IUP
Portable User Interface
Version 3.0

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 Windows.

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

Currently available interface elements can be categorized as follows:

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

Hence IUP has some advantages over other interface toolkits available:

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

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

Availability

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

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

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

- UNIX (SunOS, IRIX, and AIX) using Motif 2.x
- UNIX (FreeBSD and Linux) using GTK+ (since 3.0)
- Microsoft Windows 2000/XP/2003/Vista/7 using the Win32 API

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 the **IUP** support features at **SourceForge**: http://sourceforge.net/projects/iup/. It provides us Mailing List, CVS Repository and Downloads.

The discussion list is available at: http://lists.sourceforge.net/lists/listinfo/iup-users.
Source code, pre-compiled binaries and documentation can be downloaded at: http://sourceforge.net/project/showfiles.php?group_id=241310.
The CVS can be browsed at: http://iup.cvs.sourceforge.net/iup/.

If you want us to develop a specific feature for the toolkit, Tecgraf is available for partnerships and cooperation. Please contact **tcg@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.

Library Authors:

- Marcelo Gattass
- Luiz Henrique de Figueiredo
- Carlos Henrique Levy
- Antonio Scuri

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

Thanks to the people that worked and contributed to the library:

- André Carregal
- André Clinio
- André Costa
- André Derraik
- Carlos Augusto Mendes
- Carlos José Pereira de Lucena
- Claudio Coutinho de Biasi
- Danny Reinhold
- Diego Nehab
- Diogo Martinez
- Guilherme Fonseca Alvarenga
- Henrique Dalcin Mendes Pinheiro
- Leonardo Constantino Oliveira
- Luiz Cristóvão Gomes Coelho
- Luiz Martins
- Marian Trifon
- Mark Stroetzel Glasberg
- Mauricio Oliveira Carneiro
- Milton Jonathan
- Neil Armstrong Rezende
- Otfried Cheong
- Rafael Rieder
- Renato Borges
- Renato Cerqueira
- Roberto Beauclair
- Steve Donovan
- Tomas Guisasola Gorham
- Vinicius Almendra
- Warren Music

Thanks for the SourceForge for hosting the support features. Thanks for the LuaForge team for previously hosting the support features for many years.

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] [iup_sdj2005.pdf]
- 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 Manipulção Direta". Proceedings of VII SIBGRAPI (1994), 165-172. [sib94.pdf in Portuguese]
- 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 in Portuguese]
- Levy, C. H. "IUP/LED: Uma Ferramenta Portátil de Interface com Usuário". M.Sc. dissertation, Computer Science Department, PUC-Rio, 1993.[levy93.pdf in Portuguese]
- 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 in Portuguese]

Tecgraf Library License

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

Download

The download site for pre-compiled binaries, documentation and sources is at **SourceForge**:

http://sourceforge.net/projects/iup/files/

Use this link for the latest version: http://sourceforge.net/projects/iup/files/3.0RC4a

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://sourceforge.net/project/showfiles.php?group_id=241318.
The **CD** files can be downloaded at http://sourceforge.net/project/showfiles.php?group_id=241317.
The **IUP** files can be downloaded at http://sourceforge.net/project/showfiles.php?group_id=241310.
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. Since 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 compiler used is always gcc. Binaries for 64-bits receive the suffix: "_64". In Linux when there are different versions of gcc for the same uname, the platform name is created adding the major version number of the compiler 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 **(*_lib*)** contains pre-compiled 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_Linux26_lib.tar.gz = IUP 2.4 32-bits Libraries and Includes for Linux with Kernel version 2.6 built with gcc 3.
iup2_4_Linux26g4_64_bin.tar.gz = IUP 2.4 64-bits Executables for Linux with Kernel version 2.6 built with gcc 4.
iup2_4_Win32_vc8_lib.tar.gz = IUP 2.4 32-bits Static Libraries and Includes for Windows to use with Visual C++ 8 (2005).
iup2_4_Win32_dll9_lib.tar.gz = IUP 2.4 32-bits Dynamic Libraries (DLLs), stub libraries and Includes for Windows to use with Visual C++ 9 (2008).
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 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\Linux26g4_64
            lib\vc8
            src
        cd\
```

```
im\
lua5.1\
```

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:

Borland C++ BuilderX - http://www.tecgraf.puc-rio.br/iup/en/ide_guide/cppbx.html
Code Blocks - http://www.tecgraf.puc-rio.br/iup/en/ide_guide/codeblocks.html
Dev-C++ - http://www.tecgraf.puc-rio.br/iup/en/ide_guide/dev-cpp.html
Eclipse for C++ - http://www.tecgraf.puc-rio.br/iup/en/ide_guide/eclipse.html
Microsoft Visual C++ (Visual Studio 2003) - http://www.tecgraf.puc-rio.br/iup/en/ide_guide/msvc.html
Microsoft Visual C++ (Visual Studio 2005) - http://www.tecgraf.puc-rio.br/iup/en/ide_guide/msvc8.html
Open Watcom - http://www.tecgraf.puc-rio.br/iup/en/ide_guide/owc.html

Available Platforms

The following platforms can be available:

AIX43	IBM AIX 4.3 (ppc) / gcc 2.95 / 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
Linux24	Red Hat 7.3 (x86) / Kernel 2.4 / gcc 2.95 / Open Motif 2.1 / GTK 2.0
Linux24g3	CentOS 3.9 (x86) / Kernel 2.4 / gcc 3.2 / Open Motif 2.2 [3] / GTK 2.2
Linux24g3_64	Red Hat E.L. WS 3 (x64) / Kernel 2.4 / gcc 3.2 / Open Motif 2.2 [3] / GTK 2.2
Linux26	CentOS 4.6 (x86) / Kernel 2.6 / gcc 3.4 / Open Motif 2.2 [3] / GTK 2.4
Linux26_64	CentOS 4.6 (x64) / Kernel 2.6 / gcc 3.4 / Open Motif 2.2 [3] / GTK 2.4
Linux26g4	CentOS 5.3 (x86) / Kernel 2.6 / gcc 4.1 / Open Motif 2.2 [3] / GTK 2.12
Linux26g4_64	CentOS 5.2 (x64) / Kernel 2.6 / gcc 4.1 / OpenMotif 2.2 [3] / GTK 2.10
Linux26g4ppc	Ubuntu 7.10 (ppc) / Kernel 2.6 / gcc 4.1 / Open Motif 2.2 [3] / GTK 2.?
Linux26_ia64	Red Hat E.L. AS 4 (ia64) / Kernel 2.6 / gcc 3.4 / Open Motif 2.2 [3] / GTK 2.4
SunOS57	Sun Solaris 7 (sparc) / gcc 2.95 / Motif 2.1
SunOS58	Sