS_MINIMIZEBOX WS_MAXIMIZEBOX WS_CAPTION WS_SYSMENU WS_THICKFRAME CAPTION "Titulo" BEGIN PUSHBUTTON "Ok", IDOK, 16, 9, 33, 15 PUSHBUTTON "Cancel", IDCANCEL, 57, 9, 33, 15 END</pre>	<pre>dialogo = iup.dialog { iup.hbox { iup.fill(), iup.button(title="Ok", size="40"), iup.button(title="Cancel", size="40"), iup.fill() ;margin="15x15", gap="10" } ;title="Titulo" }</pre>
	

Now see the same dialog in LED and in C:

in LED	in C
<pre>DIALOG[TITLE="Titulo"] (HBOX[MARGIN="15x15", GAP="10"] (</pre>	<pre>dialog = IupSetAttributes(IupDialog (IupSetAttributes(IupHbox (IupFill(),</pre>

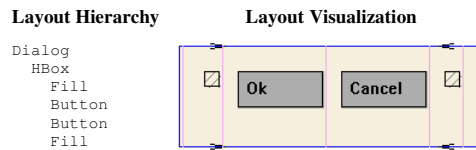
```

FILL(),
BUTTON[SIZE="40"] ("Ok",do_nothing),
BUTTON[SIZE="40"] ("Cancel",do_nothing),
FILL()
)
)

IupSetAttributes(IupButton("Ok", "do_nothing"), "SIZE=40"),
IupSetAttributes(IupButton("Cancel", "do_nothing"), "SIZE=40"),
IupFill(),
NULL
), "MARGIN=15x15, GAP=10"),
), "TITLE = Ttulo")

```

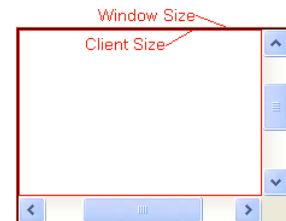
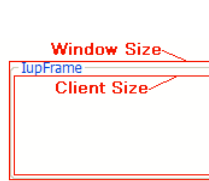
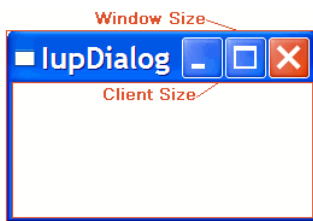
Following, the abstract layout representation of this dialog:



Layout Guide

Native Sizes (Window and Client)

Because of the dynamic nature of the abstract layout IUP elements have implicit many types of size. But the native elements have only two types of size: **Window** and **Client**. The **Window** size reflects the bounding rectangle of the element. The **Client** size reflects the inner size of the window that excludes the decorations and margins. For many elements these two sizes are equal, but for many container they are quite different. See some examples below.



The native **Client** size is used only internally to reposition the elements in the abstract layout, but it is available using the CLIENTSIZE attribute.

IUP Sizes

Natural Size

IUP does not require that the application specifies the size of any element. The sizes are automatically calculated so the contents of each element is fully displayed. This size is called **Natural** size. The **Natural** size is calculated only when the element is mapped to the native system.

The **Natural** size of a dialog is the size that allows all the elements inside the dialog to be fully displayed, and so on for all the containers inside the dialog.

So consider the following code and its result. Each button size is large enough to display their respective text. If the dialog size is increased or reduced by the size handlers in the dialog borders the buttons do not move or change their sizes.

```

dlg = iup.dialog
{
  iup.vbox
  {
    iup.button{title="Button Very Long Text"},
    iup.button{title="short"},
    iup.button{title="Mid Button"}
  }
  ;title="IupDialog", font="Helvetica, Bold 14"
}
dlg:show()

```



Current Size and User Size (SIZE or RASTERSIZE)

When the application defines the SIZE or RASTERSIZE attributes, it changes the **User** size in IUP. This size is used as a replacement for the **Natural** size if not a container, or as the minimum size for a container. The initial value is NULL. When set to NULL the **User** size is internally set to "0x0".

The returned value for SIZE or RASTERSIZE is the **Current** size in IUP. It returns the native **Window** size of the element when the element is mapped to the native system. Before mapping, the returned value is the **User** size defined by SIZE or RASTERSIZE attributes if any, otherwise they are NULL.

Defining the SIZE attribute of the buttons in the example we can make all have the same size. (In the following example the dialog size was changed after it was displayed on screen)

```

dlg = iup.dialog
{
  iup.vbox
  {
    iup.button{title="Button Very Long Text", size="50x"},
    iup.button{title="short", size="50x"},
    iup.button{title="Mid Button", size="50x"}
  }
  ;title="IupDialog", font="Helvetica, Bold 14"
}
dlg:show()

```



So when EXPAND=NO (see bellow) for elements that are not containers if **User** size is defined then the **Natural** size is ignored.

If you want to adjust sizes in the dialog do it after the layout size and positioning is done, i.e. after the dialog is mapped or after **IupRefresh** is called.

EXPAND

Another way to increase the size of elements is to use the EXPAND attribute. When there is room in the container to expand an element, the container layout will expand the elements that have the EXPAND attribute set to YES, HORIZONTAL or VERTICAL accordingly, even if they have the **User** size defined.

The default is EXPAND=NO, but for containers is EXPAND=YES.

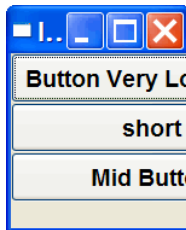
Using EXPAND in the example, we obtain the following result:

```
dlg = iup.dialog
{
  iup.vbox
  {
    iup.button{title="Button Very Long Text"},
    iup.button{title="short", expand="HORIZONTAL"},
    iup.button{title="Mid Button", expand="HORIZONTAL"}
  }
;title="IupDialog", font="Helvetica, Bold 14"
}
dlg:show()
```



So for elements that are NOT containers, when EXPAND is enabled the **Natural** size and the **User** size are ignored.

For containers the default behavior is to always expand or if expand is disabled they are limited to the **Natural** size. As a consequence (if the **User** size is not defined in all the elements) the dialog contents can only expand and its minimum size is the **Natural** size, even if EXPAND is enabled for its elements. In fact the actual dialog size can be smaller, but its contents will stop to follow the resize and they will be clipped at right and bottom.



SHRINK

To reduce the size of the dialog and its contents to a size smaller than the **Natural** size the SHRINK attribute of the dialog can be used. If set to YES all the containers of the dialog will be able to reduce its size. But be aware that elements may overlap and the layout result could be visually bad.

Notice that in the example the dialog initial size will be 0x0 because it is not defined. The picture shown is after resizing the dialog. So when using SHRINK usually you will also need to set the dialog initial size.

```
dlg = iup.dialog
{
  iup.vbox
  {
    iup.button{title="Button Very Long Text"},
    iup.button{title="short", expand="HORIZONTAL"},
    iup.button{title="Mid Button", expand="HORIZONTAL"}
  }
;title="IupDialog", shrink="yes", font="Helvetica, Bold 14"
}
dlg:show()
```



Layout Hierarchy

The layout of the elements of a dialog in IUP has a natural hierarchy because of the way they are composed together.

To create a node simply call one of the pre-defined constructors like **IupLabel**, **IupButton**, **IupCanvas**, and so on. To create a branch just call the constructors of containers like **IupDialog**, **IupFrame**, **IupVBox**, and so on. Internally they all call [IupCreate](#) to create branches or nodes. To destroy a node or branch call [IupDestroy](#).

Some of the constructors already append children to its branch, but you can append other children using [IupAppend](#). To remove from the tree call [IupDetach](#).

For the element to be visible [IupMap](#) must be called so it can be associated with a native control. **IupShow**, **IupShowXY** or **IupPopup** will automatically call **IupMap** before showing a dialog. To remove this association call [IupUnmap](#).

But there is a call order to be able to call these functions that depends on the state of the element. As you can see from these functions there are 3 states: **created**, **appended** and **mapped**. From **created** to **mapped** it is performed one step at a time. Even when the constructor receives the children as a parameter **IupApped** is called internally. When you **detach** an element it will be automatically **unmapped** if necessary. When you **destroy** an element it will be automatically **detached** if necessary.

A more simple and fast way to move an element from one position in the hierarchy to another is using [IupReparent](#).

The dialog is the root of the hierarchy tree. To retrieve the dialog of any element you can simply call [IupGetDialog](#), but there are other ways to navigate in the hierarchy tree.

To get all the children of a container call [IupGetChild](#) or [IupGetNextChild](#). To get just the next control with the same parent use [IupGetBrother](#). To get the parent of a control call [IupGetParent](#).

Layout Display

The layout size and positioning is automatically updated by **IupMap**. **IupMap** also updates the dialog layout even if it is already mapped, so using it or using **IupShow**, **IupShowXY** or **IupPopup** (they all call **IupMap**) will also update the dialog layout. The layout size and positioning can be manually updated using [IupRefresh](#), even if the dialog is not mapped.

After changing containers attributes or element sizes that affect the layout the elements are NOT immediately repositioned. Call **IupRefresh** for the container* to update all the dialog layout (* or any other element in the dialog, including the dialog itself).

To only force a redraw of an element and its children call [IupUpdate](#).

- [Creation](#)
- [Attributes](#)
- [Callbacks](#)
- [Notes](#)
- [Examples](#)
- [See Also](#)

IupFill

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

Its parent **must** be a IupHbox or a IupVbox.

It does not have a native representation.

Creation

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

Returns: the identifier of the created element, or NULL if an error occurs.

Attributes

[EXPAND](#): Ignored. If **User** size is not defined, then when inside a IupHbox EXPAND is HORIZONTAL, when inside a IupVbox EXPAND is VERTICAL. If **User** size is defined then EXPAND is NO.

[SIZE](#) / [RASTERSIZE](#) (non inheritable): Defines the width, if inside a IupHbox, or the height, if it is inside a IupVbox. When consulted behaves as the standard SIZE/RASTERSIZE attributes. SIZE uses the FONT attribute of the native parent to convert to pixels.

WID (read-only): returns NULL.

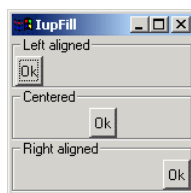
[X](#), [Y](#), [POSITION](#): also accepted.

Notes

This element is used to maintain the dialog's layout untouched after the user made size changes, and to align the interface elements.

Examples

[Browse Example Files](#)



See Also

[IupHbox](#), [IupVbox](#).

- [Creation](#)
- [Attributes](#)
- [Callbacks](#)
- [Notes](#)
- [Examples](#)
- [See Also](#)

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 *child, ...); [in C]
Ihandle* IupHboxv(Ihandle **children); [in C]
iup.hbox(child, ...: ihandle) -> (elem: ihandle) [in Lua]
hbox(child, ...) [in LED]
```

child,... : List of identifiers that will be placed in the box. NULL must be used to define the end of the list in C. It can be empty.

Returns: the identifier of the created element, or NULL if an error occurs.

Attributes

ALIGNMENT (non inheritable): Aligns the elements vertically. Possible values: "ATOP", "ACENTER", "ABOTTOM". Default: "ATOP".

EXPAND: The default value is "YES".

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 / RASTERSIZE (non inheritable): Defines the width of the box. When consulted behaves as the standard SIZE/RASTERSIZE attributes. SIZE uses the FONT attribute of the native parent to convert to pixels.

WID (read-only): returns NULL.

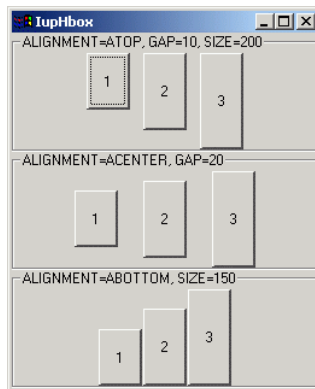
CLIENTSIZE, X, Y, POSITION: also accepted.

Notes

The box can be created with no elements and be dynamic filled using [IupAppend](#).

Examples

[Browse Example Files](#)



See Also

[IupZbox](#), [IupVBox](#)

- [Creation](#)
- [Attributes](#)
- [Callbacks](#)
- [Notes](#)
- [Examples](#)
- [See Also](#)

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 *child, ...); [in C]
Ihandle* IupVboxv(Ihandle **children); [in C]
iup.vbox(child, ...: ihandle) -> (elem: ihandle) [in Lua]
vbox(child, ...) [in LED]
```

child, ...: List of the identifiers that will be placed in the box. NULL must be used to define the end of the list in C. It can be empty.

Returns: the identifier of the created element, or NULL if an error occurs.

Attributes

ALIGNMENT (non inheritable): Horizontally aligns the elements. Possible values: "ALEFT", "ACENTER", "ARIGHT". Default: "ALEFT".

EXPAND: The default value is "YES".

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 / RASTERSIZE (non inheritable): Defines the height of the box. When consulted behaves as the standard SIZE/RASTERSIZE attributes. SIZE uses the FONT attribute of the native parent to convert to pixels.

WID (read-only): returns NULL.

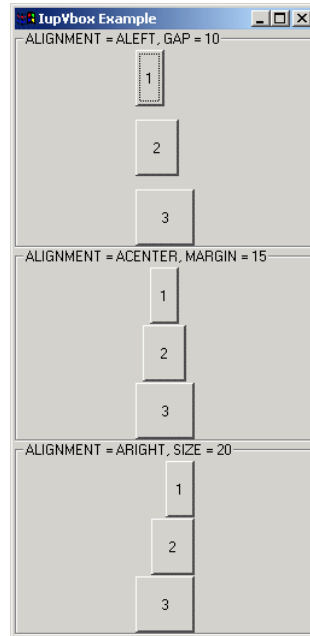
[CLIENTSIZE](#), [X](#), [Y](#), [POSITION](#): also accepted.

Notes

The box can be created with no elements and be dynamic filled using [IupAppend](#).

Examples

[Browse Example Files](#)



See Also

[IupZbox](#), [IupHbox](#)

- [Creation](#)
- [Attributes](#)
- [Callbacks](#)
- [Notes](#)
- [Examples](#)
- [See Also](#)

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 *child, ...); [in C]
Ihandle* IupZboxv (Ihandle **children); [in C]
Iup.zbox(child, ... : ihandle) -> (elem: ihandle) [in Lua]
zbox(child, ...) [in LED]
```

child, ... : List of the elements that will be placed in the box. NULL must be used to define the end of the list in C. It can be empty.

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.

Returns: the identifier of the created element, or NULL 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 "*widthxheight*", 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.

[EXPAND](#): The default value is "YES".

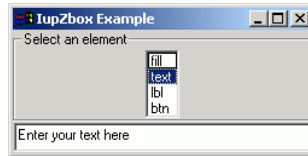
Notes

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

[Browse Example Files](#)



See Also

[IupHbox](#), [IupVBox](#)

- [Creation](#)
- [Attributes](#)
- [Callbacks](#)
- [Notes](#)
- [Examples](#)
- [See Also](#)

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 composition.

Creation

```
Ihandle* IupRadio(Ihandle *child); [in C]
iup.radio{child: ihandle} -> (elem: ihandle) [in Lua]
radio(child) [in LED]
```

child: Identifier of an interface element. Usually it is a vbox or an hbox containing the toggles associated to the radio. It can be NULL.

Returns: the identifier of the created element, or NULL if an error occurs.

Attributes

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

Examples

[Browse Example Files](#)



- [Creation](#)
- [Attributes](#)
- [Callbacks](#)
- [Notes](#)
- [Examples](#)
- [See Also](#)

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]
iup.cbox(elem1, elem2, ...: ihandle) -> (elem: ihandle) [in Lua]
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

[Browse Example Files](#)

See Also

[IupCanvas](#)

- [Creation](#)
- [Attributes](#)
- [Callbacks](#)
- [Notes](#)
- [Examples](#)
- [See Also](#)

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]
iup.sbox(elem: ihandle) -> (elem: ihandle) [in Lua]
sbox(elem) [in LED]
```

elem: This function receives as parameter the element that will be enclosed in a Sbox.

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

Attributes

DIRECTION: Indicates the direction of the resize. Possible values are:

"NORTH", "SOUTH", "EAST", "WEST". Default: "EAST".

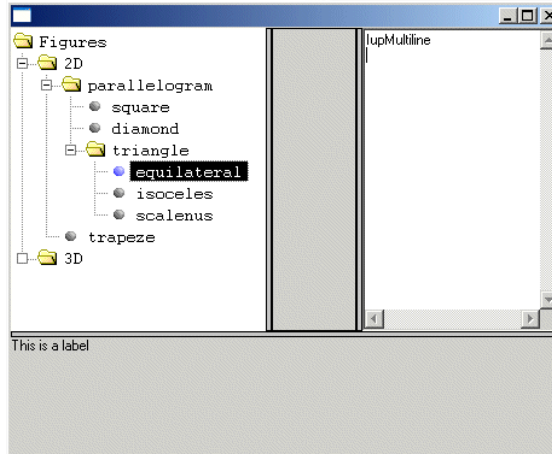
COLOR: Changes the color of the Sboxs thumb. The value should be given in "R G B" color style.

Notes

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

Examples

[Browse Example Files](#)



Example 2 image

IupCreate

Creates an interface element given its class name and parameters. This function is called from all constructors like IupDialog(...), IupLabel(...), and so on.

After **creation** the element still needs to be **attached** to a container and **mapped** to the native system so it can be visible.

Parameters/Return

```
Ihandle* IupCreate(const char *name); [in C]
Ihandle* IupCreatev(const char *name, void **params)
Ihandle *IupCreatep(const char *name, void* params0, ...)
[Not available in Lua]
```

name: class name of the element to be created

params: list of parameters limited by a NULL.

See Also

[IupAppend](#), [IupDetach](#), [IupMap](#), [IupUnmap](#), [IupDestroy](#), [IupGetClassName](#)

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.

It will automatically **unmap** and **detach** the element if necessary.

Parameters/Return

```
void IupDestroy(Ihandle* ih); [in C]
iup.Destroy(ih: ihandle) [in Lua]
or ih:destroy() [in Lua]
```

ih: Identifier of the interface element to be destroyed.

Notes

This function also deletes the names associated to the interface elements being destroyed.

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.

See Also

[IupAppend](#), [IupDetach](#), [IupMap](#), [IupUnmap](#), [IupCreate](#)

IupAppend

Inserts an interface element at the end of the container. Valid for any element that contains other elements like dialog, frame, hbox, vbox, zbox or menu.

It will NOT **map** the element into the native system.

Parameters/Return

```
Ihandle* IupAppend(Ihandle* parent, Ihandle* ih); [in C]
iup.Append(parent, ih: ihandle) -> (parent: ihandle) [in Lua]
```

parent: Identifier of a container like hbox, vbox, zbox and menu.

ih: Identifier of the element to be inserted.

This function returns the actual **parent** if the interface element was successfully inserted. Otherwise returns NULL (nil in Lua). Notice that the desired parent can contains a set of elements and containers where the child will be actually attached so the function returns the actual parent of the element.

Notes

This function can be used when elements that will compose a container are not known *a priori* and should be dynamically constructed.

If the parent is already mapped you must explicitly call **IupMap** for the appended children or for the main container.

The elements are NOT immediately repositioned. Call **IupRefresh** for the container* to update the dialog layout (* or any other element in the dialog).

See Also

[IupDetach](#), [IupHbox](#), [IupVbox](#), [IupZbox](#), [IupMenu](#), [IupMap](#), [IupUnmap](#), [IupRefresh](#)

IupDetach

Detaches an interface element from its parent.

If the child is still **mapped**, then it will be **unmapped**.

Parameters/Return

```
void IupDetach(Ihandle* child); [in C]
iup.Detach(child: ihandle) [in Lua]
or child:detach() [in Lua]
```

child: Identifier of the interface element to be detached.

Notes

This function does not destroy the IUP interface element, just **unmap** (if necessary) and **detach** it from the child list of its parent. If left **detached** it is still necessary to call **IupDestroy** to **destroy** the IUP element.

The elements are NOT immediately repositioned. Call **IupRefresh** for the parent* to update the dialog layout (* or any other element still in the dialog).

See Also

[IupAppend](#), [IupRefresh](#), [IupUnmap](#), [IupCreate](#), [IupDestroy](#)

IupMap

Creates (**maps**) the native interface objects corresponding to the given IUP interface elements.

It will also create the native element of all the children in the element's tree.

The element must be already **attached** to a container if not a dialog.

Parameters/Return

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

ih: Identifier of an interface element.

If the element was already mapped returns IUP_NOERROR.

If the native creation failed returns IUP_ERROR.

Return IUP_NOERROR if successful.

Notes

If the element is a dialog then the abstract layout will be updated even if the element is already mapped. If the dialog is visible the elements will be immediately repositioned.

If you add new elements to an already mapped dialog you must call **IupMap** for that elements. And then call **IupRefresh** to update the dialog layout.

If the WID attribute is NULL, it means the element was not already mapped. Some containers do not have a native element associated, like VBOX and HBOX. In this case their WID is inherited from their parent.

A child is only mapped if its parent is already mapped.

This function is automatically called before dialog show in **IupShow**, **IupShowXY** or **IupPopup**.

It is usefull for the application to call **IupMap** when the value of the WID attribute must be known, or the native element exist, before a dialog is made visible.

The MAP_CB callback is called just after the native element is created and the dialog layout updated, so it can also be used to create other things that depend on the WID attribute.

See Also

[IupAppend](#), [IupDetach](#), [IupUnmap](#), [IupCreate](#), [IupDestroy](#), [IupShowXY](#), [IupShow](#), [IupPopup](#), [MAP_CB](#)

IupUnmap (since 3.0)

Destroys (**unmap**) the native interface objects corresponding to the given IUP interface elements.

It will also destroy the native element of all the children in the element's tree.

It will NOT **destroy** the IUP element, and it will NOT **detach** the element from its parent.

Parameters/Return

```
void IupUnmap(Ihandle* ih); [in C]
iup.Unmap(ih: iuplua-tag) [in Lua]
```

ih: Identifier of an interface element.

See Also

[IupAppend](#), [IupDetach](#), [IupMap](#), [IupCreate](#), [IupDestroy](#)

IupReparent (Since 3.0)

Moves an interface element from one position in the hierarchy tree to another.

Both parent and child must be mapped or unmapped at the same time.

Parameters/Return

```
void IupReparent(Ihandle* child, Ihandle* parent); [in C]
iup.Reparent(parent, ih: ihandle) [in Lua]
```

parent: Identifier of the new parent.

child: Identifier of the element to be moved.

Returns IUP_NOERROR if successfully, IUP_ERROR if failed.

Notes

This function is faster and easier than doing the sequence **unmap**, **detach**, **attach** and **map**.

The elements are NOT immediately repositioned. Call **IupRefresh** for the container* to update the dialog layout (* or any other element in the dialog).

See Also

[IupAppend](#), [IupDetach](#), [IupMap](#), [IupUnmap](#), [IupRefresh](#)

IupGetParent

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

Parameters/Return

```
Ihandle* IupGetParent(Ihandle *ih); [in C]
iup.GetParent(ih: ihandle) -> ret: ihandle [in Lua]
```

ih: identifier of the interface element.

See Also

[IupGetChild](#), [IupGetNextChild](#), [IupGetBrother](#)

IupGetChild

Returns the children of the given control.

Parameters/Return

```
Ihandle *IupGetChild(Ihandle* ih, int pos); [in C]
iup.GetChild(ih, pos: number) -> ret: ihandle [in Lua]
```

ih: identifier of the interface element. pos: position of the desire child.

Notes

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

See Also

[IupGetNextChild](#), [IupGetBrother](#), [IupGetParent](#)

IupGetNextChild

Returns the children of the given control.

Parameters/Return

```
Ihandle *IupGetNextChild(Ihandle* ih, Ihandle* child); [in C]
iup.GetNextChild(ih, child: ihandle) -> ret: ihandle [in Lua]
```

ih: identifier of the interface element.

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

Notes

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 child=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](#), [IupGetParent](#), [IupGetChild](#)

IupGetBrother

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

Parameters/Return

```
Ihandle* IupGetBrother(Ihandle* ih); [in C]
iup.GetBrother(ih: ihandle) -> ih: ihandle [in Lua]
```

ih: identifier of the interface element.

See Also

[IupGetChild](#), [IupGetNextChild](#), [IupGetParent](#)

IupGetDialog

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

Parameters/Return

```
Ihandle* IupGetDialog(Ihandle *ih); [in C]
iup.GetDialog(ih: ihandle) -> (ih: ihandle) [in Lua]
```

ih: Identifier of an interface element.

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

IupRefresh

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

Parameters/Return

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

ih: identifier of the interface element.

Notes

Can be used for any control, but it will always affects the hole dialog.

The elements are immediately repositioned.

This function will NOT change the size of the dialog, except when the SIZE or RASTERSIZE attributes of the dialog where changed before the call.

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

```
IupSetAttribute(dialog, "SIZE", ...);
IupRefresh(dialog);
```

So the dialog will be resized for the new **User** size, if the new size is NULL the dialog will be resized to the **Natural** size that include all the elements.

Changing the size of elements without changing the dialog size may position some controls outside the dialog area at the left or bottom borders (the elements will be cropped at the dialog borders).

IupMap also updates the dialog layout even if it is already mapped, so using it or using IupShow, IupShowXY or IupPopup (they all call IupMap) will also update the dialog layout.

See Also

[SIZE](#), [IupMap](#)

IupUpdate

Only force a redraw of the element and its children.

Parameters/Return

```
void IupUpdate(Ihandle* ih); [in C]
iup.Update(ih: ihandle) [in Lua]
```

ih: identifier of the interface element.

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 or **IupExitLoop** is called.

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

- [Creation](#)
- [Attributes](#)
- [Callbacks](#)
- [Notes](#)
- [Examples](#)
- [See Also](#)

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 *child); [in C]
iup.dialog{child: ihandle} -> (elem: ihandle) [in Lua]
dialog(child) [in LED]
```

child: Identifier of an interface element. The dialog has only one child.

Returns: the identifier of the created element, or NULL if an error occurs.

Attributes

Common

BACKGROUND (non inheritable): Dialog background color or image. Can be a non inheritable alternative to BGCOLOR or can be the name of an image to be tiled on the background. (since 3.0)

BGCOLOR: Dialog background color.

BORDER (non inheritable) (creation only): Shows a resize border around the dialog. Default: "YES".

BORDER=NO is usefull only when RESIZE=NO, MAXBOX=NO, MINBOX=NO, MENUBOX=NO and TITLE=NULL, if any of these are defined there will be always some border.

CURSOR (non inheritable): Defines a cursor for the dialog.

SIZE (non inheritable): Dialog's size.

Additionally the following values can also be defined for width and/or 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)

If SIZE or RASTERSIZE are set changing the user size, then the current size is internally reset to 0x0, so the user size or the natural size can be used when updating the actual current size of the dialog.

TITLE (non inheritable): Dialog's title.

VISIBLE: Simply call **IupShow** or **IupHide** for the dialog.

ACTIVE, **FONT**, **EXPAND**, **X**, **Y**, **WID**, **TIP**, **CLIENTSIZE**, **RASTERSIZE**, **ZORDER**: also accepted. Note that ACTIVE, BGCOLOR, EXPAND and FONT will also affect all the controls inside the dialog.

Exclusive

DEFAULTENTER: Name of the button activated when Enter is hit.

DEFAULTESC: Name of the button activated when Esc is hit.

DIALOGFRAME: Set the common decorations for modal dialogs. This means RESIZE=NO, MINBOX=NO and MAXBOX=NO. In Windows, if the PARENTDIALOG is defined then the MENUBOX is also removed, but the Close button remains.

ICON: Dialog's icon.

FULLSCREEN: Makes the dialog occupy the whole screen.

All dialog details, such as title bar, borders, maximize button, etc, are removed. Possible values: YES, NO. In Motif you may have to click in the dialog to set its focus. In Motif if set to YES when the dialog is hidden, then it can not be changed after it is visible.

MAXBOX (creation only): Requires a maximize button from the window manager. If RESIZE=NO then MAXBOX will be set to NO. Default: YES. See comments about Motif in Notes bellow. In Windows MAXBOX is hidden only if MINBOX is hidden as well, or else it will be just disabled.

MAXSIZE: Maximum size for the dialog in raster units (pixels). The windowing system will not be able to change the size beyond this limit. Default: 65535x65535. (since 3.0)

MENU: Associates a menu to the dialog.

MENUBOX (creation only): Requires a menu box from the window manager. If hidden will also remove the Close button. Default: YES. See comments about Motif in Notes below. In Windows if hidden will hide also MAXBOX and MINBOX.

MINBOX (creation only): Requires a minimize button from the window manager. Default: YES. See comments about Motif in Notes below. In Windows MINBOX is hidden only if MAXBOX is hidden as well, or else it will be just disabled.

MINSIZE: Minimum size for the dialog in raster units (pixels). The windowing system will not be able to change the size beyond this limit. Default: 1x1. (since 3.0)

MODAL (read-only): Returns the popup state. It is "YES" if the dialog was shown using IupPopup. It is "NO" if IupShow was used or it is not visible. (since 3.0)

PARENTDIALOG (creation only): Name of a dialog to be used as parent. This dialog will be always in front of the parent dialog. The parent dialog must be mapped before mapping the child dialog. If PARENTDIALOG is not defined then the NATIVEPARENT attribute is consulted. This one must be a native handle of an existing dialog.

IMPORTANT: When the parent is destroyed the child dialog is also destroyed, then the CLOSE_CB callback of the child dialog is NOT called. The application must take care of destroying the children dialogs before destroying the parent. This is usually done when CLOSE_CB of the parent dialog is called.

PLACEMENT (IupShow only):

Changes how the dialog will be shown. Values: "FULL", "MAXIMIZED", "MINIMIZED" and "NORMAL". Default: NORMAL. After IupShow the attribute is set back to "NORMAL". FULL is similar to FULLSCREEN but only the dialog client area covers the screen area, menu and decorations will be there but out of the screen. Not used in IupPopup. In UNIX there is a chance that the placement won't work correctly, that depends on the Window Manager.

RESIZE (creation only): Allows interactively changing the dialog's size. Default: YES. See comments about Motif in Notes below.

SAVEUNDER [Windows and Motif Only] (creation only):

When this attribute is true (YES), the dialog stores the original image of the desktop region it occupies (if the system 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. To save memory disable it for your main dialog. Not available in GTK (in fact all the controls in GTK are double buffered to speed up redraw).

SHRINK: Allows changing the elements' distribution when the dialog is smaller than the minimum size. Default: NO.

STARTFOCUS: Name of the element that must receive the focus right after the dialog is shown using **IupShow** or **IupPopup**. If not defined then the first control than can receive the focus is selected (same effect of calling [IupNextField](#) for the dialog).

Exclusive [Windows and GTK Only]

DRAGDROP [Windows and GTK Only]: Enable or disable the drag&drop of files. Default: NO, but if DROPPFILES_CB is defined when the element is mapped then it will be automatically enabled.

HIDETASKBAR [Windows and GTK Only] (write-only):

Action attribute that when set to "YES", hides the dialog, but does not decrement the visible dialog count, does not call SHOW_CB and does not mark the dialog as hidden inside IUP. It is usually used to hide the dialog and keep the tray icon working without closing the main loop. It has the same effect as setting LOCKLOOP=YES and normally hiding the dialog. **IMPORTANT**: when you hide using HIDETASKBAR, you must show using HIDETASKBAR also. Possible values: YES, NO.

TRAY [Windows and GTK Only]: When set to "YES", displays an icon on the system tray.

TRAYIMAGE [Windows and GTK Only]: Name of a IUP image to be used as the tray icon.

TRAYTIP [Windows and GTK Only] :Tray icon's tooltip text.

Exclusive [Windows Only]

BRINGFRONT [Windows Only] (write-only):

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

CLIPCHILDREN [Windows Only] (creation only):

When option YES is selected, the area occupied by the children in the dialog is not redrawn, thus preventing the children 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.

COMPOSITED [Windows XP Only] (creation only):

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

CONTROL [Windows Only]: Embeds the dialog inside another window. Creation-only attribute.

HELPBUTTON [Windows Only] (creation 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.

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).

TOOLBOX [Windows Only] (creation only): This attribute makes the dialog look like a toolbar. It is only valid if the PARENTDIALOG or NATIVEPARENT attribute is also defined. Default: NO.

TOPMOST [Windows Only]: This attribute puts the dialog always in front of all other dialogs in all applications. Default: NO.

Exclusive MDI [Windows Only]

MDIACTIVE [Windows Only] (read-only): Returns the the name of the current active MDI child.

MDIACTIVATE [Windows Only] (write-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.

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

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 sequential 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.

MDICLOSEALL [Windows Only] (write-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.

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.

MDINEXT [Windows Only] (read-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 retrieve the next child before destroying the current.

Callbacks

[CLOSE_CB](#): Called right before the dialog is closed.

[DROPFILES_CB](#) [Windows and GTK Only]: Action generated when one or more files are dropped from the Windows Explorer or from Nautilus in 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 *ih); [in C]
elem:mdiactivate_cb() -> (ret: number) [in Lua]
```

ih: identifier of the element that activated the event.

[RESIZE_CB](#): Action generated when the dialog size is changed. (since 3.0)

[SHOW_CB](#): Called right after the dialog is opened, minimized or restored from a minimization.

TRAYCLICK_CB [Windows and GTK Only]: Called right after the mouse button is pressed or released over the tray icon.

```
int function(Ihandle *ih, int but, int pressed, int dclick); [in C]
elem:trayclick_cb(but, pressed, dclick: number) -> (ret: number) [in Lua]
```

ih: identifier of the element that activated the event.

but: identifies the activated mouse button. Can be: 1, 2 or 3. Note that this is different from the BUTTON_CB canvas callback definition. GTK does not get button=2 messages.

pressed: indicates the state of the button. Always 1 in GTK.

dclick: indicates a double click. In GTK double click is simulated.

Return: IUP_CLOSE will be processed.

[MAP_CB](#), [GETFOCUS_CB](#), [KILLFOCUS_CB](#), [ENTERWINDOW_CB](#), [LEAVEWINDOW_CB](#), [K_ANY](#), [HELP_CB](#): All common callbacks are supported.

Notes

Do not associate an IupDialog with the native "dialog" nomenclature in Windows, GTK or Motif. IupDialog use native standard windows in all drivers.

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 if its dialog is also visible.

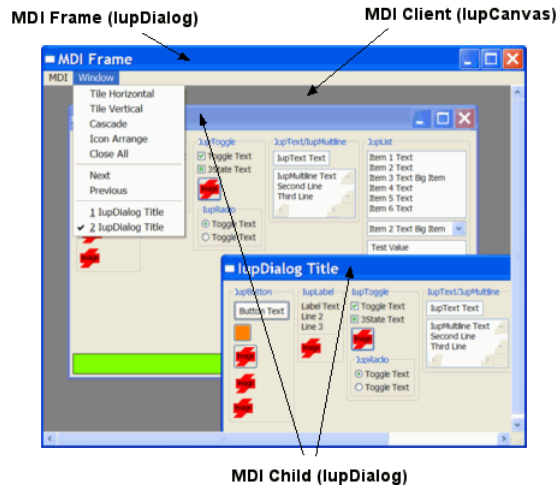
Values attributed to the SIZE attribute of a dialog are always accepted, regardless of the minimum size required by its children. For a dialog to have the minimum necessary size to fit all elements contained in it, simply define NULL (in C) to SIZE. In the case of partial dimensions, a specified dimension is always used, while a non-defined dimension uses the smallest necessary size for the elements in the corresponding direction. See the SIZE attribute for further details.

The order of callback calling is system dependent. For instance, the RESIZE_CB and the SHOW_CB are called in different order in Win32 and in X-Windows when the dialog is shown for the first time.

In Motif the decorations ICON, MENUBOX, MINBOX, MAXBOX, RESIZE and BORDER are controled by the Window Manager and may not be possible to be changed from IUP.

Windows MDI

The MDI support is composed of 3 components: the MDI frame window (IupDialog), the MDI client window (IupCanvas) and the MDI children (IupDialog). Although 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

Very simple dialog with a label and a button. The application is closed when the button is pressed.

```
#include <iup.h>

int quit_cb(void)
{
    return IUP_CLOSE;
}

int main(int argc, char* argv[])
{
    Ihandle *dialog, *quit_bt, *vbox;

    IupOpen(&argc, &argv);

    /* Creating the button */
    quit_bt = IupButton("Quit", 0);
    IupSetCallback(quit_bt, "ACTION", (Icallback)quit_cb);

    /* the container with a label and the button */
    vbox = IupVbox(
        IupSetAttributes(IupLabel("Very Long Text Label"), "EXPAND=YES, ALIGNMENT=ACENTER"),
        quit_bt,
        0);
    IupSetAttribute(vbox, "MARGIN", "10x10");
    IupSetAttribute(vbox, "GAP", "5");

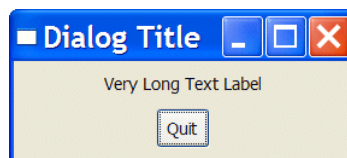
    /* Creating the dialog */
    dialog = IupDialog(vbox);
    IupSetAttribute(dialog, "TITLE", "Dialog Title");
    IupSetAttributeHandle(dialog, "DEFAULTESC", quit_bt);

    IupShow(dialog);

    IupMainLoop();

    IupDestroy(dialog);
    IupClose();

    return 0;
}
```



Other: [Browse Example Files](#)

See Also

[IupFileDialog](#), [IupMessageDlg](#), [IupDestroy](#), [IupShowXY](#), [IupShow](#), [IupPopup](#)

IupPopup

Shows a dialog or menu and restricts user interaction only to the specified element. It is equivalent of creating a Modal dialog in some toolkits.

If another dialog is shown after **IupPopup** using **IupShow**, then its interaction will not be inhibited. Every **IupPopup** call creates a new popup level that inhibits all previous dialogs interactions, but does not disable new ones. IMPORTANT: The popup levels must be closed in the reverse order they were created or unpredictable results will occur.

For a dialog this function will only return the control to the application after a callback returns IUP_CLOSE, **IupExitLoop** is called, or when the popup dialog is hidden, for example using **IupHide**. For a menu it returns automatically after a menu item is selected. IMPORTANT: If a menu item callback returns IUP_CLOSE, it will end the current popup level dialog.

Parameters/Return

```
int IupPopup(Ihandle *ih, int x, int y); [in C]
iup.Popup(ih: ihandle[, x, y: number]) -> (ret: number) [in Lua]
or ih:popup([x, y: number]) -> (ret: number) [in Lua]
```

ih: 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
- IUP_CENTERPARENT: Horizontally centralizes the dialog relative to its parent (Since 3.0)
- IUP_CURRENT: use the current position of the dialog (if used at the first time the dialog is shown, then it is replaced by IUP_CENTERPARENT if PARENTDIALOG is defined or simply by IUP_CENTER). This is the default value in Lua if the parameter is not defined or nil. (Since 3.0)

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
- IUP_CENTERPARENT: Vertically centralizes the dialog relative to its parent (Since 3.0)
- IUP_CURRENT: use the current position of the dialog (if used at the first time the dialog is shown, then it is replaced by IUP_CENTERPARENT if PARENTDIALOG is defined or simply by IUP_CENTER). This is the default value in Lua if the parameter is not defined or nil. (Since 3.0)

This function returns IUP_NOERROR if successful. Returns IUP_INVALID if not a dialog or menu. If there was an error returns IUP_ERROR..

Notes

Will call **IupMap** for the element.

See the [PLACEMENT](#) attribute for other position and show options.

IupPopup works just like **IupShow** and **IupShowXY**, but it inhibits interaction with other dialogs and interrupts the processing until IUP_CLOSE is returned in a callback or the dialog is hidden. Although it interrupts the processing, it does not destroy the dialog when it ends. To destroy the dialog, **IupDestroy** must be called.

This function can be executed more than once for the same dialog. IMPORTANT: Calling **IupPopup** for an already visible dialog will only update its position and/or size on screen, will NOT change its modal state and will NOT interrupt processing.

In GTK and Motif the inactive dialogs will still be able to move, resize and change their Z-order. Although their contents will be inactive, keyboard will be disabled, and they can not be closed from the close box.

See Also

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

IupShow

Displays a dialog in the current position, or changes a control **VISIBLE** attribute. If the dialog needs to be mapped and the current position is not known then the dialog is centered.

For a dialog to set the attribute **VISIBLE=YES** is the same as calling **IupShow**. For other controls, to call **IupShow** is the same as setting **VISIBLE=YES**.

Parameters/Return

```
int IupShow(Ihandle *ih); [in C]
iup.Show(ih: ihandle) -> (ret: number) [in Lua]
or ih:show() -> (ret: number) [in IupLua]
```

ih: identifier of the interface element.

This function returns IUP_NOERROR if successful. If there was an error returns IUP_ERROR.

Notes

For dialogs it is equivalent to **IupShowXY** using **IUP_CURRENT** (or **IUP_CENTER** if not mapped).

Will call **IupMap** for the element.

See the **IupDialog** [PLACEMENT](#) attribute for other position and show options.

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, changing its Z-order, and update its position and/or size on screen. IMPORTANT: Calling **IupShow** for a visible dialog shown with **IupPopup** does nothing.

See Also

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

IupShowXY

Displays a dialog in a given position on the screen.

Parameters/Return

```
int IupShowXY(Ihandle *ih, int x, int y); [in C]
iup.ShowXY(ih: ihandle[, x, y: number]) -> (ret: number) [in Lua]
or ih:showxy([x, y: number]) -> (ret: number) [in Lua]
```

ih: 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
- IUP_CENTERPARENT: Horizontally centralizes the dialog relative to its parent (Since 3.0)
- IUP_CURRENT: use the current position of the dialog (if used at the first time the dialog is shown, then it is replaced by IUP_CENTERPARENT if PARENTDIALOG is defined or simply by IUP_CENTER). This is the default value in Lua if the parameter is not defined or nil. (Since 3.0)

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
- IUP_CENTERPARENT: Vertically centralizes the dialog relative to its parent (Since 3.0)
- IUP_CURRENT: use the current position of the dialog (if used at the first time the dialog is shown, then it is replaced by IUP_CENTERPARENT if PARENTDIALOG is defined or simply by IUP_CENTER). This is the default value in Lua if the parameter is not defined or nil. (Since 3.0)

This function returns IUP_NOERROR if successful. Returns IUP_INVALID if not a dialog. If there was an error returns IUP_ERROR.

Notes

Will call **IupMap** for the element.

See the [PLACEMENT](#) attribute for other position and show options.

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, changing its Z-order, and update its position and/or size on screen. IMPORTANT: Calling **IupShow** for a visible dialog shown with **IupPopup** does nothing.

See Also

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

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 *ih); [in C]
iup.Hide(ih: ihandle) -> (ret: number) [in Lua]
or ih:hide() -> (ret: number) [in Lua]
```

ih: Identifier of the interface element.

This function returns IUP_NOERROR always.

Notes

Once a dialog is hidden, either by means of **IupHide** or by changing the **VISIBLE** attribute or by means of a click in the window close button, the elements inside 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](#).

- [Creation](#)
- [Attributes](#)
- [Callbacks](#)
- [Notes](#)
- [Examples](#)
- [See Also](#)

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]
iup.filedlg() -> (elem: ihandle) [in Lua]
filedlg() [in LED]
```

Returns the identifier of the created dialog, or NULL if an error occurs.

Attributes

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". Not used when **DIALOGTYPE**=DIR.

DIALOGTYPE: Type of dialog (Open, Save or Directory). Can have values "OPEN", "SAVE" or "DIR". Default: "OPEN".

In Windows, when **DIALOGTYPE**=DIR the dialog shown is not the same dialog for OPEN and SAVE, this dialog does not have the Help button neither

filters. In Motif or GTK the dialog is the same, but it only allows the user to select a directory.

DIRECTORY: Initial directory.

In Motif or GTK, if not defined the dialog opens in the current directory.

In Windows, if not defined and the application has used the dialog in the past, the path most recently used is selected as the initial directory. However, if an application is not run for a long time, its saved selected path is discarded. Also if not defined and the current directory contains any files of the specified filter types, the initial directory is the current directory. Otherwise, the initial directory is the "My Documents" directory of the current user. Otherwise, the initial directory is the Desktop folder.

EXTFILTER: (Windows and GTK Only) Defines several file filters. It has priority over **FILTERINFO** and **FILTER**. Must be a text with the format "FilterInfo1|Filter1|FilterInfo2|Filter2|...". The list ends with character '|'. Example: "Text files!*.txt;*.doc|Image files!*.gif;*.jpg;*.bmp!". In GTK there is no way how to overwrite the filters, so it is recommended to always add a less restrictive filter to the filter list.

FILE: Name of the file initially shown in the "File Name" field in the dialog. If contains a path, then it is used as the initial directory and **DIRECTORY** is ignored.

FILEEXIST (out): 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. Not set when **DIALOGTYPE=DIR** or **MULTIPLEFILES=YES**.

FILTER: String containing a list of file filters separated by ',' without spaces. Example: "*.C;*.LED;teste.*". In Motif only the first filter is used.

FILTERINFO: (Windows and GTK Only) Filter's description. If not defined the filter itself will be used as its description.

FILTERUSED (in/out): (Windows and GTK Only) the index of the filter in **EXTFILTER** to use starting at 1. It returns the selection made by the user. Set only if **EXTFILTER** is defined.

MULTIPLEFILES: (Windows and GTK 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 '|'. When the user selects just one file, the directory and the file are not separated by '|'. For example:

"/tecgraf/iup/testla.txtlb.txtlc.txtl" or
"/tecgraf/iup/test/a.txt" (only one file is selected)

In Windows the maximum size allowed for file name returned is 65Kb.

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

NOOVERWRITEPROMPT: do not prompt to overwrite an existant file when in "SAVE" dialog. Default is "NO", i.e. prompt before overwrite.

PARENTDIALOG: Makes the dialog be treated as a child of the specified dialog.

PREVIEWDC, **PREVIEWWIDTH** and **PREVIEWHEIGHT:** Read only attributes that are valid inside the **FILE_CB** callback when status="PAINT". Return the Device Context (HDC in Windows and GC in UNIX), the width and the height of the client rectangle for the preview area. Also the attributes **WID**, **HWND**, **XWINDOW** and **XDISPLAY** are valid and are relative to the preview area. (in Motif since 3.0)

SHOWHIDDEN: Show hidden files. Default: NO. (since 3.0)

SHOWPREVIEW: A preview area is shown inside the File Dialog. Can have values "YES" or "NO". Default: "NO". When this attribute is set you must set the "FILE_CB" callback to retrieve the file name and the necessary attributes to paint the preview area. In Windows, you must link with the "iup.rc" resource file so the preview area can be enabled (not necessary if using "iup.dll"). Also the **FILE_CB** callback must be defined.

STATUS (out): Indicates the status of the selection made:

- "1": New file.
- "0": Normal, existing file or directory.
- "-1": Operation cancelled.

TITLE: Dialog's title.

VALUE (out): Name of the selected file(s), or NULL if no file was selected. If **FILE** is not defined this is used as the initial value. In Windows there is a limit of 32Kb for this string.

Callbacks

FILE_CB: Action generated when a file is selected. Not called when **DIALOGTYPE=DIR**. When **MULTIPLEFILES=YES** it is called only for one file. Can be used with **SHOWPREVIEW=NO** also.

```
int function(Ihandle *ih, const char* file_name, const char* status); [in C]
elem:file_cb(file_name, status: string) -> (ret: number) [in Lua]
```

ih: identifier of the element that activated the event.

file_name: name of the file selected.

status: describes the action. Can be:

- "INIT" - when the dialog has started, file_name is NULL.
- "FINISH" - when the dialog is closed, file_name is NULL.
- "INCLUDE" - before a file or folder is included in the list. If returns **IUP_IGNORE** the item is not included. (Windows Only)
- "SELECT" - a file has been selected.
- "OK" - the user pressed the OK button. If returns **IUP_IGNORE** the action is refused and the dialog is not closed.
- "PAINT" - the preview area must be repainted. Used only when **SHOWPREVIEW=YES**. If an invalid file or directory is selected, file_name is NULL.

HELP_CB: Action generated when the Help button is pressed.

Notes

The **IupFileDlg** is a native pre-defined dialog that is not altered by [IupSetLanguage](#).

To show the dialog, use function [IupPopup](#). In Lua, use the popup function.

The dialog is mapped only inside **IupPopup**. **IupMap** does nothing.

The [IupGetFile](#) function simply creates and popup a IupFileDialog.

The FILE and the DIRECTORY attributes in Windows also accept string containing "/" as path separators. But the VALUE attribute will always return strings using the "\" character.

Examples

```
Ihandle *dlg = IupFileDialog();

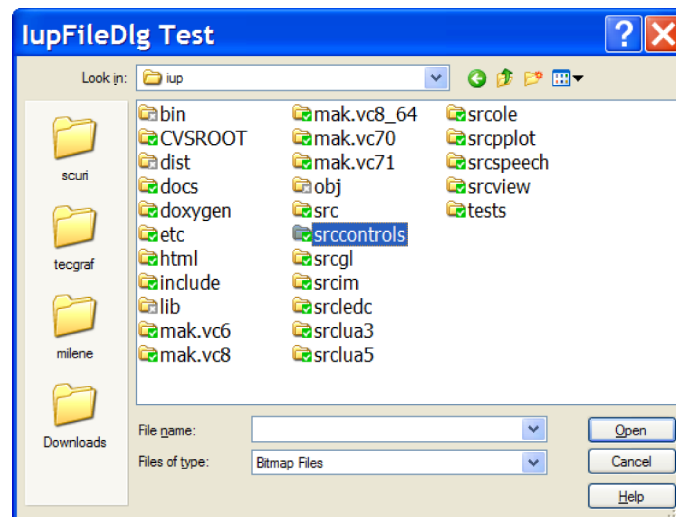
IupSetAttribute(dlg, "DIALOGTYPE", "OPEN");
IupSetAttribute(dlg, "TITLE", "IupFileDialog Test");
IupSetAttributes(dlg, "FILTER = \".*.bmp\"", FILTERINFO = \"Bitmap Files\");
IupSetCallback(dlg, "HELP_CB", (Icallback)help_cb);

IupPopup(dlg, IUP_CURRENT, IUP_CURRENT);

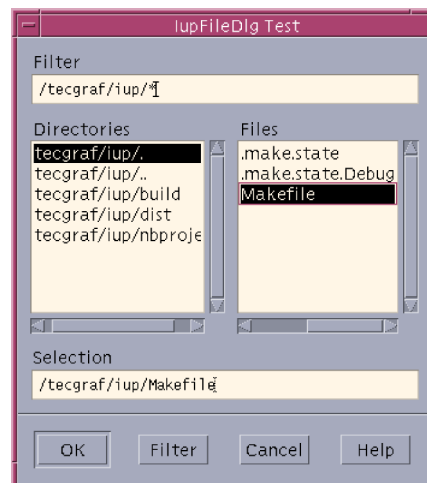
if (IupGetInt(dlg, "STATUS"))
{
    printf("OK\n");
    printf("  VALUE(%s)\n", IupGetAttribute(dlg, "VALUE"));
}
else
    printf("CANCEL\n");

IupDestroy(dlg);
```

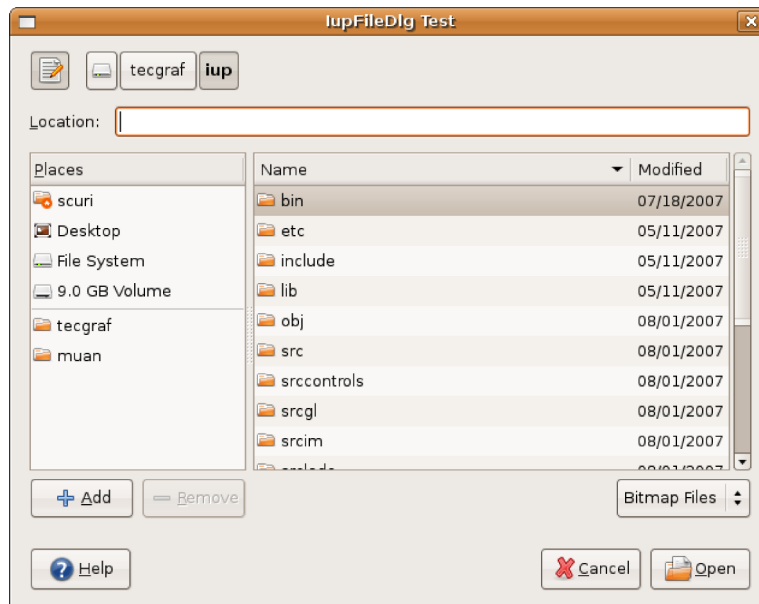
Windows XP



Motif/Mwm



GTK/GNOME



Other: [Browse Example Files](#)

See Also

[IupMessage](#), [IupScanf](#), [IupListDialog](#), [IupAlarm](#), [IupGetFile](#), [IupPopup](#)

- [Creation](#)
- [Attributes](#)
- [Callbacks](#)
- [Notes](#)
- [Examples](#)
- [See Also](#)

IupMessageDlg (since 3.0)

Creates the Message Dialog element. It is a predefined dialog for displaying a message. The dialog can be shown with the IupPopup function only.

Creation

```
Ihandle* IupMessageDlg(void); [in C]
iup.messagedlg() -> (elem: ihandle) [in Lua]
messagedlg() [in LED]
```

Returns: the identifier of the created element, or NULL if an error occurs.

Attributes

BUTTONDEFAULT: Number of the default button. Can be "1" or "2". "2" is valid only for "OKCANCEL" and "YESNO" button configurations.

BUTTONRESPONSE: Number of the pressed button. Can be "1" or "2".

BUTTONS: Buttons configuration.

Can have values: "OK", "OKCANCEL" or "YESNO". Default: "OK".

Additionally the "Help" button is displayed if the HELP_CB callback is defined.

DIALOGTYPE: Type of dialog defines which icon will be displayed besides the message text.

Can have values: "MESSAGE" (No Icon), "ERROR" (Stop-sign), "WARNING" (Exclamation-point), "QUESTION" (Question-mark) or "INFORMATION" (Letter "i"). Default: "MESSAGE".

PARENTDIALOG: (Creation Only) Name of a dialog to be used as parent. This dialog will be always in front of the parent dialog. If not defined in Motif the dialog could not be modal.

TITLE: Dialog title.

VALUE: Message text.

Callbacks

HELP_CB: Action generated when the Help button is pressed.

Notes

The **IupMessageDlg** is a native pre-defined dialog not altered by **IupSetLanguage**.

To show the dialog, use function **IupPopup**. In Lua, use the **popup** function.

The dialog is mapped only inside **IupPopup**, **IupMap** does nothing.

In Windows the position (x,y) used in **IupPopup** is ignored and the dialog is always centered on screen.

The **IupMessage** function simply creates and popup a **IupMessageDlg**.

In Windows each different dialog type is always associated with a different beep sound.

Examples

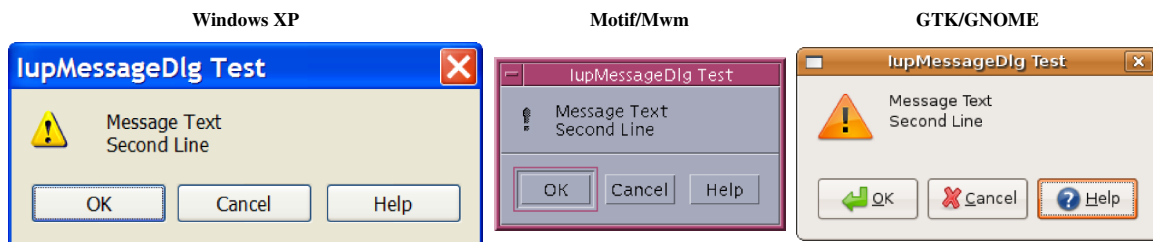
```
Ihandle* dlg = IupMessageDlg();

IupSetAttribute(dlg, "DIALOGTYPE", "WARNING");
IupSetAttribute(dlg, "TITLE", "IupMessageDlg Test");
IupSetAttribute(dlg, "BUTTONS", "OKCANCEL");
IupSetAttribute(dlg, "VALUE", "Message Text\nSecond Line");
IupSetCallback(dlg, "HELP_CB", (Icallback)help_cb);

IupPopup(dlg, IUP_CURRENT, IUP_CURRENT);

printf("BUTTONRESPONSE(%s)\n", IupGetAttribute(dlg, "BUTTONRESPONSE"));

IupDestroy(dlg);
```



See Also

[IupMessage](#), [IupScanf](#), [IupListDialog](#), [IupAlarm](#), [IupGetFile](#), [IupPopup](#)

- [Creation](#)
- [Attributes](#)
- [Callbacks](#)
- [Notes](#)
- [Examples](#)
- [See Also](#)

IupColorDlg (Windows and GTK only) (since 3.0)

Creates the Color Dialog element. It is a predefined dialog for selecting a color. The dialog can be shown with the **IupPopup** function only.

Creation

```
Ihandle* IupColorDlg(void); [in C]
iup.colordlg() -> (elem: ihandle) [in Lua]
colordlg() [in LED]
```

Returns: the identifier of the created element, or NULL if an error occurs.

Attributes

ALPHA (GTK only) (in/out): if defined enable an alpha selection additional control with its initial value. If the user pressed the Ok button contains the returned value.

COLORTABLE (in/out): list of colors separated by ";". In GTK accepts 20 values and if not present the palette will not be visible. In Windows accepts 16 values and will be always visible, even if the colors are not defined (in this case are initialized with black).

PARENTDIALOG: (Creation Only) Name of a dialog to be used as parent. This dialog will be always in front of the parent dialog.

STATUS (out): defined to "1" if the user pressed the Ok button, NULL if pressed the Cancel button.

TITLE: Dialog title.

VALUE (in/out): The initial color value and the returned selected value if the user pressed the Ok button.

Callbacks

HELP_CB: Action generated when the Help button is pressed.

Notes

The **IupColorDlg** is a native pre-defined dialog not altered by **IupSetLanguage**.

To show the dialog, use function **IupPopup**. In Lua, use the **popup** function.

The dialog is mapped only inside **IupPopup**, **IupMap** does nothing.

Examples

```
Ihandle* dlg = IupColorDlg();
```



```

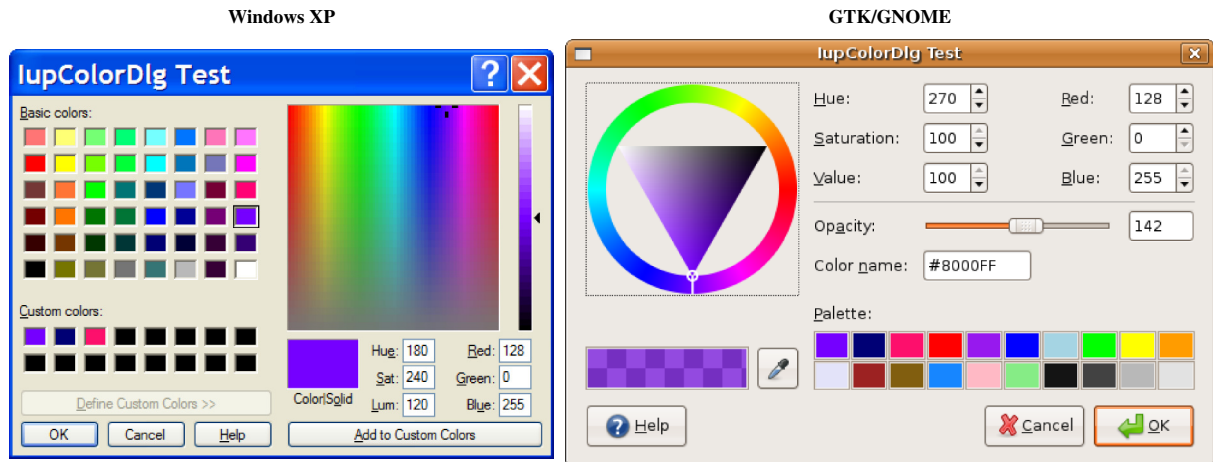
IupSetAttribute(dlg, "VALUE", "128 0 255");
IupSetAttribute(dlg, "ALPHA", "142");
IupSetAttribute(dlg, "COLORTABLE", "128 0 255;0 0 128;253 20 119");
IupSetAttribute(dlg, "TITLE", "IupColorDlg Test");
IupSetCallback(dlg, "HELP_CB", (Icallback)help_cb);

IupPopup(dlg, IUP_CURRENT, IUP_CURRENT);

if (IupGetInt(dlg, "STATUS"))
{
    printf("OK\n");
    printf("  COLOR(%s)\n", IupGetAttribute(dlg, "COLOR"));
    printf("  COLORTABLE(%s)\n", IupGetAttribute(dlg, "COLORTABLE"));
}
else
    printf("CANCEL\n");

IupDestroy(dlg);

```



See Also

[IupMessageDlg](#), [IupFileDialog](#), [IupPopup](#)

- [Creation](#)
- [Attributes](#)
- [Callbacks](#)
- [Notes](#)
- [Examples](#)
- [See Also](#)

IupFontDlg (Windows and GTK only) (since 3.0)

Creates the Font Dialog element. It is a predefined dialog for selecting a font. The dialog can be shown with the `IupPopup` function only.

Creation

```

Ihandle* IupFontDlg(void); [in C]
iup.fontdlg() -> (elem: ihandle) [in Lua]
fontdlg() [in LED]

```

Returns: the identifier of the created element, or NULL if an error occurs.

Attributes

PREVIEWTEXT (GTK only): the text shown in the preview area. If not defined GTK will provide a default text.

COLOR (Windows Only) (in/out): The initial color value and the returned selected value if the user pressed the Ok button. In Windows the Choose Font dialog allows the user to select a color from a pre-defined list of colors.

PARENTDIALOG: (Creation Only) Name of a dialog to be used as parent. This dialog will be always in front of the parent dialog.

STATUS (out): defined to "1" if the user pressed the Ok button, NULL if pressed the Cancel button.

TITLE: Dialog title.

VALUE (in/out): The initial font value and the returned selected value if the user pressed the Ok button. Has the same format of the [FONT](#) attribute.

Callbacks

HELP_CB: Action generated when the Help button is pressed.

Notes

The **IupFontDlg** is a native pre-defined dialog not altered by **IupSetLanguage**.

To show the dialog, use function **IupPopup**. In Lua, use the **popup** function.

The dialog is mapped only inside **IupPopup**, **IupMap** does nothing.

Examples

```
Ihandle* dlg = IupFontDlg();

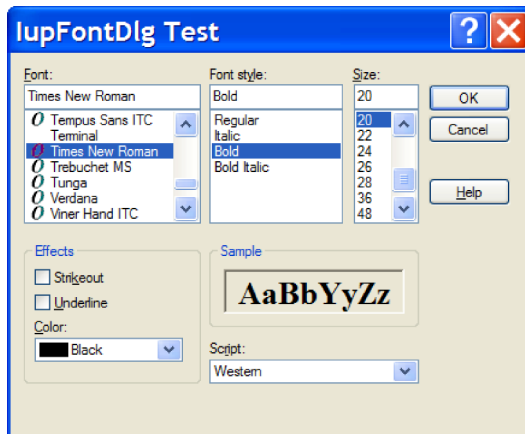
IupSetAttribute(dlg, "COLOR", "128 0 255");
IupSetAttribute(dlg, "VALUE", "Times New Roman, Bold 20");
IupSetAttribute(dlg, "TITLE", "IupFontDlg Test");
IupSetCallback(dlg, "HELP_CB", (Icallback)help_cb);

IupPopup(dlg, IUP_CURRENT, IUP_CURRENT);

if (IupGetInt(dlg, "STATUS"))
{
    printf("OK\n");
    printf("  VALUE(%s)\n", IupGetAttribute(dlg, "VALUE"));
    printf("  COLOR(%s)\n", IupGetAttribute(dlg, "COLOR"));
}
else
    printf("CANCEL\n");

IupDestroy(dlg);
```

Windows XP



GTK/GNOME



See Also

[IupMessageDlg](#), [IupFileDialog](#), [IupPopup](#)

IupAlarm

Shows a modal dialog containing a message and up to three buttons.

Creation and Show

```
int IupAlarm(const char *t, const char *m, const char *b1, const char *b2, const char *b3); [in C]
iup.Alarm(t, m, b1[, b2, b3]: string) -> (button: number) [in Lua]
```

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

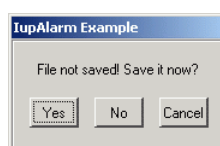
This function shows a dialog centralized on the screen, with the message and the buttons. The '\n' character can be added to the message to indicate line change.

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



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 *filename); [in C]
Iup.GetFile(filename: string) -> (filename: string, status: number) [in Lua]
```

filename: 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.

status: The function returns a 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

Notes

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(const char* title, char *text); [in C]
Iup.GetText(title, text: string) -> (text: string) [in Lua]
```

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 successful. In Lua if an error occurred returns nil.

Notes

The dialog uses a global attribute called "PARENTDIALOG" as the parent dialog if it is defined. 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, const char *title, int size, const char *list[], int option, int max_col, int max_lin, int mark[]); [in C]
Iup.ListDialog(type: number, title: string, size: number, list: table of strings, option: number, max_col: number, max_lin: number, mark: table of numbers) [in Lua]
```

type: =1 simple selection; =2 multiple selection

title: Text for the dialog's title

size: Number of options

list: List of options

option: Initial option, starting at 1 (note that this index is different from the return value, kept for compability reasons)

max_col: Maximum number of columns in the list

max_lin: Maximum number of lines in the list

mark: Flag vector, used only when type=2

When type=1, the function returns the number of the selected option (the first option is 0), or -1 if the user cancels the operation.

When type=2, the function returns -1 when the user cancels the operation. If the user does not cancel the operation the function returns a non zero value and the mark parameter will have value 1 for the options selected by the user and value 0 for non-selected options.

Notes

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

[Browse for Example Files](#)

See Also

[IupMessage](#), [IupScanf](#), [IupGetFile](#), [IupAlarm](#)

IupMessage

Shows a modal dialog containing a message. It simply creates and popup a **IupMessageDlg**.

Creation and Show

```
void IupMessage(const char *title, const char *message); [in C]
iup.Message(title: string, message: string) [in Lua]
```

Notes

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(const char *title, const char *format, ...); [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](#), [IupMessageDlg](#)

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(const char *format, ...); [in C]
iup.Scanf(format: string, ...) -> (n: number, ...) [in Lua]
```

format: 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 (d,i,o,u,x,X,e,f,g,E,G,s, and the modifiers l,h)

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.

[Browse for Example Files](#)

See Also

[IupGetFile](#), [IupMessage](#), [IupListDialog](#), [IupAlarm](#), [IupGetParam](#)

- [Creation](#)
- [Attributes](#)
- [Callbacks](#)
- [Notes](#)
- [Examples](#)

- [See Also](#)

IupGetColor

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

This dialog is included in the [Controls Library](#).

Creation and Show

```
int IupGetColor(int x, int y, unsigned char *r, unsigned char *g, unsigned char *b); [in C]
iup.GetColor(x, y, r, g, b: number) -> (r, g, b: number) [in Lua]
```

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 in C, in Lua they are optional and used for initialization only.

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

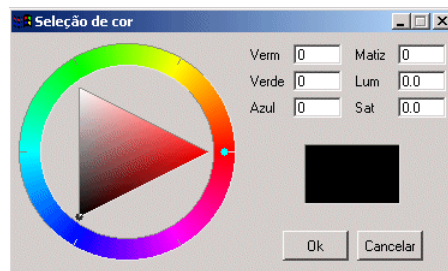
Notes

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

[Browse Example Files](#)



See Also

[IupMessage](#), [IupScanf](#), [IupListDialog](#), [IupAlarm](#), [IupGetFile](#).

- [Creation](#)
- [Attributes](#)
- [Callbacks](#)
- [Notes](#)
- [Examples](#)
- [See Also](#)

IupGetParam

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

This dialog is included in the [Controls Library](#).

Creation and Show

```
int IupGetParam(const char* title, Iparamcb action, void* user_data, const char* format,...); [in C]
iup.GetParam(title: string, action: function, format: string,...) -> (ret: number, ...) [in Lua]
```

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.

Callbacks

```
typedef int (*Iparamcb)(Ihandle* dialog, int param_index, void* user_data);
```

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

dialog: dialog handle

param_index: current parameter being changed. It is -1 if the user pressed the **OK** button. It is -2 when the dialog is **mapped**, just before shown. It is -3 if the user pressed the **Cancel** button.

user_data: a user pointer that is passed in the function call.

You can reject the change or the OK action by returning "0" in the callback, otherwise you must return "1".

You should not programmatically change the current parameter value during the callback. On the other hand you can freely change the value of other parameters.

Use the dialog attribute "PARAMn" to get the parameter "Ihandle*", but not that this is not the actual control. Where "n" is the parameter index in the order they are specified starting at 0, but separators are not counted. Use the parameter attribute "CONTROL" to get the actual control. For example:

```
Ihandle* param2 = (Ihandle*)IupGetAttribute(dialog, "PARAM2");
int value2 = IupGetInt(param2, IUP_VALUE);

Ihandle* param5 = (Ihandle*)IupGetAttribute(dialog, "PARAM5");
Ihandle* ctrl5 = (Ihandle*)IupGetAttribute(param5, "CONTROL");

if (value2 == 0)
{
    IupSetAttribute(param5, IUP_VALUE, "New Value");
    IupSetAttribute(ctrl5, IUP_VALUE, "New Value");
}
```

Since parameters are user controls and not real controls, you must update the control value and the parameter value.

Be aware that programmatically changes are not filtered. The valuator, when available, can be retrieved using the parameter attribute "AUXCONTROL". The valuator is not automatically updated when the text box is changed programmatically. The parameter label is also available using the parameter attribute "LABEL".

Attributes (inside the callback)

For the dialog:

"PARAMn" - returns an IUP Ihandle* representing the nth parameter, indexed by the declaration order not counting separators.

"OK" - returns an IUP Ihandle*, the main button.

"CANCEL" - returns an IUP Ihandle*, the close button.

For a parameter:

"LABEL" - returns an IUP Ihandle*, the label associated with the parameter.

"CONTROL" - returns an IUP Ihandle*, the real control associated with the parameter.

"AUXCONTROL" - returns an IUP Ihandle*, the auxiliary control associated with the parameter (only for Valuator).

"INDEX" - returns an integer value associated with the parameter index. **IupGetInt** can also be used.

"VALUE" - returns the parameter value as a string, but **IupGetFloat** and **IupGetInt** can also be used.

In Lua to retrieve 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 Lua]
```

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)
- f** = string (same as s, but also show a button to open a file selection dialog box)

extra is one or more additional options for the given type

[**min,max,step**] are optional limits for **integer** and **real** types. The maximum and step values can be omitted. When **min** and **max** are specified a valuator will also be added to change the value. To specify **step**, **max** must be also specified. **step** is the size of the increment.

[**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 not use the brackets '[' and ']' to avoid confusion with the specified mask.

item0|item1|item2,...| are the items of the **list**. At least one item must exist. Again the brackets are not used to increase the possibilities for the strings, instead you must use '|'. Items index are zero based start.

[**dialogtype,filter,directory,nochangedir,nooverwriteprompt**] are the respective attribute values passed to the [IupFileDialog](#) control when activated. All commas must exist, but you can let empty values to use the default values.

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.

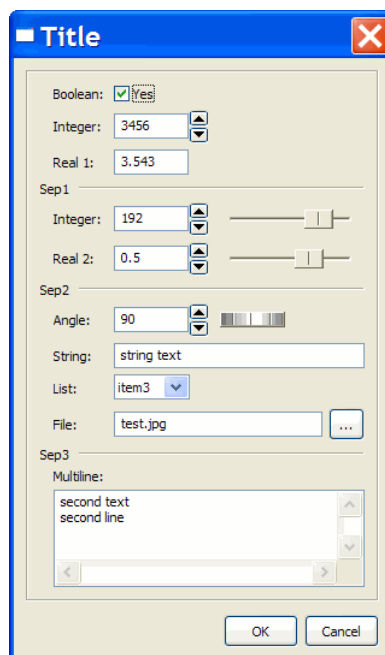
Examples

[Browse Example Files](#)

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.

```
// initial values
int pboolean = 1;
int pinteger = 3456;
float preal = 3.543f;
int pinteger2 = 192;
float preal2 = 0.5f;
float pangle = 90;
char pstring[100] = "string text";
int plist = 2;
char pstring2[200] = "second text\nsecond line";
char file_name[500] = "test.jpg";

if (!IupGetParam("Title", NULL, 0,
    "Boolean: %b[No,Yes]\n"
    "Integer: %i\n"
    "Real 1: %r\n"
    "Sep1 %t\n"
    "Integer: %i[0,255]\n"
    "Real 2: %r[-1.5,1.5,0.05]\n"
    "Sep2 %t\n"
    "Angle: %a[0,360]\n"
    "String: %s\n"
    "List: %l|item1|item2|item3|\n"
    "File: %f[OPEN|*.bmp;*.jpg|CURRENT|NO|NO]\n"
    "Sep3 %t\n"
    "Multiline: %m\n",
    &pboolean, &pinteger, &preal, &pinteger2, &preal2, &pangle, pstring, &plist, file_name, pstring2, NULL))
    return IUP_DEFAULT;
```



See Also

[IupScanf](#), [IupGetColor](#), [IupMask](#), [IupValuator](#), [IupDialog](#), [IupList](#), [IupFileDialog](#).

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 powerful approach is one of the advantages of using IUP.

Controls are automatically destroyed when the dialog is destroyed.

- [Creation](#)
- [Attributes](#)
- [Callbacks](#)
- [Notes](#)
- [Examples](#)
- [See Also](#)

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(const char *title, const char *action); [in C]
iup.button{[title = title: string]} -> elem: ihandle [in Lua]
button(title, action) [in LED]
```

title: Text to be shown to the user. It can be NULL. It will set the TITLE attribute.
action: Name of the action generated when the button is selected. It can be NULL.

Returns: the identifier of the created element, or NULL if an error occurs.

Attributes

BGCOLOR: Background color of the text.

FGCOLOR: Text color.

FONT: Character font of the text.

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

IMPRESS: Image name of the pressed button.

IMINACTIVE: Image name 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 of buttons with IMAGE defined.

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.

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.

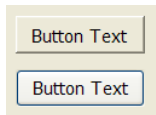
Text and images are always centered.

Buttons are activated using Enter or Space keys.

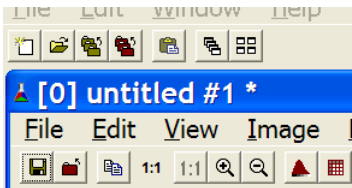
When IMPRESS and IMAGE are defined together, IUP does not show the element's border to provide a 3D effect; the user has to define the border in the image itself.

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

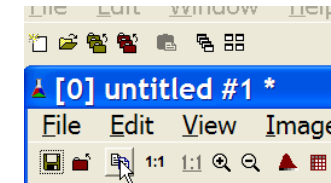
Here are some examples of buttons using Windows XP Visual Styles:



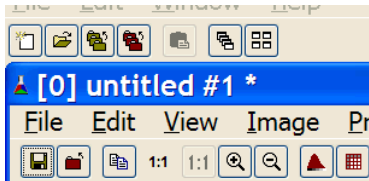
Text button without and with Visual Styles.



Windows XP without Visual Styles.

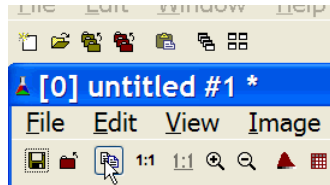


Windows XP without Visual Styles and FLAT=YES.
The border is displayed only when the mouse is over the button. The image of the button should be smaller when FLAT=YES.



Windows XP with Visual Styles.

Windows XP with Visual Styles and FLAT=YES.



Examples

Other: [Browse Example Files](#)

See Also

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

- [Creation](#)
- [Attributes](#)
- [Callbacks](#)
- [Notes](#)
- [Examples](#)
- [See Also](#)

IupCanvas

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

Creation

```
Ihandle* IupCanvas(const char *action); [in C]
iup.canvas{} -> (elem: ihandle) [in Lua]
canvas(action) [in LED]
```

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

Returns: the identifier of the created element, or NULL if an error occurs.

Attributes

BGCOLOR: Background color. 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.

CURSOR: Defines a cursor for the canvas.

SCROLLBAR (creation only): Associates a horizontal and/or vertical scrollbar to the canvas. Default: "NO".

DX: Size of the thumb in the horizontal scrollbar. Also the horizontal page size. Default: "0.1".

DY: Size of the thumb in the vertical scrollbar. Also the vertical page size. Default: "0.1".

POSX: Position of the thumb in the horizontal scrollbar. Default: "0.0".

POSY: Position of the thumb in the vertical scrollbar. Default: "0.0".

XMAX: Maximum value of the horizontal scrollbar. Default: "0.0".

XMIN: Minimum value of the horizontal scrollbar. Default: "1.0".

YMIN: Minimum value of the vertical scrollbar. Default: "0.0".

YMAX: Maximum value of the vertical scrollbar. Default: "1.0".

LINEX: The amount the thumb moves when an horizontal step is performed. Default: 1/10th of DX. (since 3.0)

LINEY: The amount the thumb moves when a vertical step is performed. Default: 1/10th of DY. (since 3.0)

XAUTOHIDE: When enabled, if DX >= XMAX-XMIN then the horizontal scrollbar is hidden. Default: "YES". (since 3.0)

YAUTOHIDE: When enabled, if DY >= YMAX-YMIN then the vertical scrollbar is hidden. Default: "YES". (since 3.0)

BORDER (creation only): Shows a border around the canvas. Default: "YES".

EXPAND: The default value is "YES". The natural size is the size of 1 character.

DRAGDROP: [Windows and GTK Only] Enable or disable the drag&drop of files. Default: NO, but if DROPFILES_CB is defined when the element is mapped then it will be automatically enabled.

DRAWSIZE: The size of the drawing area in pixels. This size is also used in the RESIZE_CB callback.

Notice that the drawing area size is not the same as RASTERSIZE. The SCROLLBAR and BORDER attributes affect the size of the drawing area.

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

MDICLIENT (creation only) [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".

HDC_WMPAINT [Windows Only]: Contains the HDC created with the BeginPaint inside the WM_PAINT message. Valid only during the ACTION callback.

[ACTIVE](#), [FONT](#), [X](#), [Y](#), [POSITION](#), [WID](#), [TIP](#), [SIZE](#), [RASTERSIZE](#), [ZORDER](#), [VISIBLE](#): also accepted.

Callbacks

ACTION: Action generated when the canvas needs to be redrawn.

```
int function(Ihandle *ih, float posx, float posy); [in C]
elem:action(posx, posy: number) -> (ret: number) [in Lua]
```

ih: identifier of the element that activated the event.
posx: thumb position in the horizontal scrollbar. The POSX attribute value.
posy: thumb position in the vertical scrollbar. The POSY attribute value.

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

DROPPFILES_CB [Windows Only]: Action generated when one or more files are dropped from the Windows Explorer in the dialog.

FOCUS_CB: Called when the canvas gets or loses the focus. It is called after the common callbacks GETFOCUS_CB and KILL_FOCUS_CB.

```
int function(Ihandle *ih, int focus); [in C]
elem:focus_cb(focus: number) -> (ret: number) [in Lua]
```

ih: identifier of the element that activated the event.
focus: is non zero if the canvas is getting the focus, is zero if it is losing the focus.

MOTION_CB: Action generated when the mouse is moved.

KEYPRESS_CB: Action generated when a key is pressed or released. It is called after the common callback K_ANY.

RESIZE_CB: Action generated when the canvas size is changed.

SCROLL_CB: Called when the scrollbar is manipulated.

WHEEL_CB: Action generated when the mouse wheel is rotated.

WOM_CB [Windows Only]: Action generated when an audio device receives an event.

MAP_CB, **GETFOCUS_CB**, **KILLFOCUS_CB**, **ENTERWINDOW_CB**, **LEAVEWINDOW_CB**, **K_ANY**, **HELP_CB**: All common callbacks are supported.

Notes

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 programmatically controlled using the global attribute **CURSORPOS**.

Examples

[Browse Example Files](#)

- [Creation](#)
- [Attributes](#)
- [Callbacks](#)
- [Notes](#)
- [Examples](#)
- [See Also](#)

IupFrame

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

Creation

```
Ihandle* IupFrame(Ihandle *child); [in C]
iup.frame{child: ihandle} -> (elem: ihandle) [in Lua]
frame(child) [in LED]
```

child: Identifier of an interface element which will receive the frame. It can be NULL.

Returns: the identifier of the created element, or NULL if an error occurs.

Attributes

FGCOLOR: Text title color. Not available in Windows when using Windows XP Visual Style.

BGCOLOR: background 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 later, 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.

Examples

[Browse Example Files](#)

- [Creation](#)
- [Attributes](#)

- [Callbacks](#)
- [Notes](#)
- [Examples](#)
- [See Also](#)

IupLabel

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

Creation

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

title: Text to be shown on the label. It can be NULL. It will set the TITLE attribute.

Returns: the identifier of the created element, or NULL if an error occurs.

Attributes

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 (non inheritable): Label's horizontal and vertical alignment. Possible values: "ALEFT", "ACENTER" and "ARIGHT", combined to "ATOP", "ACENTER" and "ABOTTOM". Default: "ALEFT:ACENTER". Partial values are also accepted, like "ARIGHT" or "xATOP", the other value will be used from the current alignment. In Motif, vertical alignment is restricted to "ACENTER". (vertical alignment since 3.0)

BGCOLOR: Background color.

ELLIPSIS [Windows and GTK only]: add an ellipsis: "..." to the text if there is not enough space to render the entire string. Can be "YES" or "NO". Default: "NO". (since 3.0)

FGCOLOR: Text color.

IMAGE (non inheritable): Label image. If set before map defines the behavior of the label to contain an image. The natural size will be size of the image in pixels.

IMINACTIVE (non inheritable) [Windows and Motif only]: Image name of the label when the inactive. If it is not defined then the IMAGE is used and the non transparent colors will be replaced by a darker version of the background color creating the disabled effect. Since GTK use an internal algorithm to build an inactive image that can not be avoided, so this attributes will not work in GTK, then it is simply ignored. In fact it is recommended that you always do NOT use this attribute and let all the drivers build the inactive images.

MARKUP [GTK only]: allows the title string to contains pango markup commands. Can be "YES" or "NO". Default: "NO".

MNEMONIC: enables or disable the use of a code to underline the character relative to a mnemonic (a key that when pressed activates a control). This will not associate any key, it will just underline the character in the title string. Can be "YES" or "NO". Default: "YES". Use "&" to mark the next character of the text as the character to be underlined. In the GTK driver it is also accepted the GTK key mark using "_" instead of "&". (since 3.0)

PADDING: internal margin. Works just like the MARGIN attribute of the IupHbox and IupVbox containers, but uses a different name to avoid inheritance problems. Not used when SEPARATOR is used. Default value: "0x0". (since 3.0)

SEPARATOR (creation only) (non inheritable): Turns the label into a line separator. Possible values: "HORIZONTAL" or "VERTICAL". When changed before mapping the EXPAND attribute is set to "HORIZONTAL" or "VERTICAL" accordingly.

TITLE (non inheritable): Label's text. If SEPARATOR or IMAGE are not defined before map, then the default behavior is to contain a text. The label behavior can not be changed after map. The '\n' character is accepted for line change. The natural size will be larger enough to include all the text in the selected font, even using multiple lines.

WORDWRAP [Windows and GTK only]: enables or disable the wrapping of lines that does not fits in the label. Can be "YES" or "NO". Default: "NO". Can only set WORDWRAP=YES if ALIGNMENT=ALEFT. (since 3.0)

[FONT](#), [EXPAND](#), [X](#), [Y](#), [POSITION](#), [WID](#), [TIP](#), [SIZE](#), [RASTERSIZE](#), [ZORDER](#), [VISIBLE](#): also accepted.

Callbacks

[ENTERWINDOW_CB](#), [LEAVEWINDOW_CB](#), [MAP_CB](#): common callbacks are supported.

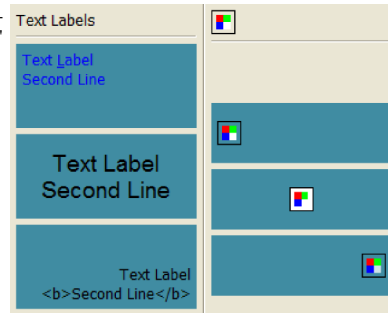
Notes

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

Examples

Default Text Label -	- Default Image Label
Horizontal Separator -	- Horizontal Separator
FGCOLOR = "0 0 255"...	
ALIGNMENT="ALEFT:ATOP" -	- BGCOLOR = "75 150 170"
BGCOLOR = "75 150 170"...	(8bpp Image)
FONT = "Helvetica, 14"	
ALIGNMENT = "ACENTER:ACENTER" -	- ALIGNMENT = "ACENTER"
	BGCOLOR = "75 150 170"
BGCOLOR = "75 150 170"	(24 bpp Image)
MARKUP = "YES" (GTK Only)	- ALIGNMENT = "ARIGHT"

ALIGNMENT = "ARIGHT:ABOTTOM"-
BGCOLOR = "75 150 170"



BGCOLOR = "75 150 170"
(32 bpp Image)

[Browse Example Files](#)

See Also

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

- [Creation](#)
- [Attributes](#)
- [Callbacks](#)
- [Notes](#)
- [Examples](#)
- [See Also](#)

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(const char *action); [in C]  
iup.list{} -> (elem: ihandle) [in Lua]  
list(action) [in LED]
```

action: String with the name of the action generated when the state of an item is changed. It can be NULL.

Returns: the identifier of the created element, or NULL 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.

SCROLLBAR (creation only): Associates an automatic horizontal and/or vertical scrollbar to the list. Default: "VERTICAL".

SELECTION: Selection interval. Valid only when EDITBOX=YES.

SELECTEDTEXT: Selection text. Valid only when EDITBOX=YES.

CLIPBOARD [write-only]: cut, copy or paste the selection to or from the clipboard. Values: "CUT", "COPY" or "PASTE". Valid only when EDITBOX=YES.

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

DROPEXPAND [Windows Only]: When DROPDOWN=Yes the size of the dropped list will expand to include the largest text. Can be "YES" or "NO". Default: "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 Lua]
```

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_cb(value: string) -> (ret: number) [in Lua]
```

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

This callback is called only when MULTIPLE==YES. And if this callback is defined the **ACTION** callback will not be called.

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_cb(c: number, after: string) -> (ret: number) [in Lua]
```

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

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:caret_cb(row, col: number) -> (ret: number) [in Lua]
```

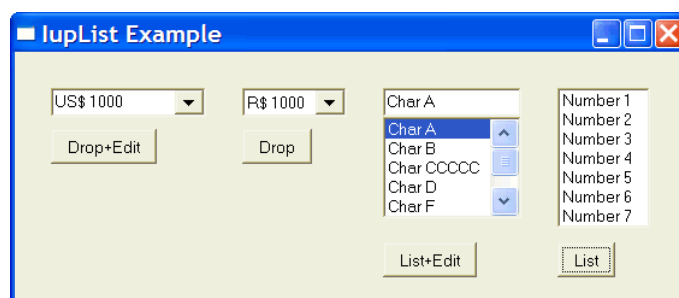
row, col: Row and column number.

Notes

Text is always left aligned.

Examples

[Browse Example Files](#)



See Also

[IupListDialog](#), [lupText](#)

- [Creation](#)
- [Attributes](#)
- [Callbacks](#)
- [Notes](#)
- [Examples](#)
- [See Also](#)

IupMultiLine

Creates an editable field with one or more lines.

Creation

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

```
iup.multiline{} -> (elem: ihandle) [in Lua]
multiline(action) [in LED]
```

action: name of the action generated when the user types something. It can be NULL.

Returns: the identifier of the created element, 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).

SCROLLBAR (creation only): Associates an automatic horizontal and/or vertical scrollbar to the multiline. Default: "YES".

SELECTION: Selection interval.

SELECTEDTEXT: Selection's text.

NC: Maximum number of characters. The maximum NC value is 2^{31} .

SIZE: Multiline size. Default: room for 5 characters in 1 line. If EXPAND is set to NO, when the dialog is resized and the multiline SIZE is not set, then the multiline initial size will be set to display its contents as the default control size calculation in IUP. But if EXPAND is enabled then the multiline will not expand to its contents and it will respect the expansion of the dialog.

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.

CLIPBOARD [write-only]: cut, copy or paste the selection to or from the clipboard. Values: "CUT", "COPY" or "PASTE".

WORDWRAP: if enable will force a word wrap of lines that are greater than the width of the control. Default: NO.

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 Lua]
```

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:caret_cb(row, col: number) -> (ret: number) [in Lua]
```

row, col: Row and column number.

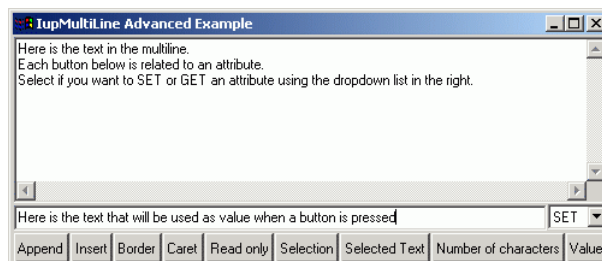
Notes

Text is always left aligned.

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

[Browse Example Files](#)



- [Creation](#)
- [Attributes](#)
- [Callbacks](#)
- [Notes](#)

- [Examples](#)
- [See Also](#)

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]
iup.spin{} -> (elem: ihandle) [in Lua]
```

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

```
Ihandle* IupSpinbox(Ihandle* ctrl); [in C]
iup.spinbox(ctrl: ihandle) -> (elem: ihandle) [in Lua]
```

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 the Shift key 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:spin_cb(inc: number) -> (ret: number) [in Lua]
```

See Also

[IupGetParam](#)

- [Creation](#)
- [Attributes](#)
- [Callbacks](#)
- [Notes](#)
- [Examples](#)
- [See Also](#)

IupText

Creates an editable field with one line.

Creation

```
Ihandle* IupText(const char *action); [in C]
iup.text{} -> (elem: ihandle) [in Lua]
text(action) [in LED]
```

action: name of the action generated when the user types something. It can be NULL.

Returns: the identifier of the created element, 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. The maximum NC value is 2³¹.

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. If EXPAND is set to NO, when the dialog is resized and the text SIZE is not set, then the text initial size will be set to display its contents as the default control size calculation in IUP. But if EXPAND is enabled then the text will not expand to its contents and it will respect the expansion of the dialog.

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

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

CLIPBOARD [write-only]: cut, copy or paste the selection to or from the clipboard. Values: "CUT", "COPY" or "PASTE".

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 Lua]
```

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:caret_cb(row, col: number) -> (ret: number) [in Lua]
```

row, col: Row and column 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

[Browse Example Files](#)

See Also

[IupMultiLine](#)

- [Creation](#)
- [Attributes](#)
- [Callbacks](#)
- [Notes](#)
- [Examples](#)
- [See Also](#)

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(const char *title, const char *action); [in C]
iup.toggle[{title = title: string}] -> (elem: ihandle) [in Lua]
toggle(title, action) [in LED]
```

title: Text to be shown on the toggle. It can be NULL. It will set the TITLE attribute.
action: name of the action generated when the toggle is selected. It can be NULL.

Returns: the identifier of the created element, 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 name. 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 name of the pressed toggle.

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

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

[Browse Example Files](#)

See Also

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

Additional Controls

Controls Library

Most of the additional 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`), and with the [CD](#) library (`cd` and `cdiup`). To use its bindings to Lua, the program must also be linked to the luacontrol library (`iupluacontrols`).

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](#).

- [Creation](#)
- [Attributes](#)
- [Callbacks](#)
- [Notes](#)
- [Examples](#)
- [See Also](#)

IupCells

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

This element is mostly based on application callbacks functions that determine the number of cells (rows and columns), their appearance and interaction. This mechanism offers full flexibility to applications, but requires programmers attention to avoid infinite loops inside these functions. Using callbacks, cells can be also grouped to form major or hierarchical elements, such as headers, footers etc. This callback approach was intentionally chosen to allow all cells to be dynamically and directly changed based on application's data structures. 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 is an additional control that depends on the CD library. It is included in the [Controls Library](#).

It inherits from [IupCanvas](#).

Originally implemented by André Clinio.

Creation

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

Returns: the identifier of the created element, or NULL if an error occurs.

Attributes

BOXED: Determines if the bounding cells' regions should be drawn with black lines. It can be "YES" or "NO". Default: "YES". If the span attributes are used, 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 be 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 always 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 always 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 respectively.

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

FULL_VISIBLE (write-only): 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 respectively.

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

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

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

[ACTIVE](#), [FONT](#), [X](#), [Y](#), [POSITION](#), [WID](#), [TIP](#), [SIZE](#), [RASTERSIZE](#), [ZORDER](#), [VISIBLE](#): also accepted.

Callbacks

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

```
int function(Ihandle* ih, int line, int column, int xmin, int xmax, int ymin, int ymax, cdCanvas* canvas); [in C]
elem:draw_cb(line, column, xmin, xmax, ymin, ymax: number, canvas: cdCanvas) -> (ret: number) [in Lua]
```

ih: identifier of the element that activated the event.

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 attribute IUP_CLIPPED is set (the default), all CD graphical primitives is clipped to the bounding region.

canvas: internal canvas CD used to draw the cells.

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* ih, int button, int pressed, int line, int column, int x, int y, char* status); [in C]
elem:mouseclick_cb(button, pressed, line, column, x, y: number, string: status) -> (ret: number) [in Lua]
```

Same as the [BUTTON_CB](#) IupCanvas callback with two additional parameters:

line, column: the grid position in the control where the event has occurred, in grid coordinates.

MOUSEMOTION_CB: called when the mouse moves over the control.

```
int function(Ihandle* ih, int line, int column, int x, int y, char *r); [in C]
elem:mousemotion_cb(x, y: number, r: string) -> (ret: number) [in Lua]
```

Same as the [MOTION_CB](#) IupCanvas callback with two additional parameters:

line, column: the grid position in the control where the event has occurred, in grid coordinates.

WIDTH_CB: called when the controls needs to know the column width

```
int function(Ihandle* ih, int column); [in C]
elem:width_cb(column: number) -> (ret: number) [in Lua]
```

ih: identifier of the element that activated the event.

column: the column index

Return: an integer that specifies the desired width (in pixels). Default is 60 pixels.

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

```
int function(Ihandle* ih, int line); [in C]
elem:height_cb(line: number) -> (ret: number) [in Lua]
```

ih: identifier of the element that activated the event.

line: the line index

Return: an integer that specifies the desired heigth (in pixels). Default is 30 pixels.

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

```
int function(Ihandle* ih); [in C]
elem:nlines_cb() -> (ret: number) [in Lua]
```

ih: identifier of the element that activated the event.

Return: an integer that specifies the number of lines. Default is 10 lines.

NCOLS_CB: called when the control needs to know its number of columns.

```
int function(Ihandle* ih); [in C]
elem:ncols_cb() -> (ret: number) [in Lua]
```

ih: identifier of the element that activated the event.

Return: an integer that specifies the number of columns. Default is 10 columns.

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

```
int function(Ihandle* ih, int line, int column); [in C]
elem:hspan_cb(line, column: number) -> (ret: number) [in Lua]
```

ih: identifier of the element that activated the event.

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

Return: an integer that specifies the desired span. Default is 1 (no span).

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

```
int function(Ihandle* ih, int line, int column); [in C]
elem:vspan_cb(line, column: number) -> (ret: number) [in Lua]
```

ih: identifier of the element that activated the event.

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

Return: an integer that specifies the desired span. Default is 1 (no span).

SCROLLING_CB: called when the scrollbars are activated.

```
int function(Ihandle* ih, int line, int column); [in C]
elem:scrolling_cb(line, column: number) -> (ret: number) [in Lua]
```

ih: identifier of the element that activated the event.

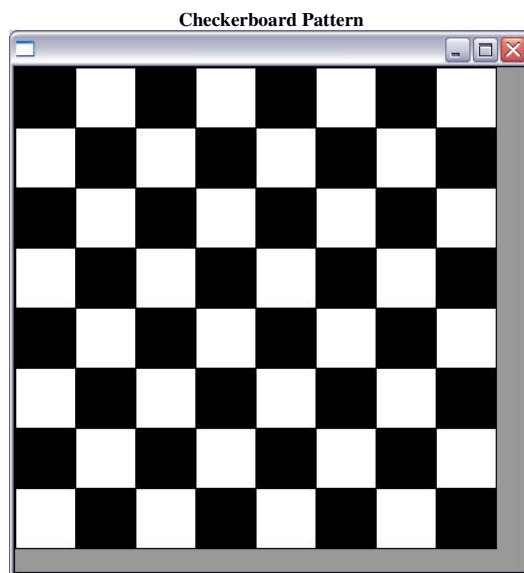
line, column: the first visible line and column indexes (in grid coordinates)

Return: If IUP_IGNORE the cell is not redrawn. By default the cell is always redrawn.

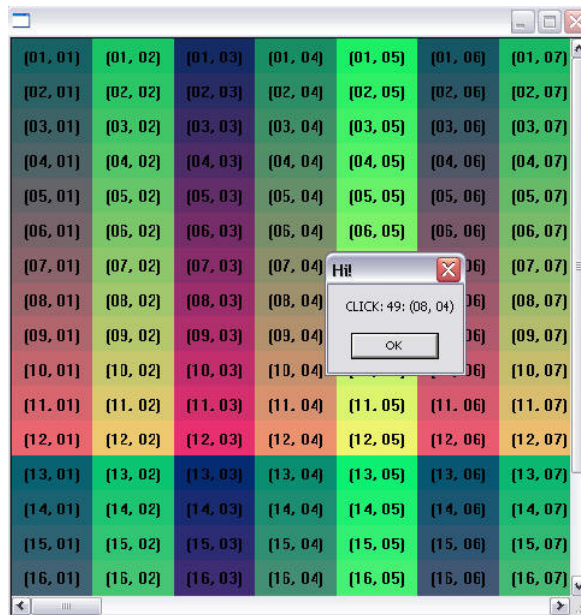
[MAP_CB](#), [GETFOCUS_CB](#), [KILLFOCUS_CB](#), [ENTERWINDOW_CB](#), [LEAVEWINDOW_CB](#), [K_ANY](#), [HELP_CB](#): All common callbacks are supported.

Examples

[Browse Example Files](#)



Numbering Cells



See Also

[IupCanvas](#)

- [Creation](#)
- [Attributes](#)
- [Callbacks](#)
- [Notes](#)
- [Examples](#)
- [See Also](#)

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.

This is an additional control that depends on the CD library. It is included in the [Controls Library](#).

It inherits from [IupCanvas](#).

Originally implemented by Andr   Clinio.

Creation

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

Returns: the identifier of the created element, or NULL if an error occurs.

Attributes

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

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

NUM_CELLS (non inheritable): 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 (non inheritable): Contains the number of lines or columns. Default: "1".

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

PREVIEW_SIZE (non inheritable): 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 (non inheritable): Contains the index of the primary color. Default "0" (black).

SECONDARY_CELL (non inheritable): 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".

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

[ACTIVE](#), [FONT](#), [X](#), [Y](#), [POSITION](#), [WID](#), [TIP](#), [SIZE](#), [RASTERSIZE](#), [ZORDER](#), [VISIBLE](#): also accepted.

Callbacks

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

```
char* function(Ihandle* ih, int cell); [in C]
elem:cell_cb(cell: number) -> (ret: string) [in Lua]
```

ih: identifier of the element that activated the event.

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

Return: a new color or NULL (nil in Lua) to ignore the change. By default nothing is changed.

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* ih, int cell); [in C]
elem:extended_cb(cell: number) -> (ret: number) [in Lua]
```

ih: identifier of the element that activated the event.

cell: index of the selected cell.

Return: If IUP_IGNORE the cell is not redrawn. By default the cell is always redrawn.

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* ih, int cell, int type); [in C]
elem:select_cb(cell, type: number) -> (ret: number) [in Lua]
```

ih: identifier of the element that activated the event.

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) .

Return: If IUP_IGNORE the selection is not accepted. By default the selection is always accepted.

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* ih, int prim_cell, int sec_cell); [in C]
elem:switch_cb(prim_cell, sec_cell: number) -> (ret: number) [in Lua]
```

ih: identifier of the element that activated the event.

prim_cell: index of the actual primary cell.

sec_cell: index of the actual secondary cell.

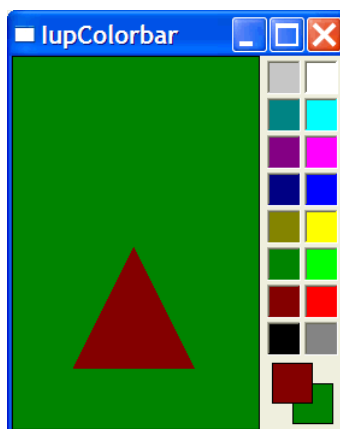
Return: If IUP_IGNORE the switch is not accepted. By default the switch is always accepted.

[MAP_CB](#), [GETFOCUS_CB](#), [KILLFOCUS_CB](#), [ENTERWINDOW_CB](#), [LEAVEWINDOW_CB](#), [K_ANY](#), [HELP_CB](#): All common callbacks are supported.

Examples

[Browse Example Files](#)

Creates a Colorbar for selection of two colors.



See Also

[IupCanvas](#), [IupImage](#)

- [Creation](#)
- [Attributes](#)

- [Callbacks](#)
- [Notes](#)
- [Examples](#)
- [See Also](#)

IupColorBrowser

Creates an element for selecting colors from the HLS (Hue Saturation Brightness) model, which allows the user to interactively choose a color.

This is an additional control that depends on the CD library. It is included in the [Controls Library](#).

For a dialog that simply returns the selected color, you can use function [IupGetColor](#).

Creation

```
Ihandle* IupColorBrowser(void); [in C]
iup.colorbrowser() (elem: ihandle) [in Lua]
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 C]
elem:drag_cb(r: number, g: number, b: number) -> (ret: number) [in Lua]
```

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); [in C]
elem:change_cb(r: number, g: number, b: number) -> (ret: number) [in Lua]
```

Examples

[Browse Example Files](#)

- [Creation](#)
- [Attributes](#)
- [Callbacks](#)
- [Notes](#)
- [Examples](#)
- [See Also](#)

IupDial

Creates a dial for regulating a given angular variable.

This is an additional control that depends on the CD library. It is included in the [Controls Library](#).

It inherits from [IupCanvas](#).

Creation

```
Ihandle* IupDial(const char *type); [in C]
iup.dial(type: string) -> (elem: ihandle) [in Lua]
dial(type) [in LED]
```

type: optional dial type, can be NULL. See TYPE attribute.

Returns: the identifier of the created element, or NULL if an error occurs.

Attributes

BGCOLOR: background color. If not defined will inherit the value from the parent tree, or from the dialog.

DENSITY: number of lines per pixel in the handle of the dial. Default is "0.2".

EXPAND: The default is "NO".

FGCOLOR: foreground color. The default value is "64 64 64". Not used for the circular dial.

SIZE: the default is "16x80", "80x16" or "40x35" according to the dial type.

TYPE (creation only) (non inheritable): dial layout configuration "VERTICAL", "HORIZONTAL" or "CIRCULAR". Default: "HORIZONTAL".

UNIT: Contains the unit of the angle. Can be "DEGREES" or "RADIANS". Default is "RADIANS".

VALUE (non inheritable): The dial angular value. The value is reset to zero when the interaction is started, except for TYPE=CIRCULAR. When type is vertical or horizontal, the dial measures relative angles. When type is circular the dial measure absolute angles, where the origin is at 3 O'Clock.

[ACTIVE](#), [FONT](#), [X](#), [Y](#), [POSITION](#), [WID](#), [TIP](#), [RASTERSIZE](#), [ZORDER](#), [VISIBLE](#): also accepted.

Callbacks

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 *ih, double angle)
elem:button_press_cb(angle: number) -> (ret: number) [in Lua]
```

ih: identifier of the element that activated the event.
angle: the dial value.

BUTTON_RELEASE_CB: Called when the user releases the left mouse button after pressing it over the dial.

```
int function(Ihandle *ih, double angle)
elem:button_release_cb(angle: number) -> (ret: number) [in Lua]
```

ih: identifier of the element that activated the event.
angle: the dial value.

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 *ih, double angle); [in C]
elem:mousemove_cb(angle: number) -> (ret: number) [in Lua]
```

ih: identifier of the element that activated the event.
angle: the dial value.

[MAP_CB](#), [GETFOCUS_CB](#), [KILLFOCUS_CB](#), [ENTERWINDOW_CB](#), [LEAVEWINDOW_CB](#), [K_ANY](#), [HELP_CB](#): All common callbacks are supported.

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 the mouse move callback is also called for every repetition.

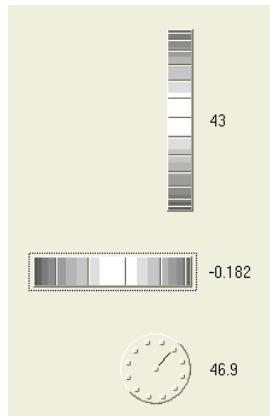
When the wheel is rotated only the mouse move callback is called, and it increments the last angle the dial was rotated.

In all cases the value is incremented or decremented by $\pi/10$ (18 degrees).

If you press Shift while using the arrow keys the increment is reduced to $\pi/100$ (1.8 degrees). Press the Home key in the circular dial to reset to 0. (since 3.0)

Examples

[Browse Example Files](#)



See Also

[IupCanvas](#)

- [Creation](#)
- [Attributes](#)
- [Callbacks](#)
- [Notes](#)
- [Examples](#)
- [See Also](#)

IupGauge

Creates a Gauge control. Shows a percent value that can be updated to simulate a progression. It inherits from [IupCanvas](#).

This is an additional control that depends on the CD library. It is included in the [Controls Library](#).

Creation

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

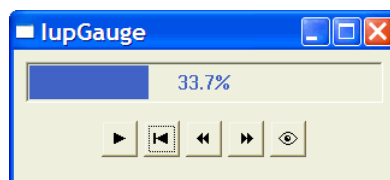
SIZE: The default is "170x17".

EXPAND: The default is "NO".

Examples

[Browse Example Files](#)

Creates a Gauge with a control bar.



See Also

[IupCanvas](#)

- [Creation](#)
- [Attributes](#)
- [Callbacks](#)
- [Notes](#)
- [Examples](#)
- [See Also](#)

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](#).

This is an additional control that depends on the CD library. It is included in the [Controls Library](#).

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* elem, ...); [in C]
Ihandle* IupTabsv(Ihandle** elems); [in C]
iup.tabs(elem, ...: ihandle) -> (elem: ihandle) [in Lua]
tabs(elem, ...) [in LED]
```

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

This function returns the created Tabs's identifier, or NULL if an error occurs. NULL defines the end of the list in C.

Attributes

ALIGNMENT: In this case it is propagated to the Zbox when changed (ALIGNMENT 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_ACTIVE: Indicates the font to be used when the tab is selected.

FONT_INACTIVE: Indicates the font to be used when the tab is inactive.

TABSIZE: Contains the text width of a single tab in pixels. 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_cb(new_tab, old_tab: ihandle) -> (ret: number) [in Lua]
```

self: Ihandle* of the control

new_tab: Ihandle* of the tab selected by the user

old_tab: Ihandle* of the previously selected tab

If returns IUP_IGNORE the tab is not changed.

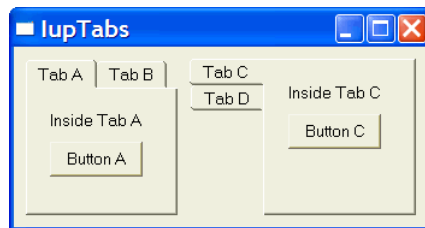
Notes

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

[Browse Example Files](#)



See Also

[IupCanvas](#)

- [Creation](#)
- [Attributes](#)
- [Callbacks](#)
- [Notes](#)
- [Examples](#)
- [See Also](#)

IupVal

Creates the Valuator control. It allows creating a regulator similar to IupDial, but with well-defined limits.

This is an additional control that depends on the CD library. It is included in the [Controls Library](#).

It inherits from [IupCanvas](#).

Creation

```
Ihandle* IupVal(char *type); [in C]
iup.val{type: string} -> (elem: ihandle) [in Lua]
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 valuator are bottom to up, and horizontal valuator 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. It is not the size of the increment. The increment size is "step*(max-min)". Default is "0.01".

PAGESTEP: Controls the increment for pagedown and pageup keys. It is not the size of the increment. The increment size is "pagestep*(max-min)". Default is "0.1".

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.

HANDLER_IMAGE: Handler image name. When defined the handle will not be drawn an this image will be used instead.

HANDLER_IMAGE_INACTIVE: Inactive handler image name. Used when **HANDLER_IMAGE** is used and the control is inactive.

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_cb(val: number) -> (ret: number) [in Lua]
```

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:button_press_cb(val: number) -> (ret: number) [in Lua]
```

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:button_release_cb(val: number) -> (ret: number) [in Lua]
```

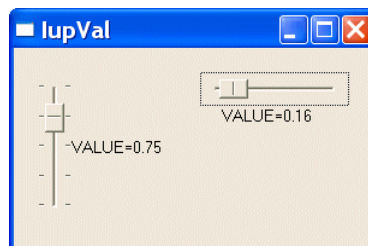
Notes

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

[Browse Example Files](#)



See Also

[IupCanvas](#)

- [Creation](#)
- [Attributes](#)
- [Callbacks](#)
- [Notes](#)
- [Examples](#)
- [See Also](#)

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](#).

This is an additional control that depends on the CD library. It is included in the [Controls Library](#).

It has two modes of operation: normal and callback mode. In normal mode string values are stored in attributes for each cell. In callback mode these attributes are ignored and the cells are filled with strings returned by the "VALUE_CB" callback. So the existence of this callback defines the mode the matrix will operate.

Creation

```
Ihandle* IupMatrix(char *action_cb); [in C]
iup.matrix{} -> (elem: ihandle) [in Lua]
```

`matrix(action_cb)` [in LED]

action_cb: 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](#)
[ALIGNMENT_n](#)
[BGCOLOR](#)
[FGCOLOR](#)
[FONT](#)
[FOCUS_CELL](#)
[VALUE](#)
[SORTSIGN_n](#)

Size Attributes

[NUMCOL](#)
[NUMCOL_VISIBLE](#)
[NUMLIN](#)
[NUMLIN_VISIBLE](#)
[WIDTHDEF](#)
[WIDTH_n](#)
[HEIGHT_n](#)
[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_CB](#) - Action generated when a keyboard event occurs.
[CLICK_CB](#) - Action generated when any mouse button is pressed over a cell.
[RELEASE_CB](#) - Action generated when any mouse button is released 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.
[SCROLLTOP_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.
[VALUE_EDIT_CB](#) - Action generated to notify the application that the value of a cell was edited.
[MARK_CB](#) - Action generated to verify the selection state of a cell.
[MARKEDIT_CB](#) - Action generated to notify the application that the selection state of a cell was changed.

Additional Methods in Lua

```
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* h, const char* a, int l, int c, char* v);
void IupMatStoreAttribute(Ihandle* ih, const char* a, int l, int c, char* v);
char* IupMatGetAttribute(Ihandle* ih, const char* a, int l, int c);
int IupMatGetInt(Ihandle* ih, const char* a, int l, int c);
float IupMatGetFloat(Ihandle* ih, const char* a, int l, int c);
void IupMatSetfAttribute(Ihandle* ih, const char* a, int l, int c, const 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(ih, "", 30, 10, v) = IupSetAttribute(n, "30:10", v)
IupMatSetAttribute(ih, "BGCOLOR", 30, 10, v) = IupSetAttribute(n, "BGCOLOR30:10", v)
IupMatSetAttribute(ih, "ALIGNMENT", 10, 0, v) = IupSetAttribute(n, "ALIGNMENT10:0", v) (*)
(*) noticed that in this case the second value will be ignored.
```

These functions are not available in Lua, since you can simply write:

```
elem["bgcolor"..l.." "..c] = v
or
elem["bgcolor30:10"] = v
```

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 lines height to fit titles with multiple lines. The user must change the lines size manually, by using attribute HEIGHTn. In IUPs 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 IUPs 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 arrows direction.
- **Page Up** and **Page Down:** Moves a visible page up or down.
- **Home:** Moves the focus to the first 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 below 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.

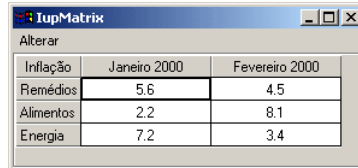
A marked cell will have its background attenuated to indicate that it is marked.

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

[Browse Example Files](#)

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.



IupMatrix		
Alterar		
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 explicitly set the REDRAW attribute.

ALIGNMENTn: Alignment of the cells in column n (n >= 0). No redraw is done. Possible values:

"ALEFT", "ACENTER" or "ARIGHT". Default: "ALEFT" for n=0 and "ACENTER" for n>0.

BGCOLOR: Background color of the matrix.

BGCOLOR*:C: Background color of column C (no redraw is done for this case).

BGCOLORL:*: Background color of line L (no redraw is done for this case).

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

When more than one attribute are defined, the background color will be selected following this priority: BGCOLORL:C, BGCOLORL:*, BGCOLOR*:C, and last BGCOLOR. (L or C >= 0)

Default BGCOLOR is white (255 255 255) for cells and the parent's BGCOLOR for titles.

Since the matrix control can be larger than the matrix itself, the empty area will always be filled with the parent's BGCOLOR.

FGCOLOR: Text color.

FGCOLOR*:C: Text color of column C (no redraw is done for this case).

FGCOLORL:*: Text color of line L (no redraw is done for this case).

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

When more than one attribute are define, the text color of a cell will be selected following this priority: FGCOLORL:C, FGCOLORL:*, FGCOLOR*:C, and last FGCOLOR. (L or C >= 0)

Default FGCOLOR is black (0 0 0).

FONT: Character font of the text.

FONTL:*: 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).

FONTL: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 cells line and column. But when updated or retrieved during cell editing, the edit control will be updated or consulted instead of the matrix cell. 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.

SORTSIGNn: Shows a sort sign (up or down arrow) in the column n 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 ($n \geq 0$). 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. Use the same units as SIZE.

HEIGHTn: Height of column n, where n is the number of the wished column ($n \geq 0$). If the height value is 0, the column will not be shown on the screen.

RASTERWIDTHn: Same as WIDTHn but in pixels.

RASTERHEIGHTn: Same as HEIGHTn but in pixels.

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 the name IupMatrixCrossCursor.

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 carets 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_CB: Action generated when a keyboard event occurs.

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

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). Notice that this value can be NULL if the cell does not have a value and the key pressed is not a character.

Possible return values are: IUP_DEFAULT validates the key, IUP_IGNORE ignores the key, IUP_CONTINUE forwards the key to IUPs conventional processing, or 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_cb(lin, col: number, r:string) -> (ret: number) [in Lua]
```

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.

RELEASE_CB: Action generated when any mouse button is released 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_cb(lin, col: number, r:string) -> (ret: number) [in Lua]
```

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_cb(lin, col: number) -> (ret: number) [in Lua]
```

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_cb(lin, col: number) -> (ret: number) [in Lua]
```

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_cb(lin, col: number) -> (ret: number) [in Lua]
```

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.

SCROLLTOP_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:scrolltop_cb(lin, col: number) -> (ret: number) [in Lua]
```

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 *green, unsigned int *blue); [in C]
elem:bgcolor_cb(lin, col: number) -> (red, green, blue, ret: number) [in Lua]
```

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 *green, unsigned int *blue); [in C]
elem:fgcolor_cb(lin, col: number) -> (red, green, blue, ret: number) [in Lua]
```

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, cdCanvas* cnv); [in C]
elem:draw_cb(lin, col, x1, x2, y1, y2: number, cnv: cdCanvas) -> (ret: number) [in Lua]
```

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.
cnv: internal canvas CD used to draw the matrix.

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_cb(lin, col: number) -> (ret: number) [in Lua]
```

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 lists 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_cb(drop: ihandle, lin, col: number) -> (ret: number) [in Lua]
```

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 v); [in C]
elem:dropselect_cb(lin, col: number, drop: ihandle, t: string, i, v: number) -> (ret: number) [in Lua]
```


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 (1 or 0).

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_cb(lin, col, mode: number) -> (ret: number) [in Lua]
```

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. So to make the cell read-only return IUP_IGNORE when mode=1. If the callback returns IUP_CONTINUE when mode=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 callback returns IUP_IGNORE when mode=0, the editing continues.

If the callback does not exists the value can always be edited and it is always accepted.

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. Called both for common cells and for line and column titles.

```
char* function(Ihandle* self, int lin, int col); [in C]
elem:value_cb(lin, col: number) -> (ret: string) [in Lua]
```

self: Identifier of the matrix interacting with the user.
lin, col: Coordinates of the cell.

Must return the string to be redrawn.

IMPORTANT: The existence of this callback defines the callback operation mode of the matrix.

VALUE_EDIT_CB: Action generated to notify the application that the value of a cell was edited. 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_cb(lin, col: number, newval: string) -> (ret: number) [in Lua]
```

self: Identifier of the matrix interacting with the user.
lin, col: Coordinates of the cell.
newval: String containing the new cell value

MARK_CB: Action generated to verify the selection state of a cell. Called only for common cells and only in callback mode.

```
int function(Ihandle* self, int lin, int col); [in C]
elem:mark_cb(lin, col: number) -> (ret: number) [in Lua]
```

self: Identifier of the matrix interacting with the user.
lin, col: Coordinates of the cell.

Must return the selection state (marked=1, not marked 0). If not defined the attribute "**MARKL:C**" will be returned.

MARKEDIT_CB: Action generated to notify the application that the selection state of a cell was changed. Since it is a notification, it cannot refuse the mark modification. Called only for common cells and only in callback mode.

```
int function(Ihandle* self, int lin, int col, int marked); [in C]
elem:markedit_cb(lin, col, marked: number) -> (ret: number) [in Lua]
```

self: Identifier of the matrix interacting with the user.
lin, col: Coordinates of the cell.
marked: selection state (marked=1, not marked 0).

If not defined the attribute "**MARKL:C**" will be updated. So if you define the **MARKEDIT_CB** the "**MARKL:C**" will NOT be updated and the callback **MARK_CB** must return the selection state. If you do not want to implement the **MARK_CB** callback set the "**MARKL:C**" attribute inside the **MARKEDIT_CB** callback.

The canvas callbacks [ACTION](#), [SCROLL_CB](#), [KEYPRESS_CB](#), [MOTION_CB](#), [MAP_CB](#), [RESIZE_CB](#) and [BUTTON_CB](#) can be changed but you should save and call the original definitions or the matrix will not correctly work. This can not be done in Lua.

See [IupCanvas](#).

- [Creation](#)
- [Attributes](#)
- [Callbacks](#)
- [Notes](#)
- [Examples](#)
- [See Also](#)

IupTree

Creates a tree containing nodes of branches or leaves. Both branches and leaves can have an associated text and image. It inherits from [IupCanvas](#).

This is an additional control that depends on the CD library. It is included in the [Controls Library](#).

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.

The selected node is the node with the focus rectangle, marked nodes have their background inverted.

Creation

```
Ihandle* IupTree(void); [in C]
iup.tree{} -> (elem: ihandle) [in Lua]
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 IMG PAPER (written sheet of paper). By default:

```
"IMAGELEAF" uses "IMGLEAF"
"IMAGEBRANCHCOLLAPSED" uses "IMGCOLLAPSED"
"IMAGEBRANCHEXPANDED" uses "IMGEXPANDED"

"IMGBLANK" and "IMG PAPER" are designed for use as "IMAGELEAF"
```

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 callback:

```
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 is 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 table and userdata.

```
int IupTreeSetUserId(Ihandle *self, int id, void *userid); [in C]
iup.TreeSetUserId(self: ihandle, id: number, userid: userdata) [in Lua]
iup.TreeSetTableId(self: ihandle, id: number, table: table) [in Lua]
```

self: Identifier of the IupTree interacting with the user.

id: Node identifier.

userid: User pointer associated to the node. Use NULL (nil) value to remove the association.

table: Table that should be associated to the node or leaf. Use nil value to remove the association.

Associations to Lua objects in Lua 5 are referenced in the Lua REGISTRY. So they can be retrieved later. This means also that the associated object will not be garbage collected until its reference is removed.

Also, the user should not use the same table to reference different nodes (neither in the same nor across different trees.)

```
void* IupTreeGetUserId(Ihandle *self, int id); [in C]
iup.TreeGetUserId(self: ihandle, id: number) -> (ret: userdata) [in Lua]
iup.TreeGetTable(self: ihandle, id: number) -> (ret: table) [in Lua]
```

self: Identifier of the IupTree interacting with the user.

id: Node identifier.

Returns the pointer or table associated to the node.

```
int IupTreeGetId(Ihandle *self, void *userid); [in C]
iup.TreeGetId(self: ihandle, userid: userdata) -> (ret: number) [in Lua]
iup.TreeGetTableId(self: ihandle, table: table) -> (ret: number) [in Lua]
```

self: Identifier of the IupTree interacting with the user.

userid: Pointer associated to the node.

table: Table that should be associated to the node or leaf.

Returns the id of the node on success and -1 (nil) on failure.

```
iup.TreeSetValue(self: ihandle, tree: table) [in Lua]
```

self: Identifier of the IupTree interacting with the user.

tree: table of nodes.

Initializes the tree using the given Lua table as values for the tree nodes. For example:

```
tree = {
  {
    "Horse",
    "Whale";
    branchname = "Mammals"
  },
  {
    "Shrimp",
    "Lobster";
    branchname = "Crustaceans"
  };
  branchname = "Animals"
}
iup.TreeSetValue(handle, tree)
```

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:

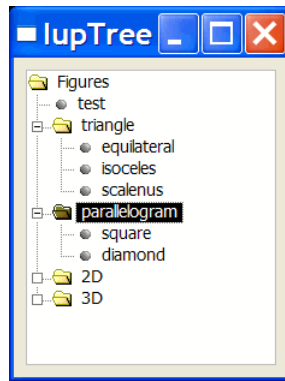
```
IupTreeSetAttribute(n, "KIND" , 30, v) = IupSetAttribute(n, "KIND30", v)
IupTreeSetAttribute(n, "ADDLEAF" , 10, v) = IupSetAttribute(n, "ADDLEAF10", v)
```

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

Examples

[Browse Example Files](#)

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

The selected node is scrolled so it becomes visible.

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

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 (write-only - id dependent attributes are not stored). Ex. "IMAGE2".

Name of an IupImage

Default: NULL.

An image is internally duplicated at the moment is set. So the application does not need to keep the image after setting the attribute. This is valid for all IMAGE* attributes of the IupTree.

IMAGEEXPANDEDid: Defines the image that will be shown on a specific node. It has no effect over leaves and is valid only for expanded branches. This attribute must always be used with the id number. This attribute can only be set (write-only - id dependent attributes are not stored). Ex. "IMAGEEXPANDED3".

Name of an IupImage

Default: NULL.

IMAGELEAF: Defines the image that will be shown for all leaves. Must be a 16x16 image.

Name of an IupImage

Default: "IMGLEAF".

Internal values "IMGBLANK" and "IMG PAPER" are also available.

IMAGEBRANCHCOLLAPSED: Defines the image that will be shown for all collapsed branches. Must be a 16x16 image.

Name of an IupImage

Default: "IMGCOLLAPSED".

IMAGEBRANCHEXPANDED: Defines the image that will be shown for all expanded branches. Must be a 16x16 image.

Name of an IupImage

Default: "IMGEXPANDED".

Nodes

NAME: Changes or retrieves the name of the selected node. Using the "**NAMEid**" form, where id is the node identifier, it is possible to change the name of any node.

The value must be a node name.

STATE: Changes or retrieves the state of the selected branch. Using the "**STATEid**" form, where id is the node identifier, it is possible to change the state of any branch. This attribute only works on branches. If it is used on a leaf, nothing will happen. Possible values:

- "EXPANDED": Expanded branch state (shows its children)"COLLAPSED": Collapsed branch state (hides its children)

DEPTH: If set, it defines the node's depth. Does not verify if the resulting tree is valid. If retrieved, it returns the node's depth. Using the "**DEPTHid**" form, where id is the node identifier, it is possible to refer to any node.

The value must be the node's depth.

KIND: Returns the kind of the selected node. Using the "**KINDid**" form, where id is the node identifier, it is possible to retrieve the kind of any node. This attribute can only be retrieved. Possible values:

- "LEAF": The node is a leaf"BRANCH": The node is a branch

PARENT: Returns the identifier of the selected node's parent. Using the "**PARENTid**" form, where id is the node identifier, it is possible to retrieve the identifier of any node. This attribute can only be retrieved.

COLOR: Color of the provided node. Using the form "**COLORid**", where id is the node identifier, it is possible to set or retrieve the color of any node. The value should be a string in the format "R G B" where R, G, B are numbers from 0 to 255.

Action

ADDLEAF: Adds a new leaf after the selected node. The id of the new leaf will be the id of the selected node + 1. The selected node is marked and all others unmarked. The selected node position remains the same. Using the "**ADDLEAFid**" form, where id is the node identifier, it is possible to insert a leaf after any node. In this case, the id of the inserted node will be id + 1. If the specified node does not exist, nothing happens. This attribute can only be set.

The value must be a leaf name.

ADDBRANCH: Adds a new branch after the selected node. The id of the new branch will be the id of the selected node + 1. The selected node is marked and all others unmarked. The selected node position remains the same. Using the "**ADDBRANCHid**" form, where id is the node identifier, it is possible to insert a branch after any node. In this case, the id of the inserted node will be id + 1. By default, all branches created are expanded. If the specified node does not exist, nothing happens. This attribute can only be set.

The value must be a branch name.

DELNODE: Removes the marked node (or its children). Using the "**DELNODEid**" form, where id is the node identifier, it is possible to remove any node. The root cannot be removed. If the specified node does not exist, nothing happens. This attribute can only be set. Possible values:

- "MARKED": Deletes all marked nodes (and all their children)
- "SELECTED": Deletes only the selected node (and its children)
- "CHILDREN": Deletes only the children of the selected node

Returns the identifier of the marked node's parent.

REDRAW: Forces an immediate redraw. It is necessary to force a redraw whenever the user adds or removes a node or a branch. The value is ignored.

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_cb(id, status: number) -> (ret: number) [in Lua]
```

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_cb(ids: table, n: number) -> (ret: number) [in Lua]
```

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_cb(id: number) -> (ret: number) [in Lua]
```

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_cb(id: number) -> (ret: number) [in Lua]
```

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_cb(id: number) -> (ret: number) [in Lua]
```

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_cb(id: number, name: string) -> (ret: number) [in Lua]
```

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:showrename_cb(id: number: string) -> (ret: number) [in Lua]
```

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 loses it focus.

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

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 loses 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); [in C]
elem:dragdrop_cb(drag_id, drop_id, isshift, iscontrol: number) -> (ret: number) [in Lua]
```

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.
issift: 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_cb(id: number) -> (ret: number) [in Lua]
```

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

The canvas callbacks KEYPRESS_CB, [ACTION](#), [SCROLL_CB](#), [MAP_CB](#), [RESIZE_CB](#) and [BUTTON_CB](#) can be changed but you should save and call the original definitions or the tree will not correctly work. This can not be done in Lua.

See also [IupCanvas](#).

- [Creation](#)
- [Attributes](#)
- [Callbacks](#)
- [Notes](#)
- [Examples](#)
- [See Also](#)

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 controls library (iupgl), 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 iuplua.h file must also be included in the source code. The program must be linked to the lua control library (iuplua).l).

To link with the OpenGL libraries in Windows add: opengl32.lib. In Motif add before the Motif libraries: -LGL.

The Lua binding does not include OpenGL functions. But you can use any OpenGL binding available. The IupLua console contains the [LuaGL](#) binding.

Creation

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

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

Returns: the identifier of the created element, or NULL if an error occurs.

Attributes

The IupGLCanvas element handle all attributes defined for a conventional canvas, see [IupCanvas](#).

Apart from these attributes, IupGLCanvas handle specific attributes used to define the kind of buffer to be instanced. Such attributes are all **creation only** attributes and must be set before the element is mapped on the native system. After the mapping, specifying these special attributes has no effect.

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 colors 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".

SHAREDCONTEXT: name of another IupGLCanvas that will share its display lists and textures. That canvas must be mapped before this canvas.

ERROR (read-only): If an error is found, returns a string containing a description of the error in English.

CONTEXT, **VISUAL** and **COLORMAP** (read-only): Returns "GLXContext", "XVisualInfo*" and "Colormap" respectively, in UNIX. And "HGLRC", "HDC" and "HPALETTE" respectively, in Win32.

Callbacks

The IupGLCanvas element understands all callbacks defined for a conventional canvas, see [IupCanvas](#).

Additionally:

RESIZE_CB: By default the resize callback sets:

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

Auxiliary Functions

```
void IupGLMakeCurrent(Ihandle* ih); [in C]
iup.GLMakeCurrent(ih: ihandle) [in Lua]
```

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

```
int IupGLIsCurrent(Ihandle* ih); [in C]
iup.GLIsCurrent(ih: ihandle) -> status: number [in Lua]
```

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

```
void IupGLSwapBuffers(Ihandle* ih); [in C]
iup.GLSwapBuffers(ih: ihandle) [in Lua]
```

Makes the BACK buffer visible. This function is necessary when a double buffer is used.

```
void IupGLPalette(Ihandle* ih, int index, float r, float g, float b); [in C]
iup.GLPalette(ih: ihandle, index, r, g, b: number); [in Lua]
```

Defines a color in the color palette. This function is necessary when INDEX color is used.

Notes

IMPORTANT: An OpenGL canvas when put inside an IupFrame in Win32 will not work.

The IupGLCanvas works with the GTK base driver in Windows and in UNIX (X-Windows).

Examples

[Browse Example Files](#)

See Also

[IupCanvas](#)

- [Creation](#)
- [Attributes](#)
- [Callbacks](#)
- [Notes](#)
- [Examples](#)
- [See Also](#)

IupPPlot

Creates a PPlot plot. It inherits from [IupCanvas](#).

PPlot is a library for creating plots that is system independent. It is maintained by Pier Philipsen (pierphil@xs4all.nl) and available at SourceForge <http://pplot.sourceforge.net/> with a very flexible license ([BSD License](#)). IupPPlot library includes the PPlot source code so no external references are needed. Also the standard PPlot distribution source code was changed to improve features and visual appearance.

IupPPlot first implementation was written by Marian Trifon (mtrif@katamail.com), then completed and documented by the IUP team.

Initialization and Usage

The **IupPPlotOpen** function must be called after a **IupOpen**, so that the control can be used. The "iup_pplot.h" file must also be included in the source code. The program must be linked to the controls library (iup_pplot), and with the CD library.

To make the control available in Lua, use the initialization function in C, **iupplotlua_open**, after calling **iuplua_open**. The iuplua_pplot.h file must also be included in the source code. The program must be linked to the lua control library (iuplua_pplot).

Guide

Each plot can contain 2 **axis** (X and Y), a **title**, a **legend box**, a **grid**, a **dataset area** and as many **datasets** you want.

Each data set is added using the **IupPPlotAdd** function. All other plot parameters are configured by attributes.

If no attribute is set, the default values are chosen to try to best display the plot.

When setting attributes the plot is NOT redrawn until the REDRAW attribute is set or a redraw event occurs.

The **dataset area** is delimited by a margin. Data is only plotted inside the dataset area. Axis and main title are positioned independent of this margin. It is very important to set the margins when using axis automatic scaling or the axis itself may be hidden.

The **legend box** is a list of the dataset names, each one drawn with the same color of the correspondent dataset. The box is located in one of the four corners of the dataset area.

The **grid** is automatically spaced accordingly the current axis displayed values.

The **title** is always centered in the top of the plot.

The **axis** are always positioned at the origin, except when CrossOrigin is disabled, then it is positioned at the left-bottom. If values are only positive then the origin will be placed in left bottom position. If values are negative and positive then origin will be placed inside the plot. The ticks in the axis are also automatically distributed.

Interaction

Zoom

Zoom in can be done selecting a region using the left mouse button. Zoom out is done with a single click of the left mouse button. If the **Ctrl+X** key combination is pressed the zoom selection is restricted to the X axis, the Y axis will be left unchanged. If the **Ctrl+Y** key combination is pressed the zoom selection is restricted to the Y axis, the X axis will be left unchanged. If the **Ctrl+R** key combination is pressed the zoom selection is restored to a free rectangle.

Each zoom in operation is stacked, so each zoom out operation goes back to the previous zoom selection.

Zoom operates on **AXS_XMAX**, **AXS_XMIN**, **AXS_YMAX**, **AXS_YMIN** even if **AUTOMIN/MAX** is enabled. The axis may be hidden depending on the selected rectangle.

CrossHair Cursor

If you press the **Ctrl+Shift** key combination, while holding the left mouse button down, a cross hair cursor will be displayed for each dataset in the plot. The X coordinate will control the cursor, the Y coordinate will reflect each dataset correspondent value.

Selection and Editing

Selection and editing of a dataset can be enabled using the **DS_EDIT** attribute.

To select all the samples in the dataset press the **Shift** key while clicking with the left mouse button near a sample in the dataset. To deselect all samples press the **Shift** key while clicking with the left mouse button in the background.

To select or deselect individual samples press the **Ctrl** key while clicking with the left mouse button near the sample in the dataset.

After selecting samples use the **Del** key to remove the selected samples. Also use the arrow keys to move the Y coordinate of the selected samples. Press the **Ctrl** key to increase the step size when moving the Y coordinate.

To Do's

- Add support of legend text near each dataset plot.
- Adjust AutoScale to start and end at major axis ticks.
- Improve visual appearance and ticks of bar mode.
- Improve the display of values near each sample.
- PPlot force the definitions of the margins. It should have a way to automatically calculate the margins when doing automatic scaling.
- PPlot has callback mode feature that is not available.
- PPlot force the Grid to be automatically spaced following the major ticks.
- PPlot generates lots of warnings on all compilers.
- PPlot uses C++ and the STL library. So it may have portability issues. So far it does not compile under Visual C++ 6. Use of STL is simple and can be removed.

Creation

```
Ihandle* IupPPlot(void); [in C]
iup.pplot{} -> (elem: ihandle) [in Lua]
pplot(action) [in LED]
```

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

Auxiliary Functions

```
void IupPPlotBegin(Ihandle* self, int strXdata); [in C]
iup.PPlotBegin(self: ihandle, strXdata: number) [in Lua]
```

Prepares a dataset to receive samples. If strXdata is 1 then the X axis value is a string.

```
void IupPPlotAdd(Ihandle* self, float x, float y); [in C]
iup.PPlotAdd(self: ihandle, x, y: number) [in Lua]
```

Adds a sample to the dataset. Can only be called if **IupPPlotBegin** was called with strXdata=0.

```
void IupPPlotAddStr(Ihandle* self, const char* x, float y); [in C]
iup.PPlotAddStr(self: ihandle, x: string, y: number) [in Lua]
```

Same as **IupPPlotAdd**, but allows to use a string as the X axis value. Can only be called if **IupPPlotBegin** was called with strXdata=1.

```
int IupPPlotEnd(Ihandle* self); [in C]
iup.PPlotEnd(self: ihandle) -> (index: number) [in Lua]
```

Adds a 2D dataset to the plot and returns the dataset index. Redraw is NOT done until the REDRAW attribute is set. Also it will change the current dataset index to the

return value. You can only set attributes of a dataset AFTER you added the dataset. Can only be called if **IupPPlotBegin** was called. Whenever you create a dataset all its "DS_*" attributes will be set to the default values. Notice that DS_MODE must be set before other "DS_*" attributes.

```
void IupPPlotInsert(Ihandle *self, int index, int sample_index, float x, float y); [in C]
void IupPPlotInsertStr(Ihandle *self, int index, int sample_index, const char* x, float y);
iup.PPlotInsert(self: ihandle, index, sample_index, x, y: number) [in Lua]
iup.PPlotInsertStr(self: ihandle, index, sample_index, x, y: number)
```

Inserts a sample in a dataset. It is used after the dataset is added to the plot.

```
void IupPPlotTransform(Ihandle* self, float x, float y, int *ix, int *iy); [in C]
iup.PPlotTransform(self: ihandle, x, y: number) -> (ix, iy: number) [in Lua]
```

Converts coordinates in plot units to pixels. It should be used in PREDRAW_CB and POSTDRAW_CB callbacks only. Output variables can be NULL if not used.

```
void IupPPlotPaintTo(Ihandle* self, cdCanvas* cnv); [in C]
iup.PPlotPaintTo(self: ihandle, cnv: cdCanvas) [in Lua]
```

Plots to the given CD canvas instead of the display canvas.

Attributes

The IupPPlot element understands all attributes defined for a conventional canvas, see [IupCanvas](#).

Apart from these attributes, IupPPlot understands specific attributes used to define the plot parameters:

REDRAW [write-only]: redraw the plot and update the display. Value is ignored. All other attributes will not update the display, so you can set many attributes without visual output.

USE_IMAGERGB: defines if the double buffer will use standard driver image (NO - faster) or an RGB image (YES - slower). Default: NO. The IMAGERGB driver has anti-aliasing which can improve the line drawing.

USE_GDI+ [Windows Only]: defines if the double buffer will use GDI+ (YES) for drawing or standard GDI (NO). Default: NO. The GDI+ driver has anti-aliasing which can improve the line drawing.

FONT: the font use in all text elements of the plot: title, legend and labels.

BGCOLOR: the background color. The default value is white "255 255 255".

FGCOLOR: the title color. The default value is black "0 0 0".

TITLE: the title. Located always at the top center area.

TITLEFONTSIZE, **TITLEFONTSTYLE**: the title font size and style. The default values depends on the FONT attribute and the returned value is NULL. Set to NULL, to use the FONT attribute values. Style can be "PLAIN", "BOLD", "ITALIC" or "BOLDITALIC".

MARGINLEFT, **MARGINRIGHT**, **MARGINTOP**, **MARGINBOTTOM**: margin of the dataset area. Default: "15", "15", "30", "15".

LEGENDSHOW: shows or hides the legend box. Can be YES or NO. Default: NO.

LEGENDPOS: legend box position. Can be: "TOPLEFT", "TOPRIGHT", "BOTTOMLEFT", or "BOTTOMRIGHT. Default: "TOPRIGHT".

LEGENDFONTSIZE, **LEGENDFONTSTYLE**: the legend box text font size and style.

GRIDLINESTYLE: line style of the grid. Can be: "CONTINUOUS", "DASHED", "DOTTED", "DASH_DOT", "DASH_DOT_DOT". Default is "CONTINUOUS".

GRIDCOLOR: grid color. Default: "200 200 200".

GRID: shows or hides the grid in both or a specific axis. Can be: YES (both), HORIZONTAL, VERTICAL or NO. Default: NO.

REMOVE [write-only]: removes a dataset given its index.

CLEAR [write-only]: removes all datasets. Value is ignored.

COUNT [read-only]: total number of datasets.

CURRENT: current dataset index. Default is -1. When a dataset is added it becomes the current dataset.

DS_LEGEND: legend text of the current dataset. Default is dynamically generated: "plot 0", "plot 1", "plot 2", ...

DS_COLOR: color of the current dataset and it legend text. Default is dynamically generated for the 6 first datasets, others are default to black "0 0 0". The first 6 are: 0="255 0 0", 1="0 0 255", 2="0 255 0", 3="0 255 255", 4="255 0 255", 5="255 255 0".

DS_MODE: drawing mode of the current dataset. Can be: "LINE", "BAR", "MARK" or "MARKLINE". Default: "LINE". This must be set before other "DS_*" attributes.

DS_LINESTYLE: line style of the current dataset. Can be: "CONTINUOUS", "DASHED", "DOTTED", "DASH_DOT", "DASH_DOT_DOT". Default is "CONTINUOUS".

DS_LINEWIDTH: line width of the current dataset. Default: 1.

DS_MARKSTYLE: mark style of the current dataset. Can be: "PLUS", "STAR", "CIRCLE", "X", "BOX", "DIAMOND", "HOLLOW_CIRCLE", "HOLLOW_BOX", "HOLLOW_DIAMOND". Default is "X".

DS_MARKSIZE: mark size of the current dataset. Default: 7.

DS_SHOWVALUES: enable or disable the display of the values near each sample. Can be YES or NO. Default: NO.

DS_REMOVE [write-only]: removes a sample from the current dataset given its index.

DS_EDIT: enable or disable the current dataset selection and editing. Can be YES or NO. Default: NO.

AXS_XCOLOR, AXS_YCOLOR: axis, ticks and label color. Default: "0 0 0".

AXS_XMAX, AXS_XMIN, AXS_YMAX, AXS_YMIN: minimum and maximum displayed values of the respective axis. Automatically calculated values when AUTOMIN or AUTOMAX are enabled.

AXS_XAUTOMIN, AXS_XAUTOMAX, AXS_YAUTOMIN, AXS_YAUTOMAX: configures the automatic scaling of the minimum and maximum display values. Can be YES or NO. Default: YES.

AXS_XLABEL, AXS_YLABEL: text label of the respective axis.

AXS_XLABELCENTERED, AXS_YLABELCENTERED: text label position at center (YES) or at top/right (NO). Default: YES.

AXS_XREVERSE, AXS_YREVERSE: reverse the axis direction. Can be YES or NO. Default: NO.

AXS_XCROSSORIGIN, AXS_YCROSSORIGIN: allow the axis to cross the origin and to be placed inside the dataset area. Can be YES or NO. Default: YES.

AXS_XSCALE, AXS_YSCALE: configures the scale of the respective axis. Can be: LIN (liner), LOG10 (base 10), LOG2 (base 2) and LOGN (base e). Default: LIN.

AXS_XFONTSIZE, AXS_YFONTSIZE, AXS_XFONTSTYLE, AXS_YFONTSTYLE: axis label text font size and style. See TITLEFONTSIZE and TITLEFONTSTYLE.

AXS_XTICK, AXS_YTICK: enable or disable the axis tick display. Can be YES or NO. Default: YES.

AXS_XTICKFORMAT, AXS_YTICKFORMAT: axis tick number C format string. Default: "%.0f".

AXS_XTICKFONTSIZE, AXS_YTICKFONTSIZE, AXS_XTICKFONTSTYLE, AXS_YTICKFONTSTYLE: axis tick number font size and style. See TITLEFONTSIZE and TITLEFONTSTYLE.

AXS_XAUTOTICK, AXS_YAUTOTICK: configures the automatic tick spacing. Can be YES or NO. Default: YES.

AXS_XTICKMAJORSPAN, AXS_YTICKMAJORSPAN: The spacing between major ticks. Default is 1 when AUTOTICK is disabled.

AXS_XTICKDIVISION, AXS_YTICKDIVISION: number of ticks between each major tick. Default is 5 when AUTOTICK is disabled.

AXS_XAUTOTICKSIZE, AXS_YAUTOTICKSIZE: configures the automatic tick size. Can be YES or NO. Default: YES.

AXS_XTICKSIZE, AXS_YTICKSIZE: size of ticks in pixels. Default is 5 when AUTOTICKSIZE is disabled.

AXS_XTICKMAJORSIZE, AXS_YTICKMAJORSIZE: size of major ticks in pixels. Default is 8 when AUTOTICKSIZE is disabled.

Callbacks

The IupGLCanvas element understands all callbacks defined for a conventional canvas, see [IupCanvas](#).

Additionally:

DELETE_CB: Action generated when the Del key is pressed to removed a sample from a dataset. If multiple points are selected it is called once for each selected point.

```
int function(Ihandle *self, int index, int sample_index, float x, float y); [in C]
elem:delete_cb(index, sample_index, x, y: number) -> (ret: number) [in Lua]
```

index: index of the dataset

sample_index: index of the sample in the dataset

x: X coordinate value of the sample

y: Y coordinate value of the sample

If the return value is IUP_IGNORE then the sample is not deleted.

DELETEBEGIN_CB, DELETEEND_CB: Actions generated when a delete operation will begin or end. But they are called only if DELETE_CB is also defined.

```
int function(Ihandle *self); [in C]
elem:deletebegin_cb() -> (ret: number) [in Lua]
elem:deleteend_cb() -> (ret: number) [in Lua]
```

If the DELETEBEGIN_CB callback returns IUP_IGNORE the delete operation for all the selected samples are aborted.

SELECT_CB: Action generated when a sample is selected. If multiple points are selected it is called once for each new selected point. It is called only if the selection state of the sample is about to be changed.

```
int function(Ihandle *self, int index, int sample_index, float x, float y, int select); [in C]
elem:select_cb(index, sample_index, x, y, select: number) -> (ret: number) [in Lua]
```

index: index of the dataset

sample_index: index of the sample in the dataset

x: X coordinate value of the sample

y: Y coordinate value of the sample

select: boolean value that a non zero value indicates if the point is to be selected.

If the return value is IUP_IGNORE then the sample is not selected.

SELECTBEGIN_CB, SELECTEND_CB: Actions generated when a selection operation will begin or end. But they are called only if SELECT_CB is also defined.

```
int function(Ihandle *self); [in C]
elem:selectbegin_cb() -> (ret: number) [in Lua]
elem:selectend_cb() -> (ret: number) [in Lua]
```

If the SELECTBEGIN_CB callback returns IUP_IGNORE the selection operation is aborted.

EDIT_CB: Action generated when a sample is selected. If multiple points are selected it is called once for each new selected point. It is called only if the selection state of the sample is about to be changed.

```
int function(Ihandle *self, int index, int sample_index, float x, float y, float *new_x, float *new_y); [in C]
elem:edit_cb(index, sample_index, x, y, new_x, new_y: number) -> (new_x, new_y, ret: number) [in Lua]
```

index: index of the dataset
sample_index: index of the sample in the dataset
x: X coordinate value of the sample
y: Y coordinate value of the sample
new_x: the new X coordinate value of the sample
new_y: the new Y coordinate value of the sample

If the return value is IUP_IGNORE then the sample is not edited. The application can change the new value before it is edited.

EDITBEGIN_CB, EDITEND_CB: Actions generated when an editing operation will begin or end. But they are called only if EDIT_CB is also defined.

```
int function(Ihandle *self); [in C]
elem:editbegin_cb() -> (ret: number) [in Lua]
elem:editend_cb() -> (ret: number) [in Lua]
```

If the EDITBEGIN_CB callback returns IUP_IGNORE the editing operation is aborted.

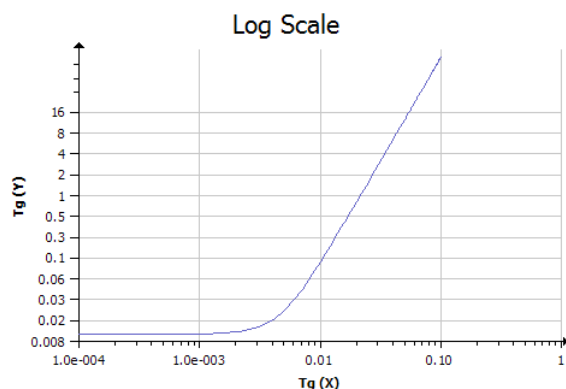
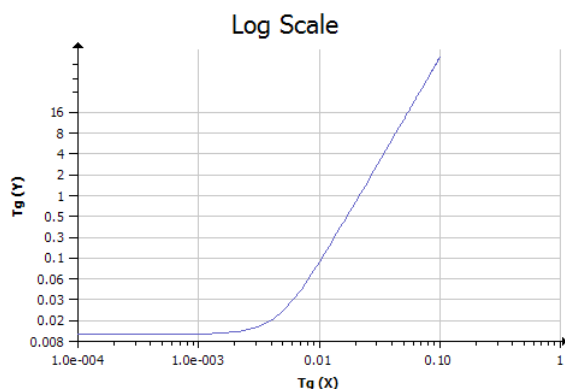
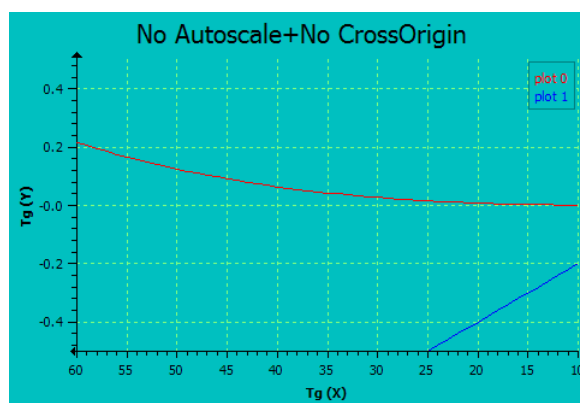
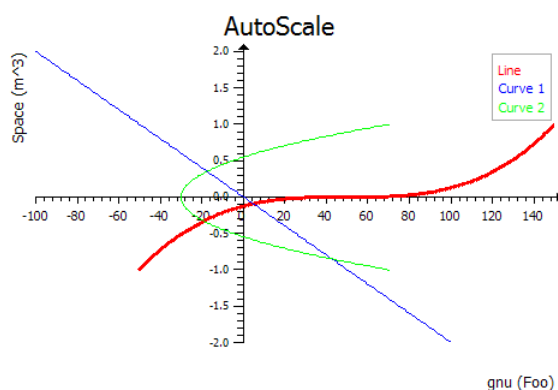
PREDRAW_CB, POSTDRAW_CB: Actions generated before and after the redraw operation. Predraw can be used to draw a different background and Postdraw can be used to draw additional information in the plot. Predraw has no restrictions, but Postdraw is clipped to the dataset area. To position elements in plot units, use the **IupPlotTransform** function.

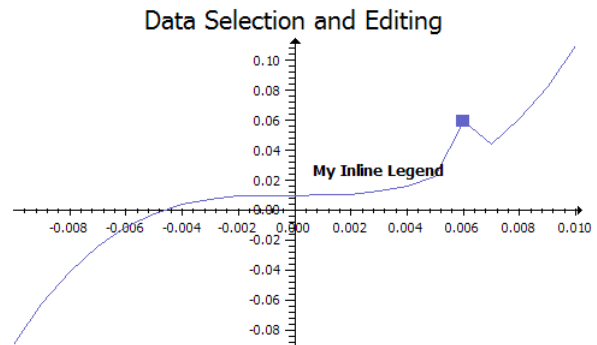
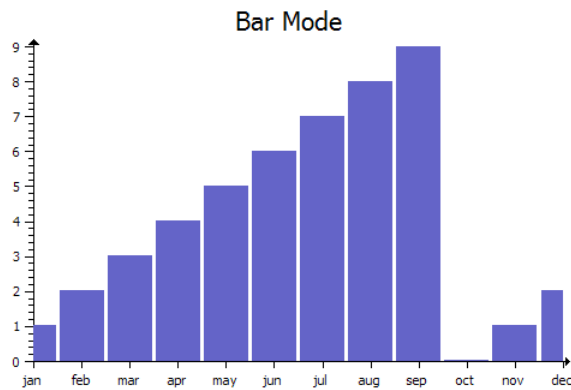
```
int function(Ihandle *self, cdCanvas* cnv); [in C]
elem:predraw_cb(cnv: cdCanvas) -> (ret: number) [in Lua]
elem:postdraw_cb(cnv: cdCanvas) -> (ret: number) [in Lua]
```

cnv: the CD canvas where the draw operation occurs.

Examples

[Browse Example Files](#)





See Also

[IupCanvas](#)

- [Creation](#)
- [Attributes](#)
- [Callbacks](#)
- [Notes](#)
- [Examples](#)
- [See Also](#)

IupOleControl

The IupOleControl hosts a 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 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 controls library (iupole).

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 lua control library (iupluaole).

Creation

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

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

[Browse Example Files](#)

See Also

IupCanvas

- [Creation](#)
- [Attributes](#)
- [Callbacks](#)
- [Notes](#)
- [Examples](#)
- [See Also](#)

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 library (iupspeech). 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.

Notes

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 (Visual C++ 7).

Examples

[Browse Example Files](#)

Keyboard

Keyboard navigation in 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 through the callbacks GETFOCUS_CB and KILLFOCUS_CB.

Two keys are also important in keyboard navigation: "Enter" and "Esc". But they are only active if the application registers 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.

Usually 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 header, which should be included in the application to use the key definitions. 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 some key combinations are not available because they are restrited by the system.

The iup_isprint(key) macro informs if a key can be directly used as a printable character. The isxkey(key) macro informs if a given key is an extended code. These macros are also available in Lua as a function with the same name.

In the table bellow there are the most common definitions. Change the definition to K_s*, K_c*, K_m* and K_y* when the repective modifier is pressed (Shift, Control, Alt and Sys). Sys in Windows is the Windows key and in Mac is the Apple key. Check the "iupkey.h" file header for all the definitions. To detect the combination of two or more modifiers use global attribute "MODKEYSTATE".

Note: Using the GTK in Windows does not generates the Win modifier key, the K_Print and the K_Pause keys (up to GTK version 2.8.18).

Key	Code / Callback
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
;	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
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

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 Arrow	K_UP
PgUp	K_PGUP
Left Arrow	K_LEFT
Middle	K_MIDDLE
Right Arrow	K_RIGHT
End	K_END
Down Arrow	K_DOWN
PgDn	K_PGDN
Pause	K_PAUSE
Print Screen	K_Print
Context Menu	K_Menu
`	K_acute
ç	K_ccedilla
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 that can have the focus. It is relative to the given element and does not depend on the element currently with the focus.

It will search for the next element first in the children, then in the brothers, then in the uncles and their children, and so on.

This sequence is not the same sequence used by the Tab key, which is dependent on the native system.

Parameters/Return

```
Ihandle* IupNextField(Ihandle* ih); [in C]
iup.NextField(ih: ihandle) -> (ih: ihandle) [in Lua]
```

ih: identifier of the interface element.

Returns the element that received the focus or NULL if failed.

See Also

[IupPreviousField](#).

IupPreviousField

Shifts the focus to the previous element that can have the focus. It is relative to the given element and does not depend on the element currently with the focus.

Parameters/Return

```
Ihandle* IupPreviousField(Ihandle* ih); [in C]
iup.PreviousField(ih: ihandle) -> (ih: ihandle) [in Lua]
```

ih: identifier of the interface element.

Returns the element that received the focus or NULL if failed.

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]
iup.GetFocus() -> elem: ihandle [in Lua]
```

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 *ih); [in C]
iup.SetFocus(ih: ihandle) -> ih: ihandle [in Lua]
```

ih: identifier of the interface element that will receive the keyboard focus. Only elements that can have the keyboard focus, are mapped, active and visible can be used, other elements are ignored.

Returns the identifier of the interface element that previously had 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.

- [Creation](#)
- [Attributes](#)
- [Callbacks](#)
- [Notes](#)
- [Examples](#)
- [See Also](#)

IupItem

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

Creation

```
Ihandle* IupItem(const char *title, const char *action); [in C]  
iup.item[{title = title: string}] -> elem: ihandle [in Lua]  
item(title, action) [in LED]
```

title: Text to be shown on the item. It can be NULL. It will set the TITLE attribute.

action: Name of the action generated when the item is selected. It can be NULL.

Returns: the identifier of the created element, or NULL if an error occurs.

Attributes

KEY: Associates a key to the item. The key will be used when navigating in the parent menu that contains the item. If the same character key is present in the title, then it will be underlined.

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".

In Windows, you can also underline a key using the prefix "&" just before the Key.

Menu items are activated using the Enter key.

Attention: Never use the same menu item in different menus.

Examples

[Browse Example Files](#)

See Also

[IupSeparator](#), [IupSubmenu](#), [IupMenu](#).

- [Creation](#)
- [Attributes](#)
- [Callbacks](#)
- [Notes](#)
- [Examples](#)
- [See Also](#)

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 *child, ...); [in C]  
Ihandle* IupMenuv(Ihandle **children); [in C]  
iup.menu(child, ...: ihandle) -> (elem: ihandle) [in Lua]  
menu(child, ...) [in LED]
```

child, ... : List of identifiers that will be grouped by the menu. NULL must be used to mark the end of the list in C. It can be empty.

Returns: the identifier of the created element, 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.

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.

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:

```
{"<item_name>"} - menu item
{"<submenu_name>","<menu>"} - submenu
{} - separator
```

For example:

```
mnu = iup.menu
{
  iup.submenu
  {
    iup.menu
    {
      iup.item{title="IupItem 1 Checked",value="ON"},
      iup.separator(),
      iup.item{title="IupItem 2 Disabled",active="NO"}
    }
    ;title="IupSubMenu 1"
  },
  iup.item{title="IupItem 3"},
  iup.item{title="IupItem 4"}
}
```

The same example using the cleaner syntax:

```
mnu = iup.menu
{
  {
    "IupSubMenu 1",
    iup.menu
    {
      {"IupItem 1 Checked";value="ON"},
      {} ,
      {"IupItem 2 Disabled";active="NO"}
    }
  },
  {"IupItem 3"},
  {"IupItem 4"}
}
```

Examples

[Browse Example Files](#)

See Also

[IupDialog](#), [IupPopup](#), [IupItem](#), [IupSeparator](#), [IupSubmenu](#)

- [Creation](#)
- [Attributes](#)
- [Callbacks](#)
- [Notes](#)
- [Examples](#)
- [See Also](#)

IupSeparator

Creates the separator interface element. It shows a line between two menu items.

Creation

```
Ihandle* IupSeparator(void); [in C]
iup.separator{} -> (elem: ihandle) [in Lua]
separator() [in LED]
```

Returns: the identifier of the created element, or NULL if an error occurs.

Notes

The separator is ignored when it is part of the definition of the items in a bar menu.

Examples

[Browse Example Files](#)

See Also

[IupItem](#), [IupSubMenu](#), [IupMenu](#).

- [Creation](#)
- [Attributes](#)
- [Callbacks](#)
- [Notes](#)
- [Examples](#)
- [See Also](#)

IupSubmenu

Creates a menu item that, when selected, opens another menu.

Creation

```
Ihandle* IupSubmenu(const char *title, Ihandle *menu); [in C]
iup.submenu(menu: ihandle[, title = title: string]) -> (elem: ihandle) [in Lua]
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. It can be NULL. It will set the TITLE attribute.
menu: menu identifier.

Returns: the identifier of the created element, 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.

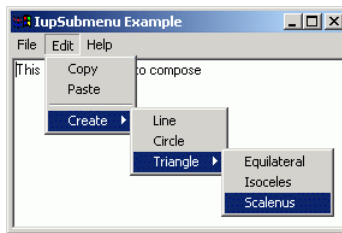
Callbacks

OPEN_CB: Called just before the submenu is opened.

MENUCLOSE_CB: Called right before the submenu is closed.

Examples

[Browse Example Files](#)



See Also

[IupItem](#), [IupSeparator](#), [IupMenu](#).

- [Creation](#)
- [Attributes](#)
- [Callbacks](#)
- [Notes](#)
- [Examples](#)
- [See Also](#)

IupImage, IupImageRGB, IupImageRGBA

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

(**IupImageRGB** and **IupImageRGBA** since IUP 3.0)

Creation

```
Ihandle* IupImage(int width, int height, const unsigned char *pixels); [in C]
Ihandle* IupImageRGB(int width, int height, const unsigned char *pixels); [in C]
Ihandle* IupImageRGBA(int width, int height, const unsigned char *pixels); [in C]

iup.image(line0: table, line1: table, ..., colors = colors: table) -> (elem: ihandle) [in Lua]
iup.imagergb(width = width: number, height = height: number, pixels = pixels: table) -> (elem: ihandle) [in Lua]
iup.imagergba(width = width: number, height = height: number, pixels = pixels: table) -> (elem: ihandle) [in Lua]

image(width, height, pixel0, pixel1, ...) [in LED]
imagergb(width, height, pixel0, pixel1, ...) [in LED]
imagergba(width, height, pixel0, pixel1, ...) [in LED]
```

width: Image width in pixels.

height: Image height in pixels.

pixels: Vector containing the value of each pixel. **IupImage** uses 1 value per pixel, **IupImageRGB** uses 2 values and **IupImageRGBA** uses 3 values per pixel. Each value is always 8 bit. Origin is at the top-left corner and data is oriented top to bottom, and left to right. The pixels array is duplicated internally so you can discard it after the call.

pixel0, pixel1, ...: Value of the pixels. But for **IupImageRGB** and **IupImageRGBA** in fact will be one value for each color channel.
line0, line1: unnamed tables, one for each line containing pixels values. See Notes below.
colors: table named colors containing the colors indices. See Notes below.

Returns: the identifier of the created element, or NULL 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 triple (For ex: "64 190 255"). 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" value must be defined within an index less than 16.

Used only for images created with **IupImage**.

BGCOLOR: The color used for transparency. If not defined uses the BGCOLOR of the control that contains the image.

BBP (read-only): returns the number of bits per pixel in the image. Images created with **IupImage** returns 8, with **IupImageRGB** returns 24 and with **IupImageRGBA** returns 32. (since 3.0)

CHANNELS (read-only): returns the number of channels in the image. Images created with **IupImage** returns 1, with **IupImageRGB** returns 3 and with **IupImageRGBA** returns 4. (since 3.0)

HEIGHT (read-only): Image height in pixels.

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. Default: "0:0".

RASTERSIZE (read-only): returns the image size in pixels. (since 3.0)

WID (read-only): returns the pixels data pointer. (since 3.0)

WIDTH (read-only): Image width in pixels.

Notes

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

An image created with all **IupImage*** can be reused for different buttons and labels.

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](#).

Colors

In Windows and Motif, the alpha channel in RGBA images is composed with the BGCOLOR when the image is used in a control. The images are mapped to 24bpp. In GTK the alpha channel is composited internally by GDK.

For **IupImage**, if a color is not set, then it is used a default color for the 16 first colors. The default color table is the same for Windows, GTK and Motif:

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

For images with more than 16 colors, and up to 256 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.

Samples

You can download 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](#).

The [EdPatt](#) and the [IMLAB](#) applications can load and save images in simplified 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 images in IupLua.

IupLua

In Lua, the 8bpp image can also be created using an unnamed table, using a series of tables for each line. Width and height will be guessed from the tables sizes. For example:

But after the image is created in Lua, the line tables are not accessible anymore, since "img[1]" will return the attribute "1" value which is the color "0 1 0". To access the original table values you must use "raawget", for example:

will return the first line of the original table. But if the image was created in C then there is no way to access its values.

[illegible]

Examples

See Also

IupImageLib (up to 2.x)


Initialization

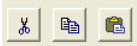


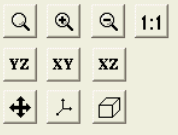






The library is quite large because of the images. To avoid using all the images get the source code and extract only the image you need.

This function loads all the images in the library.

This function releases all the images of the library.

The following names are defined after the library initialization. The images do NOT include the button borders, this is just a preview for buttons!

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_ONELLEFT" "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 with the IM library and with the function library (im and iupim libraries). 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 iuplua.h file must also be included in the source code. The program must be linked to the lua functions library (iuplua.h library).

Load

```
Ihandle* IupLoadImage(const char* file_name); [in C]
iup.LoadImage({file_name: string} -> {elem: ihandle}) [in Lua]
```

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]
iup.SaveImage({elem: ihandle, file_name, format: string} -> {ret: number}) [in Lua]
```

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

Associates a name with an interface element.

Parameters/Return

```
Ihandle *IupSetHandle(const char *name, Ihandle *ih); [in C]
iup.SetHandle(name: string, ih: ihandle) -> old_ih: ihandle [in Lua]
```

name: name of the interface element.

ih: identifier of the interface element. Use NULL to remove the association.

Returns the identifier of the interface element previously associated to the parameter **name**.

See Also

[IupGetHandle](#)

IupGetHandle

Returns the identifier of an interface element that has an associated name using IupSetHandle or using LED.

Parameters/Return

```
Ihandle *IupGetHandle(const char *name); [in C]
iup.GetHandle(name: string) -> ih: ihandle [in Lua]
```

name: name of an interface element.

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**.

See Also

[IupSetHandle](#).

IupGetName

Returns the name of an interface element if the element has an associated name using IupSetHandle or using LED. .

Parameters/Return

```
char* IupGetName(Ihandle* ih); [in C]
iup.GetName(ih: ihandle) -> (name: string) [in Lua]
```

ih: Identifier of the interface element.

Returns the name of an interface element.

Notes

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

Returns the names of all interface elements that have an associated name using IupSetHandle or using LED.

Parameters/Return

```
int IupGetAllNames(char *names[], int max_n); [in C]
iup.GetAllNames(max_n: number) -> (names: table, n: number) [in Lua]
```

names: table receiving the names

max_n: maximum number of names the table can receive.

Returns the number of names loaded to the table.

Notes

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

Returns the names of all dialogs that have an associated name using IupSetHandle or using LED. Other dialogs will not be returned.

Parameters/Return

```
int IupGetAllDialogs(char *names[], int max_n); [in C]
iup.GetAllDialogs(max_n: number) -> (names: table, n: number) [in Lua]
```

names: table receiving the names

max_n: maximum number of names the table can receive.

Returns the number of names loaded to the table.

Notes

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 OLD IUP font. See the [Old Character Fonts](#) table for a list of the old font names.

Deprecated function. Use the new common font names. See the [FONT](#) attribute.

Parameters/Return

```
char* IupMapFont(const char *iupfont); [in C]
iup.MapFont(iupfont : string) -> (driverfont : string) [in Lua]
```

This function returns the name of the native font.

See Also

[IupUnMapFont](#)

IupUnMapFont

Retrieves the name of the OLD IUP font, given the native font. See the [Old Character Fonts](#) table for a list of the old font names.

Deprecated function. Use the new common font names. See the [FONT](#) attribute.

Parameters/Return

```
char* IupUnMapFont(const char *driverfont); [in C]
iup.UnMapFont(driverfont :string) -> (iupfont : string) [in Lua]
```

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](#)

Old Character Fonts

Deprecated names. Use the new common font names. See the [FONT](#) attribute. Notice that size here is in **points** in Windows or GTK, and in **pixels** in Motif.

"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"

- [Creation](#)
- [Attributes](#)
- [Callbacks](#)
- [Notes](#)
- [Examples](#)
- [See Also](#)

IupTimer

Creates a timer which periodically invokes a callback when the time is up. Each timer should be destroyed using [IupDestroy](#).

Creation

```
Ihandle* IupTimer(void); [in C]
iup.timer{} -> (elem: ihandle) [in Lua]
timer() [in LED]
```

Returns: the identifier of the created element, or NULL if an error occurs.

Attributes

TIME: The time interval in milliseconds. In Windows the minimum value is 10ms.

RUN: Starts and stops the timer. Possible values: "YES" or "NO". Returns the current timer state.

WID (read-only): Returns the native serial number of the timer. Returns -1 if not running. A timer is mapped only when it is running.

Callbacks

ACTION_CB: Called when the time is up.

```
int function(Ihandle *ih); [in C]
elem:action_cb() -> (ret: number) [in Lua]
```

ih: identifier of the element that activated the event.

Return: IUP_CLOSE will be processed.

Examples

[Browse Example Files](#)

- [Creation](#)
- [Attributes](#)
- [Callbacks](#)
- [Notes](#)
- [Examples](#)
- [See Also](#)

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 Ihandle* to store private attributes.

Creation

```
Ihandle* IupUser(void); [in C]
[There is no equivalent in Lua]
[There is no equivalent in LED]
```

Returns: the identifier of the created element, or NULL if an error occurs.

IupGetClassName (renamed from IupGetType in 3.0)

Returns the name of the class of an interface element.

Parameters/Return

```
char* IupGetClassName(Ihandle* ih); [in C]
iup.GetClassName(ih: ihandle) -> (name: string) [in Lua]
```

ih: Identifier of the interface element.

Notes

The following names are known:

```
"image"
"button"
"canvas"
"dialog"
"fill"
```

```

"frame"
"hbox"
"item"
"separator"
"submenu"
"label"
"list"
"menu"
"radio"
"text"
"toggle"
"vbox"
"zbox"
"multiline"
"user"
"matrix"
"tree"
"dial"
"gauge"
"val"
"glcanvas"
"tabs"
"cells"
"colorbrowser"
"colorbar"
"spin"
"sbox"
"cbbox"

```

IupGetClassType (Since 3.0)

Returns the name of the native type of an interface element.

Parameters/Return

```

char* IupGetClassType(Ihandle* ih); [in C]
iup.GetClassType(ih: ihandle) -> (name: string) [in Lua]

```

ih: Identifier of the interface element.

Notes

There are only a few pre-defined class types:

```

"void" - No native representation - HBOX, VBOX, ZBOX, FILL, RADIO
"control" - Native controls - BUTTON, LABEL, TOGGLE, LIST, TEXT, MULTILINE, ITEM, SEPARATOR, SUBMENU, FRAME, others
"canvas" - Drawing canvas, also used as a base control for custom controls.
"dialog"
"image"
"menu"

```

IupHelp

Opens the given URL. In UNIX executes Netscape or Firefox (in Linux) 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 or using the global attribute "HELPPAPP".

Parameters/Return

```

int IupHelp(char* url); [in C]
iup.Help(url: string) [in Lua]

```

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

Returns 1 if successfull, -1 if failed. In Windows can return -2 if file not found.

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);
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);
int iupMaskSetFloat(Ihandle *h, int autofill, float min, float max);
int iupMaskMatSetInt(Ihandle *h, int autofill, int min, int max, int lin, int col);

```

```
int iupMaskMatSetFloat(Ihandle *h, int autofill, float min, float max, int lin, int col);
```

These functions set a mask that defines a limit to a 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 the mask 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);
int iupMaskMatRemove(Ihandle *h, int lin, int col);
```

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);
int iupMaskMatCheck (Ihandle *h, int lin, int col);
```

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);
int iupMaskGetFloat(Ihandle *h, float *fval);
int iupMaskGetInt(Ihandle *h, int *ival);
int iupMaskMatGet(Ihandle *h, char **val, int lin, int col);
int iupMaskMatGetFloat(Ihandle *h, float *fval, int lin, int col);
int iupMaskMatGetDouble(Ihandle *h, double *dval, int lin, int col);
int iupMaskMatGetInt(Ihandle *h, int *ival, int lin, int col);
```

These functions check if the text satisfies the mask, and they retrieve the fields 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.

Callbacks

MATCH_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); [in C]

elem:match_cb() -> (ret: number) [in Lua]
```

This callback is called only when the text satisfies the mask.

Notes

IMPORTANT: The iupmaskSet functions will replace the application keyboard callback. The previous callback will be called only if the text satisfies the mask. So you must set the application callback prior to calling the iupmaskSet functions. When using IupSetAttribute, must also call IupSetFunction before setting the mask.

To make the use of masks simpler, the following predefined masks were created:

IUPMASK_FLOAT - Float number

IUPMASK_UFLOAT - Unsigned Float number

IUPMASK_EFLOAT - Float number with exponential notation

IUPMASK_INT - Integer number

IUPMASK_UINT - Unsigned Integer number

Examples

[Browse Example Files](#)

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

IUP

3.0

Introduction

Internal SDK documentation of the IUP library, automatically generated using Doxygen (<http://www.doxygen.org/>).

Code Standards

Function Names (prefix format)

- IupFunc - User API, implemented in the core
- iupFunc - Internal Core API, implemented in the core, used in the core or in driver
- iupxxxFunc - Windows Internal API, implemented in driver xxx, used in driver xxx
- iupdrvFunc - Driver API, implemented in driver, used in the core or driver
- xxxFunc - Driver xxx local functions

Globais Variables (lower case format)

- iupxxx_var

Local Variables (lower case format, using module name)

- iyyy_var

File Names

- iupyyy.h - public headers
- iup_yyy.h/c - core
- iupxxx_yyy.h/c - driver

Structures

- Iyyy

File Comments (at start)

- Check an existant file for example.

Include Defines

- __IUPXXX_H (same file name, upper case, "__" preffix and replace "." by "_")

Documentation

- In the header, using Doxygen commands.
- Check an existant header for example.

Control SDK

Detailed Description

[Control Creation Guide](#)

Modules

[Attribute Environment](#)

[Base Class](#)

[Common Attribute Functions](#)

[List of Dialogs](#)

[Keyboard Focus](#)

[Key Coding and Key Callbacks](#)

[Object Handle](#)

[Class Registration](#)

Control Creation Guide (since 3.0)

Introduction

All the IUP controls use the same internal API to implement their functionalities.

Each control, needs to export only one function that register the control so it can be used by IupCreate and other functions. Actually another utility function is exported to simplify the creation of the control.

Internally the control must implement the methods of the IUP class, and create functions that handle attributes.

Control Class Registration

The new control must export function to register the control. This function is quite simple and it is just a call to [iupRegisterClass](#). For example:

```
void IupXxxOpen(void)
{
    iupRegisterClass(iupXxxClass());
}
```

The function iupXxxClass is internal to the control and it creates the control class.

Control Class Implementation

The function that creates the class will (1) initialize a base class, then (2) fill its configuration parameters, (3) set the class methods, (4) register the callbacks and (5) register the attributes. For example:

```
Iclass* iupXxxClass(void)
{
    /* (1) - initialize a base class */
    Iclass* ic = iupClassCreateBase(NULL);

    /* (2) - configuration parameters */
    ic->name = "xxx";
    ic->format = ""; /* no creation parameters */
    ic->nativetype = IUP_TYPECONTROL;
    ic->childtype = IUP_CHILDNONE;
    ic->interactive = 1;

    /* (3) - class methods */
    ic->Create = iXxxCreateMeth;
    ic->Map = iXxxMapMeth;
    ic->Destroy = iXxxDestroyMeth;
    ic->ComputeNaturalSize = iXxxComputeNaturalSizeMeth;
    ic->SetCurrentSize = iXxxSetCurrentSizeMeth;
    ic->LayoutUpdate = iXxxLayoutUpdateMeth;
```



```

ic->DisplayUpdate = iXxxDisplayUpdateMeth;
ic->SetPosition = iXxxSetPositionMeth;

/* (4) - callbacks */
iupClassRegisterCallback(ic, "XXX_CB", "i");
iupClassRegisterCallback(ic, "MAP_CB", "");
iupClassRegisterCallback(ic, "HELP_CB", "");
iupClassRegisterCallback(ic, "GETFOCUS_CB", "");
iupClassRegisterCallback(ic, "KILLFOCUS_CB", "");
iupClassRegisterCallback(ic, "ENTERWINDOW_CB", "");
iupClassRegisterCallback(ic, "LEAVEWINDOW_CB", "");
iupClassRegisterCallback(ic, "K_ANY", "i");

/* (5) - attributes */

/* Common */
iupClassRegisterAttribute(ic, "SIZE", iupGetSizeAttrib, iupDlgSetSizeAttrib, 0, 1, 0);
iupClassRegisterAttribute(ic, "RASTER_SIZE", iupGetRasterSizeAttrib, iupDlgSetRasterSizeAttrib, 0, 1, 0);
iupClassRegisterAttribute(ic, "WID", iupGetWidAttrib, iupNoSetAttrib, 0, 1, 0);
iupClassRegisterAttribute(ic, "FONT", 0, iupdrvSetFontAttrib, IupGetGlobal("DEFAULTFONT"), 1, 0);

/* Common, but only after Map */
iupClassRegisterAttribute(ic, "ACTIVE", iupGetActiveAttrib, iupSetActiveAttrib, "YES", 0, 0);
iupClassRegisterAttribute(ic, "VISIBLE", iupGetVisibleAttrib, iupSetVisibleAttrib, "YES", 0, 0);
iupClassRegisterAttribute(ic, "ZORDER", 0, iupdrvSetZorderAttrib, 0, 0, 0);

/* only the default value. */
iupClassRegisterAttribute(ic, "BORDER", 0, 0, "YES", 0, 0);

return ic;
}

```

You can use the `iupXxxClass` equivalent function of other controls to initialize a new base class for a new control that inherits the functionalities of the base class.

If the control is a native control then it usually will have separate modules for each driver. The `iupXxxClass` function could call a `iupdrvXxxClassInit(ic)` function to initialize methods and attributes that are driver dependent.

Control Creation

All controls can be created using the `IupCreate` functions. But it is a common practice to have a convenience function to create the control:

```

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

```

Control Exported Functions

The file header with the exported functions should look like this:

```

#ifndef __IUPGL_H
#define __IUPGL_H

#ifdef __cplusplus
extern "C" {
#endif

void IupXxxOpen(void);

Ihandle* IupXxx(void);

#ifdef __cplusplus
}
#endif

#endif

```

Object Handle

[\[Control SDK\]](#)

Detailed Description

Object handle for all the elements. Defines the `Ihandle` structure.

See [iup_object.h](#)

Data Structures

struct [Ihandle](#)

Typedefs

typedef `_InativeHandle` [InativeHandle](#)
 typedef `_IcontrolData` [IcontrolData](#)

Enumerations

```

enum {
    IUP_EXPAND_NONE = 0x00, IUP_EXPAND_H0 = 0x10, IUP_EXPAND_H1 = 0x20, IUP_EXPAND_W1 = 0x40,

    IUP_EXPAND_W0 = 0x80
}

```

Functions

```
void ** iupGetParamList (void *first, va_list arglist)
int iupHandleCheck (Ihandle *ih)
```

Typedef Documentation

typedef struct _InativeHandle [InativeHandle](#)

A simple definition that do not depends on the native system, but helps a lot when writing native code. See [iup_object.h](#) for definitions.

typedef struct _IcontrolData [IcontrolData](#)

Each control may define its own structure in its private module.

Enumeration Type Documentation

anonymous enum

Expand configuration

Function Documentation

```
void** iupGetParamList( void * first,
                        va_list arglist
                        )
```

Returns an array of parameters. Must call free for the returned value after usage. Used by the creation functions of objects that receives a NULL terminated array of parameters.

```
int iupHandleCheck( Ihandle * ih )
```

Checks if the handle is still valid based on the signature.

Base Class

[[Control SDK](#)]

Detailed Description

See [iup_class.h](#)

Data Structures

struct [Iclass](#)

Typedefs

```
typedef char *(* IattribGetFunc )(Ihandle *ih)
typedef char *(* IattribGetIdFunc )(Ihandle *ih, const char *name_id)
typedef int(* IattribSetFunc )(Ihandle *ih, const char *value)
typedef int(* IattribSetIdFunc )(Ihandle *ih, const char *name_id, const char *value)
```

Enumerations

```
enum {
    IUP\_TYPEVOID, IUP\_TYPECONTROL, IUP\_TYPECANVAS, IUP\_TYPEDIALOG,
    IUP\_TYPEIMAGE, IUP\_TYPEMENU
}
enum { IUP\_CHILDNONE, IUP\_CHILDONE, IUP\_CHILDMany }
enum { IUP\_NOT\_MAPPED, IUP\_MAPPED }
enum { IUP\_NO\_INHERIT, IUP\_INHERIT }
```

Functions

```
Iclass * iupClassGetBase (Iclass *parent)
void iupClassRelease (Iclass *ic)
void iupClassRegisterAttribute (Iclass *ic, const char *name, IattribGetFunc get, IattribSetFunc set, const char *default_value, int mapped, int inherit)
void iupClassRegisterCallback (Iclass *ic, const char *name, const char *format)
char * iupClassCallbackGetFormat (Iclass *ic, const char *name)
void iupLayoutSetExpand (Ihandle *ih)
void iupBaseSetCurrentSizeMethod (Ihandle *ih, int w, int h, int shrink)
void iupBaseSetPositionMethod (Ihandle *ih, int x, int y)
```

Typedef Documentation

typedef char*([IattribGetFunc](#))(Ihandle *ih)

GetAttribute called for a specific attribute. Used by [iupClassRegisterAttribute](#).

typedef char*([IattribGetIdFunc](#))(Ihandle *ih, const char *name_id)

GetAttribute called for a specific attribute when attrib_id is true.
Same as IattribGetFunc but handle attribute names with number ids at the end.

When calling `iupClassRegisterAttribute` just use a typecast.
Pure numbers are translated into `IDVALUEid`. Used by [iupClassRegisterAttribute](#).

`typedef int(* IattribSetFunc)(Ihandle *ih, const char *value)`

SetAttribute called for a specific attribute.
If returns 0, the attribute will not be stored in the attribute environment.
When `IupSetAttribute` is called using `value=NULL`, the `default_value` is passed to this function. Used by [iupClassRegisterAttribute](#).

`typedef int(* IattribSetIdFunc)(Ihandle *ih, const char *name_id, const char *value)`

SetAttribute called for a specific attribute when `attrib_id` is true.
Same as `IattribSetFunc` but handle attribute names with number ids at the end.
When calling `iupClassRegisterAttribute` just use a typecast.
Pure numbers are translated into `IDVALUEid`. Used by [iupClassRegisterAttribute](#).

Enumeration Type Documentation

anonymous enum

Known native types.

Enumerator:

<code>IUP_TYPEVOID</code>	No native representation - HBOX, VBOX, ZBOX, FILL, RADIO (handle==(void*)-1 always)
<code>IUP_TYPECONTROL</code>	Native controls - BUTTON, LABEL, TOGGLE, LIST, TEXT, MULTILINE, ITEM, SEPARATOR, SUBMENU, FRAME, others
<code>IUP_TYPECANVAS</code>	Drawing canvas, also used as a base control for custom controls.

anonymous enum

Possible number of children.

anonymous enum

Attribute map state dependency. Used by [iupClassRegisterAttribute](#).

anonymous enum

Attribute inheritance dependency. Used by [iupClassRegisterAttribute](#).

Function Documentation

[Iclass*](#) `iupClassGetBase(Iclass * parent)`

Allocates memory for the `Iclass` structure and initializes the attribute handling functions table.

`void iupClassRelease(Iclass * ic)`

Release the memory allocated by the class and parent classes.

```
void iupClassRegisterAttribute( Iclass *      ic,
                                const char *  name,
                                IattribGetFunc get,
                                IattribSetFunc set,
                                const char *  default_value,
                                int            mapped,
                                int            inherit
                                )
```

Register attribute handling functions. `get`, `set` and `default_value` can be `NULL`. `default_value` should point to a constant string. It will not be duplicated internally.
Notice that when an attribute is not defined then `default_value=NULL` and `inherit=IUP_INHERIT`.

```
void iupClassRegisterCallback( Iclass *      ic,
                               const char *  name,
                               const char *  format
                               )
```

Register the parameters of a callback. Used by language bindings. `format` follows the format specification of the class creation parameters format, but it adds the "double" option and remove array options. It can have none, one or more of the following.

- "b" = (unsigned char) - byte
- "i" = (int) - integer
- "f" = (float) - real
- "d" = (double) - real
- "s" = (char*) - string
- "h" = (Ihandle*) - element handle The default return value for all callbacks is "i" (int). But the return value can be specified using one of the above parameters, after all parameters using "=" to separate it from them.

```
char* iupClassCallbackGetFormat( Iclass *      ic,
                                 const char *  name
                                 )
```

Returns the format of the parameters of a registered callback. If `NULL` then the default callback definition is assumed.

`void iupLayoutSetExpand(Ihandle * ih)`

Updates the `expand` member of the `IUP` object from the `EXPAND` attribute. Should be called in the begining of the `ComputeNaturalSize` of each element.

```
void iupBaseSetCurrentSizeMethod( Ihandle * ih,
                                  int        w,
```

```
        int      h,  
        int      shrink  
    )
```

SetCurrentSize method for controls that are not containers.

```
void iupBaseSetPositionMethod( Ihandle * ih,  
                               int      x,  
                               int      y  
                               )
```

SetPositionSize method for controls that are not containers.

Common Attribute Functions

[\[Control SDK\]](#)

Detailed Description

Used by the controls when calling iupClassRegisterAttribute.

See [iup_class.h](#)

Functions

```
int iupBaseNoSetAttrib (Ihandle *ih, const char *value)  
char * iupBaseNoGetAttrib (Ihandle *ih)  
char * iupBaseGetWidAttrib (Ihandle *ih)  
    int iupBaseSetRasterSizeAttrib (Ihandle *ih, const char *value)  
    int iupBaseSetSizeAttrib (Ihandle *ih, const char *value)  
char * iupBaseGetSizeAttrib (Ihandle *ih)  
char * iupBaseGetRasterSizeAttrib (Ihandle *ih)  
char * iupBaseGetPositionAttrib (Ihandle *ih)  
char * iupBaseGetVisibleAttrib (Ihandle *ih)  
    int iupBaseSetVisibleAttrib (Ihandle *ih, const char *value)  
char * iupBaseGetActiveAttrib (Ihandle *ih)  
    int iupBaseSetActiveAttrib (Ihandle *ih, const char *value)  
    int iupdrvBaseSetZorderAttrib (Ihandle *ih, const char *value)  
    int iupdrvBaseSetCursorAttrib (Ihandle *ih, const char *value)  
char * iupdrvBaseGetXAttrib (Ihandle *ih)  
char * iupdrvBaseGetYAttrib (Ihandle *ih)  
    int iupdrvBaseSetTipAttrib (Ihandle *ih, const char *value)  
    int iupdrvBaseSetTipVisibleAttrib (Ihandle *ih, const char *value)  
void iupdrvBaseLayoutUpdateMethod (Ihandle *ih)  
void iupdrvBaseUnMapMethod (Ihandle *ih)  
char * iupdrvGetTitleAttrib (Ihandle *ih)  
    int iupdrvSetTitleAttrib (Ihandle *ih, const char *value)  
char * iupdrvGetClientSizeAttrib (Ihandle *ih)  
    int iupdrvSetBgColorAttrib (Ihandle *ih, const char *value)  
    int iupdrvSetFgColorAttrib (Ihandle *ih, const char *value)  
    int iupSetFontStyleAttrib (Ihandle *ih, const char *value)  
    int iupSetFontSizeAttrib (Ihandle *ih, const char *value)  
char * iupGetFontStyleAttrib (Ihandle *ih)  
char * iupGetFontSizeAttrib (Ihandle *ih)  
char * iupGetFontFaceAttrib (Ihandle *ih)
```

Function Documentation

```
int iupSetFontStyleAttrib( Ihandle * ih,  
                          const char * value  
                          )
```

Changes the FONT style only.

```
int iupSetFontSizeAttrib( Ihandle * ih,  
                         const char * value  
                         )
```

Changes the FONT size only.

```
char* iupGetFontStyleAttrib( Ihandle * ih )  
    Returns the FONT style.
```

```
char* iupGetFontSizeAttrib( Ihandle * ih )  
    Returns the FONT size.
```

```
char* iupGetFontFaceAttrib( Ihandle * ih )  
    Returns the FONT face.
```

Class Registration

[\[Control SDK\]](#)

Detailed Description

All controls are registered so the creation using `IupCreate` can work.

See [iup_register.h](#)

Functions

[Iclass](#) * [iupRegisterFindClass](#) (const char *name)

void [iupRegisterClass](#) ([Iclass](#) *ic)

Function Documentation

[Iclass](#) * [iupRegisterFindClass](#)(const char * *name*)

Returns a class instance from a class name. The class name must be previously registered using [iupRegisterClass](#).

void [iupRegisterClass](#)([Iclass](#) * *ic*)

Register a class.

Attribute Environment

[Control SDK]

Detailed Description

The attributes that are not stored at the control, are stored in a hash table (see [Hash Table](#)).

As a general rule use:

- `IupGetAttribute`, `IupSetAttribute`, ... : when care about control implementation, inheritance, hash table and default value
- `iupAttribGetStr`, `iupAttribSetStr`, ... : when NOT care, ONLY access the hash table
- `iupAttribGetStrDefault` : same as `iupAttribGetStr`, but checks the default value These different functions have very different performances and results. So use them wisely.

See [iup_attrib.h](#)

Functions

int [iupAttribIsInternal](#) (const char *name)

int [iupAttribIsPointer](#) (const char *name)

int [iupAttribIsInheritable](#) (const char *name)

void [iupAttribSetStr](#) (Ihandle *ih, const char *name, char *value)

void [iupAttribStoreStr](#) (Ihandle *ih, const char *name, const char *value)

void [iupAttribSetStrf](#) (Ihandle *ih, const char *name, const char *format,...)

void [iupAttribSetInt](#) (Ihandle *ih, const char *name, int num)

void [iupAttribSetFloat](#) (Ihandle *ih, const char *name, float num)

char * [iupAttribGetStr](#) (Ihandle *ih, const char *name)

char * [iupAttribGetStrDefault](#) (Ihandle *ih, const char *name)

int [iupAttribGetInt](#) (Ihandle *ih, const char *name)

int [iupAttribGetIntDefault](#) (Ihandle *ih, const char *name)

float [iupAttribGetFloat](#) (Ihandle *ih, const char *name)

float [iupAttribGetFloatDefault](#) (Ihandle *ih, const char *name)

Function Documentation

int [iupAttribIsInternal](#)(const char * *name*)

Returns true if the attribute name if in the internal format "_IUP...".

int [iupAttribIsPointer](#)(const char * *name*)

Returns true if the attribute name is a known pointer.

int [iupAttribIsInheritable](#)(const char * *name*)

Returns true if the attribute is inheritable. valid only for a small group of common attributes. Not used internally. The inheritance is handled directly by the Attribute Function registration in the [Base Class](#).

```
void iupAttribSetStr( Ihandle * ih,
                    const char * name,
                    char * value
                    )
```

Sets the attribute only in the attribute environment as a pointer. It ignores children.

```
void iupAttribStoreStr( Ihandle * ih,
                      const char * name,
                      const char * value
                      )
```

Sets the attribute only in the attribute environment as a string. The string is internally duplicated. It ignores children.

```
void iupAttribSetStrf( Ihandle *  ih,
                     const char * name,
                     const char * format,
                     ...
                     )
```

Sets the attribute only in the attribute environment as a string. The string is internally duplicated. Use same format as printf. It ignores children.

```
void iupAttribSetInt( Ihandle *  ih,
                    const char * name,
                    int          num
                    )
```

Sets an integer attribute only in the attribute environment. It will be stored as a string. It ignores children.

```
void iupAttribSetFloat( Ihandle *  ih,
                      const char * name,
                      float         num
                      )
```

Sets an floating point attribute only in the attribute environment. It will be stored as a string. It ignores children.

```
char* iupAttribGetStr( Ihandle *  ih,
                     const char * name
                     )
```

Returns the attribute from the attribute environment. It ignores inheritance.

```
char* iupAttribGetStrDefault( Ihandle *  ih,
                             const char * name
                             )
```

Returns the attribute from the attribute environment, but if not defined then returns the registered default value if any. It ignores inheritance.

```
int iupAttribGetInt( Ihandle *  ih,
                    const char * name
                    )
```

Returns the attribute from the attribute environment as an integer. It ignores inheritance.

```
int iupAttribGetIntDefault( Ihandle *  ih,
                           const char * name
                           )
```

Returns the attribute from the attribute environment as an integer, but if not defined then returns the registered default value if any. It ignores inheritance.

```
float iupAttribGetFloat( Ihandle *  ih,
                       const char * name
                       )
```

Returns the attribute from the attribute environment as a floating point. It ignores inheritance.

```
float iupAttribGetFloatDefault( Ihandle *  ih,
                               const char * name
                               )
```

Returns the attribute from the attribute environment as a floating point, but if not defined then returns the registered default value if any. It ignores inheritance.

List of Dialogs

[\[Control SDK\]](#)

Detailed Description

See [iup_dlglist.h](#)

Functions

```
void iupDlgListAdd (Ihandle *ih)
void iupDlgListRemove (Ihandle *ih)
Ihandle * iupDlgListFirst (void)
Ihandle * iupDlgListNext (void)
void iupDlgListVisibleInc (void)
void iupDlgListVisibleDec (void)
int iupDlgListVisibleCount (void)
```

Function Documentation

```
void iupDlgListAdd( Ihandle * ih )
```

Adds a dialog to the list. Used only in IupDialog.

```
void iupDlgListRemove( Ihandle * ih )
```

Removes a dialog from the list. Used only in IupDestroy.

```
Ihandle* iupDlgListFirst( void )
```

Starts a loop for all the created dialogs.

Ihandle* iupDlgListNext(void)

Retrieve the next dialog on the list. Must call iupDlgListFirst first.

void iupDlgListVisibleInc(void)

Increments the number of visible dialogs.

void iupDlgListVisibleDec(void)

Decrements the number of visible dialogs.

int iupDlgListVisibleCount(void)

Returns the number of visible dialogs.

Keyboard Focus

[\[Control SDK\]](#)

Detailed Description

See [iup_focus.h](#)

Functions

int [iupFocusCanAccept](#) (Ihandle *ih)
void [iupCallGetFocusCb](#) (Ihandle *ih)
void [iupCallKillFocusCb](#) (Ihandle *ih)

Function Documentation

int iupFocusCanAccept(Ihandle * *ih*)

Utility to check if a control can have the keyboard input focus. Checks the control class, active state and visible.

void iupCallGetFocusCb(Ihandle * *ih*)

Call GETFOCUS_CB and FOCUS_CB.

void iupCallKillFocusCb(Ihandle * *ih*)

Call KILLFOCUS_CB and FOCUS_CB.

Key Coding and Key Callbacks

[\[Control SDK\]](#)

Detailed Description

See [iup_key.h](#)

Functions

char * [iupKeyCodeToName](#) (int code)
int [iupKeyNameToCode](#) (const char *name)
int [iupKeyCanCaps](#) (int code)
void [iupKeyForEach](#) (void(*func)(const char *name, int code, void *user_data), void *user_data)
int [iupKeyCallKeyCb](#) (Ihandle *ih, int c)
int [iupKeyCallKeyPressCb](#) (Ihandle *ih, int code, int press)

Function Documentation

char* iupKeyCodeToName(int *code*)

Returns the key name from its code. Returns NULL if code not found.

int iupKeyNameToCode(const char * *name*)

Returns the key code from its name. Returns 0 if name not found.

int iupKeyCanCaps(int *code*)

Returns true if the key code can be changed by CAPSLOCK.

void iupKeyForEach(void(*) (const char *name, int code, void *user_data) *func*,
void * *user_data*
)

Calls a function for each defined key.

```
int iupKeyCallKeyCb( Ihandle * ih,
                    int      c
                    )
```

Calls the K_ANY or K_* callbacks. Should be called when a keyboard event occurred.

```
int iupKeyCallKeyPressCb( Ihandle * ih,
                          int      code,
                          int      press
                          )
```

Calls the KEYPRESS_CB callback. Should be called when a keyboard event occurred.

Driver Interface

Detailed Description

Each driver must export the symbols defined here.

See [iup_drv.h](#)

Modules

[Driver Font Interface](#)
[Driver Information Interface](#)

Functions

```
int iupdrvSetGlobal( const char *name, const char *value )
char * iupdrvGetGlobal( const char *name )
void iupdrvSetIdleFunction( Icallback func )
void iupdrvScreenToClient( Ihandle *ih, int *x, int *y )
int iupdrvIsVisible( Ihandle *ih )
int iupdrvIsActive( Ihandle *ih )
void iupdrvSetVisible( Ihandle *ih, int enable )
void iupdrvSetActive( Ihandle *ih, int enable )
void iupdrvDisplayUpdate( Ihandle *ih )
void iupdrvReparent( Ihandle *child, Ihandle *parent )
void iupdrvDrawFocusRect( Ihandle *ih, void *gc, int x, int y, int w, int h )
```

Function Documentation

```
int iupdrvSetGlobal( const char * name,
                    const char * value
                    )
```

Sets a global environment attribute. Called from IupSetGlobal and IupStoreGlobal. Must return 1 is process the attribute, or 0 is not.

```
char* iupdrvGetGlobal( const char * name )
```

Returns a global environment attribute. Called from IupGetGlobal.

```
void iupdrvSetIdleFunction( Icallback func )
```

Changes the idle callback. Called from IupSetFunction.

```
void iupdrvScreenToClient( Ihandle * ih,
                          int *    x,
                          int *    y
                          )
```

Convert the coordinates from screen relative to client area relative.

```
int iupdrvIsVisible( Ihandle * ih )
```

Returns true if the element is visible.

```
int iupdrvIsActive( Ihandle * ih )
```

Returns true if the element is active.

```
void iupdrvSetVisible( Ihandle * ih,
                      int      enable
                      )
```

Changes the visible state of an element. Not used for dialogs.

```
void iupdrvSetActive( Ihandle * ih,
                      int      enable
                      )
```

Changes the active state of an element.

```
void iupdrvDisplayUpdate( Ihandle * ih )
```


Force a redraw of a control.

```
void iupdrvReparent( Ihandle * child,
                    Ihandle * parent
                    )
```

Reparent the native control.

```
void iupdrvDrawFocusRect( Ihandle * ih,
                          void * gc,
                          int x,
                          int y,
                          int w,
                          int h
                          )
```

Draws a focus rectangle

Driver Font Interface

[\[Driver Interface\]](#)

Detailed Description

Each driver must export the symbols defined here.

See [iup_drvfont.h](#)

Functions

```
void iupdrvGetCharSize( Ihandle *ih, int *charwidth, int *charheight)
void iupdrvStringSize (Ihandle *ih, const char *str, int *w, int *h)
void iupdrvMultiLineStringSize (Ihandle *ih, const char *str, int *w, int *h)
char * iupdrvGetSystemFont (void)
int iupdrvSetNativeFontAttrib (Ihandle *ih, const char *value)
char * iupGetFontAttrib (Ihandle *ih)
int iupSetFontAttrib (Ihandle *ih, const char *value)
int iupFontParsePango (const char *value, char *typeface, int *size, int *bold, int *italic, int *underline, int *strikeout)
int iupFontParseWin (const char *value, char *fontname, int *height, int *bold, int *italic, int *underline, int *strikeout)
int iupFontParseX (const char *value, char *fontname, int *height, int *bold, int *italic, int *underline, int *strikeout)
```

Function Documentation

```
void iupdrvGetCharSize( Ihandle * ih,
                       int * charwidth,
                       int * charheight
                       )
```

Retrieve the character size for the selected font.

```
void iupdrvStringSize( Ihandle * ih,
                       const char * str,
                       int * w,
                       int * h
                       )
```

Retrieve the string size for the selected font.

```
void iupdrvMultiLineStringSize( Ihandle * ih,
                                const char * str,
                                int * w,
                                int * h
                                )
```

Retrieve the multi-lined string size for the selected font.

```
char* iupdrvGetSystemFont( void )
Returns the System default font.
```

```
int iupdrvSetNativeFontAttrib( Ihandle * ih,
                              const char * value
                              )
NATIVEFONT attribute set function.
```

```
char* iupGetFontAttrib( Ihandle * ih )
FONT attribute get function.
```

```
int iupSetFontAttrib( Ihandle * ih,
                     const char * value
                     )
FONT attribute set function.
```

```
int iupFontParsePango( const char * value,
                      char *      typeface,
                      int *        size,
                      int *        bold,
                      int *        italic,
                      int *        underline,
                      int *        strikeout
                      )
```

Parse the common font format description. Returns a non zero value if successful.

```
int iupFontParseWin( const char * value,
                    char *      fontname,
                    int *        height,
                    int *        bold,
                    int *        italic,
                    int *        underline,
                    int *        strikeout
                    )
```

Parse the old IUP Windows font format description. Returns a non zero value if successful.

```
int iupFontParseX( const char * value,
                  char *      fontname,
                  int *        height,
                  int *        bold,
                  int *        italic,
                  int *        underline,
                  int *        strikeout
                  )
```

Parse the X-Windows font format description. Returns a non zero value if successful.

Driver Information Interface

[\[Driver Interface\]](#)

Detailed Description

Each driver must export the symbols defined here. But in this case the functions are shared by different drivers in the same system.

For example, the GTK driver and the Windows driver share the same implementation of these functions when the GTK driver is compiled in Windows. The GTK driver and the Motif driver share the same implementation of these functions when the GTK driver is compiled in UNIX.

See [iup_drvinfo.h](#)

Functions

```
void iupdrvGetFullSize( int *width, int *height)
void iupdrvGetScreenSize( int *width, int *height)
int iupdrvGetScreenDepth( void)
char * iupdrvGetSystemVersion( void)
char * iupdrvGetSystemName( void)
char * iupdrvGetComputerName( void)
char * iupdrvGetUserName( void)
void iupdrvGetKeyState( char *key)
void iupdrvGetCursorPos( int *x, int *y)
void * iupdrvGetDisplay( void)
int iupdrvGetWindowDecor( void *wnd, int *border, int *caption)
char * iupdrvGetCurrentDirectory( void)
int iupdrvSetCurrentDirectory( const char *dir)
int iupdrvIsFile( const char *name)
int iupdrvIsDirectory( const char *name)
int iupdrvMakeDirectory( const char *name)
```

Function Documentation

```
void iupdrvGetFullSize( int * width,
                      int * height
                      )
```

Retrieve the main desktop full size.

```
void iupdrvGetScreenSize( int * width,
                        int * height
                        )
```

Retrieve the main desktop available size.

```
int iupdrvGetScreenDepth( void )
```

Retrieve the default desktop bits per pixel.

```
char* iupdrvGetSystemVersion( void )
```

Returns a string with the system version number.

char* iupdrvGetSystemName(void)

Returns a string with the system name.

char* iupdrvGetComputerName(void)

Returns a string with the computer name.

char* iupdrvGetUserName(void)

Returns a string with the user name.

void iupdrvGetKeyState(char * *key*)

Returns the key state for Shift, Ctrl, Alt and sYs, in this order. Left and right keys are considered. Should declare "char key[5]". Values could be space (" ") or "SCAY".

void iupdrvGetCursorPos(int * *x*,
int * *y*
)

Returns the current position of the mouse cursor.

void* iupdrvGetDisplay(void)

Returns the driver "Display" in UNIX and NULL in Windows. Must be implemented somewhere else.

int iupdrvGetWindowDecor(void * *wnd*,
int * *border*,
int * *caption*
)

Returns the decoration size of the native window. In Windows will also includes the menu if any. Used in iupdrvDialogGetDecoration only.

char* iupdrvGetCurrentDirectory(void)

Returns the current directory.

int iupdrvSetCurrentDirectory(const char * *dir*)

Changes the current directory.

int iupdrvIsFile(const char * *name*)

Returns true if the given name is an existant file.

int iupdrvIsDirectory(const char * *name*)

Returns true if the given name is an existant directory.

int iupdrvMakeDirectory(const char * *name*)

Creates a new direcotry.

Utilities

Modules

[Simple Array](#)

[Assert Utilities](#)

[String Utilities](#)

[Language Dependent String Messages](#)

[Hash Table](#)

Simple Array

[\[Utilities\]](#)

Detailed Description

Expandable array using a simple pointer.

See [iup_array.h](#)

Functions

Iarray * [iupArrayCreate](#) (int start_count, int elem_size)

void [iupArrayDestroy](#) (Iarray *array)

void * [iupArrayGetData](#) (Iarray *array)

void * [iupArrayInc](#) (Iarray *array)

void * [iupArrayAdd](#) (Iarray *array, int new_count)

int [iupArrayCount](#) (Iarray *array)

Function Documentation

```

Iarray* iupArrayCreate( int start_count,
                      int elem_size
                      )

```

Creates an array with an initial room for elements, and the element size. The array count starts at 0. And the maximum number of elements starts at the given count. The maximum number of elements is increased by the start count, every time it needs more memory. Must call [iupArrayInc](#) to proper allocates memory.

```

void iupArrayDestroy( Iarray * array )
    Destroys the array.

```

```

void* iupArrayGetData( Iarray * array )
    Returns the pointer that contains the array.

```

```

void* iupArrayInc( Iarray * array )
    Increments the number of elements in the array. The array count starts at 0. If the maximum number of elements is reached, the memory allocated is increased by the initial start count. Returns the pointer that contains the array.

```

```

void* iupArrayAdd( Iarray * array,
                  int    new_count
                  )

```

Increments the number of elements in the array by a given count. The array count starts at 0. If the maximum number of elements is reached, the memory allocated is increased by the given count. Returns the pointer that contains the array.

```

int iupArrayCount( Iarray * array )
    Returns the actual number of elements in the array.

```

Assert Utilities

[\[Utilities\]](#)

Detailed Description

All functions of the main API (Iup***) calls iupASSERT to check the parameters.

The IUP main library must be recompiled with the IUP_ASSERT define to enable these checks. iupASSERT is not called inside driver dependent functions nor in each control implementation, it is used only in the functions of the main API and in some utilities.

See [iup_assert.h](#)

Defines

```

#define iupASSERT(_expr)  ((_expr)? (void)0: iupAssert(#_expr, __FILE__, __LINE__, NULL))
#define iupERROR(_msg)    iupError(_msg)

```

Define Documentation

```

#define iupASSERT(_expr )  ((_expr)? (void)0: iupAssert(#_expr, __FILE__, __LINE__, NULL))

```

If the expression if false, displays a message with information of the source code where the assert happen.

Parameters:

_expr The evaluated expression.

It is a macro that calls a function only if IUP_ASSERT is defined.

```

#define iupERROR(_msg )    iupError(_msg)

```

Displays an error message. Also used by the iupASSERT.

It is a macro that calls a function only if IUP_ASSERT is defined.

String Utilities

[\[Utilities\]](#)

Detailed Description

See [iup_str.h](#)

Functions

```

char * iupStrDup (const char *str)
int iupStrEqual (const char *str1, const char *str2)
int iupStrEqualNoCase (const char *str1, const char *str2)
int iupStrEqualPartial (const char *str1, const char *str2)
int iupStrBoolean (const char *str)
void iupStrLnCount (const char *str, int *max_len, int *num_lin)

```

```

const char * iupStrNextLine( const char *str, int *len)
    int iupStrCountChar( const char *str, int c)
char * iupStrCopyUntil( char **str, int c)
char * iupStrGetMemory( int size)
void iupStrLower( char *dstr, const char *sstr)
    int iupStrGetRGB( const char *str, unsigned char *r, unsigned char *g, unsigned char *b)
    int iupStrToInt( const char *str, int *i)
    int iupStrToIntInt( const char *str, int *i1, int *i2, char sep)
    int iupStrToFloat( const char *str, float *f)
    int iupStrToFloatFloat( const char *str, float *f1, float *f2, char sep)
    int iupStrToStrStr( const char *str, char *str1, char *str2, char sep)
char * iupStrFileGetExt( const char *file_name)
char * iupStrFileGetTitle( const char *file_name)
char * iupStrFileGetPath( const char *file_name)
char * iupStrFileMakeFileName( const char *path, const char *title)

```

Function Documentation

char* iupStrDup(const char * *str*)

Returns a copy of the given string. If str is NULL it will return NULL.

int iupStrEqual(const char * *str1*,
const char * *str2*
)

Returns a non zero value if the two strings are equal. str1 or str2 can be NULL.

int iupStrEqualNoCase(const char * *str1*,
const char * *str2*
)

Returns a non zero value if the two strings are equal but ignores case. str1 or str2 can be NULL.

int iupStrEqualPartial(const char * *str1*,
const char * *str2*
)

Returns a non zero value if the two strings are equal up to a number of characters defined by the strlen of the second string. str1 or str2 can be NULL.

int iupStrBoolean(const char * *str*)

Returns 1 if the string is "1", "YES", "ON" or "TRUE".

Returns 0 if the string is "0", "NO", "OFF" or "FALSE", or the string is NULL or empty.

void iupStrLineCount(const char * *str*,
int * *max_len*,
int * *num_lin*
)

Returns the number of lines and the size of the largest line in a string. Detects also CR/LF pairs.

const char* iupStrNextLine(const char * *str*,
int * *len*
)

Returns the a pointer to the next line and the size of the current line. Detects also CR/LF pairs. If str is NULL it will return NULL.

int iupStrCountChar(const char * *str*,
int *c*
)

Returns the number of repetitions of the character occurs in the string.

char* iupStrCopyUntil(char ** *str*,
int *c*
)

Returns a new string containing a copy of the string up to the character. The string is then incremented to after the position of the character.

char* iupStrGetMemory(int *size*)

Returns a buffer with the specified size+1.

The buffer is resused after 10 calls. It must NOT be freed. Use size=-1 to free all the internal buffers.

void iupStrLower(char * *dstr*,
const char * *sstr*
)

Converts a string into lower case.

int iupStrGetRGB(const char * *str*,
unsigned char * *r*,
unsigned char * *g*,
unsigned char * *b*
)

Extract a RGB triple from the string.

int iupStrToInt(const char * *str*,
int * *i*
)

)

Converts the string to an int. The string must contains only the integer value. Returns a a non zero value if sucessfull.

```
int iupStrToIntInt( const char * str,
                    int *      i1,
                    int *      i2,
                    char        sep
                  )
```

Converts the string to two int. The string must contains two integer values in sequence, separated by the given character (usually 'x' or ':'). Returns the number of converted values. Values not extracted are not changed.

```
int iupStrToFloat( const char * str,
                   float *      f
                 )
```

Converts the string to an float. The string must contains only the real value. Returns a a non zero value if sucessfull.

```
int iupStrToFloatFloat( const char * str,
                        float *      f1,
                        float *      f2,
                        char          sep
                      )
```

Converts the string to two float. The string must contains two real values in sequence, separated by the given character (usually 'x' or ':'). Returns the number of converted values. Values not extracted are not changed.

```
int iupStrToStrStr( const char * str,
                    char *      str1,
                    char *      str2,
                    char        sep
                  )
```

Extract two strings from the string. separated by the given character (usually 'x' or ':'). Returns the number of converted values. Values not extracted are not changed.

```
char* iupStrFileGetExt( const char * file_name )
```

Returns the file extension of a file name.

```
char* iupStrFileGetTitle( const char * file_name )
```

Returns the file title of a file name.

```
char* iupStrFileGetPath( const char * file_name )
```

Returns the file path of a file name.

```
char* iupStrFileMakeFileName( const char * path,
                              const char * title
                            )
```

Concat path and title addind '/' between if path does not have it.

Language Dependent String Messages

[\[Utilities\]](#)

Detailed Description

String database that is dependend of the selected language.

See [iup_strmessage.h](#)

Functions

```
void iupStrMessageShowError (Ihandle *parent, const char *message)
char * iupStrMessageGet (const char *message)
```

Function Documentation

```
void iupStrMessageShowError( Ihandle * parent,
                             const char * message
                           )
```

Pre-defined dialog to show an error message. Based in IupMessageDlg. Message can be a registered coded message or a commom string.

```
char* iupStrMessageGet( const char * message )
```

Returns a common string from a registered coded message. The returned string depends on the global LANGUAGE attribute.

Hash Table

[\[Utilities\]](#)

Detailed Description

The hash table can be indexed by strings or pointer address, and each value can contain strings, pointers or function pointers.

See [iup_table.h](#)

Enumerations

```
enum Itable\_IndexTypes { IUPTABLE\_POINTERINDEXED = 10, IUPTABLE\_STRINGINDEXED }
enum Itable\_Types { IUPTABLE\_POINTER, IUPTABLE\_STRING, IUPTABLE\_FUNCPOINTER }
```

Functions

```
Itable * iupTableCreate (enum Itable\_IndexTypes indexType)
Itable * iupTableCreateSized (enum Itable\_IndexTypes indexType, unsigned int initialSizeIndex)
void iupTableDestroy (Itable *n)
int iupTableCount (Itable *it)
void iupTableSet (Itable *n, const char *key, void *value, enum Itable\_Types itemType)
void iupTableSetFunc (Itable *n, const char *key, Ifunc func)
void * iupTableGet (Itable *n, const char *key)
Ifunc iupTableGetFunc (Itable *n, const char *key, void **value)
void * iupTableGetTyped (Itable *n, const char *key, enum Itable\_Types *itemType)
void iupTableRemove (Itable *n, const char *key)
char * iupTableFirst (Itable *it)
char * iupTableNext (Itable *it)
void * iupTableGetCurr (Itable *it)
char * iupTableRemoveCurr (Itable *it)
```

Enumeration Type Documentation

enum [Itable_IndexTypes](#)

How the table key is interpreted.

Enumerator:

IUPTABLE_POINTERINDEXED a pointer address is used as key.
IUPTABLE_STRINGINDEXED a string as key

enum [Itable_Types](#)

How the value is interpreted.

Enumerator:

IUPTABLE_POINTER regular pointer for strings and other pointers
IUPTABLE_STRING string duplicated internally
IUPTABLE_FUNCPOINTER function pointer

Function Documentation

Itable* [iupTableCreate](#)(enum [Itable_IndexTypes](#) *indexType*)

Creates a hash table with an initial default size. This function is equivalent to [iupTableCreateSized](#)(0);

Itable* [iupTableCreateSized](#)(enum [Itable_IndexTypes](#) *indexType*,
 unsigned int *initialSizeIndex*
)

Creates a hash table with the specified initial size. Use this function if you expect the table to become very large. *initialSizeIndex* is an array into the (internal) list of possible hash table sizes. Currently only indexes from 0 to 8 are supported. If you specify a higher value here, the maximum allowed value will be used.

void [iupTableDestroy](#)(Itable * *n*)

Destroys the Itable. This function does also free the memory of strings contained in the table!!!!

int [iupTableCount](#)(Itable * *it*)

Returns the number of keys stored in the table.

```
void iupTableSet( Itable *      n,
                  const char * key,
                  void *      value,
                  enum Itable\_Types itemType
                  )
```

Store an element in the table.

```
void iupTableSetFunc( Itable * n,
                    const char * key,
                    Ifunc func
                    )
```

Store a function pointer in the table. Type is set to IUPTABLE_FUNCPOINTER.

```
void* iupTableGet( Itable * n,
                  const char * key
                  )
```

Retrieves an element from the table. Returns NULL if not found.

```

Ifunc iupTableGetFunc( Itable *    n,
                      const char * key,
                      void **      value
                    )

```

Retrieves a function pointer from the table. If not a function or not found returns NULL. value always contains the element pointer.

```

void* iupTableGetTyped( Itable *    n,
                      const char * key,
                      enum Itable_Types * itemType
                    )

```

Retrieves an element from the table and its type.

```

void iupTableRemove( Itable *    n,
                    const char * key
                  )

```

Removes the entry at the specified key from the hash table and frees the memory used by it if it is a string...

```

char* iupTableFirst( Itable * it )

```

Key iteration function. Returns a key. To iterate over all keys call iupTableFirst at the first and call iupTableNext in a loop until 0 is returned... Do NOT change the content of the hash table during iteration. During an iteration you can use context with [iupTableGetCurr\(\)](#) to access the value of the key very fast.

```

char* iupTableNext( Itable * it )

```

Key iteration function. See [iupTableNext](#).

```

void* iupTableGetCurr( Itable * it )

```

Returns the value at the current position. The current context is an iterator that is filled by [iupTableNext\(\)](#). iupTableGetCurr() is faster than [iupTableGet\(\)](#), so when you want to access an item stored at a key returned by [iupTableNext\(\)](#), use this function instead of [iupTableGet\(\)](#).

```

char* iupTableRemoveCurr( Itable * it )

```

Removes the current element and returns the next key. Use this function to remove an element during an iteration.

- [All](#)
- [Functions](#)
- [Typedefs](#)
- [Enumerations](#)
- [Enumerator](#)
- [Defines](#)

- [i](#)

Here is a list of all documented functions, variables, defines, enums, and typedefs with links to the documentation:

- i -

- IattribGetFunc : [iup_class.h](#)
- IattribGetIdFunc : [iup_class.h](#)
- IattribSetFunc : [iup_class.h](#)
- IattribSetIdFunc : [iup_class.h](#)
- IcontrolData : [iup_object.h](#)
- InativeHandle : [iup_object.h](#)
- Itable_IndexTypes : [iup_table.h](#)
- Itable_Types : [iup_table.h](#)
- IUP_TYPECANVAS : [iup_class.h](#)
- IUP_TYPECONTROL : [iup_class.h](#)
- IUP_TYPEVOID : [iup_class.h](#)
- iupArrayAdd() : [iup_array.h](#)
- iupArrayCount() : [iup_array.h](#)
- iupArrayCreate() : [iup_array.h](#)
- iupArrayDestroy() : [iup_array.h](#)
- iupArrayGetData() : [iup_array.h](#)
- iupArrayInc() : [iup_array.h](#)
- iupASSERT : [iup_assert.h](#)
- iupAttribGetFloat() : [iup_attrib.h](#)
- iupAttribGetFloatDefault() : [iup_attrib.h](#)
- iupAttribGetInt() : [iup_attrib.h](#)
- iupAttribGetIntDefault() : [iup_attrib.h](#)
- iupAttribGetStr() : [iup_attrib.h](#)
- iupAttribGetStrDefault() : [iup_attrib.h](#)
- iupAttribIsInheritable() : [iup_attrib.h](#)
- iupAttribIsInternal() : [iup_attrib.h](#)
- iupAttribIsPointer() : [iup_attrib.h](#)
- iupAttribSetFloat() : [iup_attrib.h](#)
- iupAttribSetInt() : [iup_attrib.h](#)
- iupAttribSetStr() : [iup_attrib.h](#)
- iupAttribSetStrf() : [iup_attrib.h](#)
- iupAttribStoreStr() : [iup_attrib.h](#)
- iupBaseSetCurrentSizeMethod() : [iup_layout.h](#)
- iupBaseSetPositionMethod() : [iup_layout.h](#)
- iupCallGetFocusCb() : [iup_focus.h](#)
- iupCallKillFocusCb() : [iup_focus.h](#)
- iupClassCallbackGetFormat() : [iup_class.h](#)
- iupClassGetBase() : [iup_class.h](#)
- iupClassRegisterAttribute() : [iup_class.h](#)
- iupClassRegisterCallback() : [iup_class.h](#)
- iupClassRelease() : [iup_class.h](#)

- [iupDlgListAdd\(\) : iup_dlglist.h](#)
- [iupDlgListFirst\(\) : iup_dlglist.h](#)
- [iupDlgListNext\(\) : iup_dlglist.h](#)
- [iupDlgListRemove\(\) : iup_dlglist.h](#)
- [iupDlgListVisibleCount\(\) : iup_dlglist.h](#)
- [iupDlgListVisibleDec\(\) : iup_dlglist.h](#)
- [iupDlgListVisibleInc\(\) : iup_dlglist.h](#)
- [iupdrvDisplayUpdate\(\) : iup_drv.h](#)
- [iupdrvDrawFocusRect\(\) : iup_drv.h](#)
- [iupdrvGetCharSize\(\) : iup_drvfont.h](#)
- [iupdrvGetComputerName\(\) : iup_drvinfo.h](#)
- [iupdrvGetCurrentDirectory\(\) : iup_drvinfo.h](#)
- [iupdrvGetCursorPos\(\) : iup_drvinfo.h](#)
- [iupdrvGetDisplay\(\) : iup_drvinfo.h](#)
- [iupdrvGetFullSize\(\) : iup_drvinfo.h](#)
- [iupdrvGetGlobal\(\) : iup_drv.h](#)
- [iupdrvGetKeyState\(\) : iup_drvinfo.h](#)
- [iupdrvGetScreenDepth\(\) : iup_drvinfo.h](#)
- [iupdrvGetScreenSize\(\) : iup_drvinfo.h](#)
- [iupdrvGetSystemFont\(\) : iup_drvfont.h](#)
- [iupdrvGetSystemName\(\) : iup_drvinfo.h](#)
- [iupdrvGetSystemVersion\(\) : iup_drvinfo.h](#)
- [iupdrvGetUserName\(\) : iup_drvinfo.h](#)
- [iupdrvGetWindowDecor\(\) : iup_drvinfo.h](#)
- [iupdrvIsActive\(\) : iup_drv.h](#)
- [iupdrvIsDirectory\(\) : iup_drvinfo.h](#)
- [iupdrvIsFile\(\) : iup_drvinfo.h](#)
- [iupdrvIsVisible\(\) : iup_drv.h](#)
- [iupdrvMakeDirectory\(\) : iup_drvinfo.h](#)
- [iupdrvMultiLineStringSize\(\) : iup_drvfont.h](#)
- [iupdrvReparent\(\) : iup_drv.h](#)
- [iupdrvScreenToClient\(\) : iup_drv.h](#)
- [iupdrvSetActive\(\) : iup_drv.h](#)
- [iupdrvSetCurrentDirectory\(\) : iup_drvinfo.h](#)
- [iupdrvSetGlobal\(\) : iup_drv.h](#)
- [iupdrvSetIdleFunction\(\) : iup_drv.h](#)
- [iupdrvSetNativeFontAttrib\(\) : iup_drvfont.h](#)
- [iupdrvSetVisible\(\) : iup_drv.h](#)
- [iupdrvStringSize\(\) : iup_drvfont.h](#)
- [iupERROR : iup_assert.h](#)
- [iupFocusCanAccept\(\) : iup_focus.h](#)
- [iupFontParsePango\(\) : iup_drvfont.h](#)
- [iupFontParseWin\(\) : iup_drvfont.h](#)
- [iupFontParseX\(\) : iup_drvfont.h](#)
- [iupGetFontAttrib\(\) : iup_drvfont.h](#)
- [iupGetFontFaceAttrib\(\) : iup_drvfont.h](#)
- [iupGetFontSizeAttrib\(\) : iup_drvfont.h](#)
- [iupGetFontStyleAttrib\(\) : iup_drvfont.h](#)
- [iupGetParamList\(\) : iup_object.h](#)
- [iupHandleCheck\(\) : iup_object.h](#)
- [iupKeyCallKeyCb\(\) : iup_key.h](#)
- [iupKeyCallKeyPressCb\(\) : iup_key.h](#)
- [iupKeyCanCaps\(\) : iup_key.h](#)
- [iupKeyCodeToName\(\) : iup_key.h](#)
- [iupKeyForEach\(\) : iup_key.h](#)
- [iupKeyNameToCode\(\) : iup_key.h](#)
- [iupLayoutSetExpand\(\) : iup_layout.h](#)
- [iupRegisterClass\(\) : iup_register.h](#)
- [iupRegisterFindClass\(\) : iup_register.h](#)
- [iupSetFontAttrib\(\) : iup_drvfont.h](#)
- [iupSetFontSizeAttrib\(\) : iup_drvfont.h](#)
- [iupSetFontStyleAttrib\(\) : iup_drvfont.h](#)
- [iupStrBoolean\(\) : iup_str.h](#)
- [iupStrCopyUntil\(\) : iup_str.h](#)
- [iupStrCountChar\(\) : iup_str.h](#)
- [iupStrDup\(\) : iup_str.h](#)
- [iupStrEqual\(\) : iup_str.h](#)
- [iupStrEqualNoCase\(\) : iup_str.h](#)
- [iupStrEqualPartial\(\) : iup_str.h](#)
- [iupStrFileGetExt\(\) : iup_str.h](#)
- [iupStrFileGetPath\(\) : iup_str.h](#)
- [iupStrFileGetTitle\(\) : iup_str.h](#)
- [iupStrFileMakeFileName\(\) : iup_str.h](#)
- [iupStrGetMemory\(\) : iup_str.h](#)
- [iupStrGetRGB\(\) : iup_str.h](#)
- [iupStrLineCount\(\) : iup_str.h](#)
- [iupStrLower\(\) : iup_str.h](#)
- [iupStrMessageGet\(\) : iup_strmessage.h](#)
- [iupStrMessageShowError\(\) : iup_strmessage.h](#)
- [iupStrNextLine\(\) : iup_str.h](#)
- [iupStrToFloat\(\) : iup_str.h](#)
- [iupStrToFloatFloat\(\) : iup_str.h](#)
- [iupStrToInt\(\) : iup_str.h](#)
- [iupStrToIntInt\(\) : iup_str.h](#)
- [iupStrToStrStr\(\) : iup_str.h](#)
- [IUPTABLE_FUNCPOINTER : iup_table.h](#)
- [IUPTABLE_POINTER : iup_table.h](#)
- [IUPTABLE_POINTERINDEXED : iup_table.h](#)
- [IUPTABLE_STRING : iup_table.h](#)
- [IUPTABLE_STRINGINDEXED : iup_table.h](#)
- [iupTableCount\(\) : iup_table.h](#)

- iupTableCreate() : [iup_table.h](#)
- iupTableCreateSized() : [iup_table.h](#)
- iupTableDestroy() : [iup_table.h](#)
- iupTableFirst() : [iup_table.h](#)
- iupTableGet() : [iup_table.h](#)
- iupTableGetCurr() : [iup_table.h](#)
- iupTableGetFunc() : [iup_table.h](#)
- iupTableGetTyped() : [iup_table.h](#)
- iupTableNext() : [iup_table.h](#)
- iupTableRemove() : [iup_table.h](#)
- iupTableRemoveCurr() : [iup_table.h](#)
- iupTableSet() : [iup_table.h](#)
- iupTableSetFunc() : [iup_table.h](#)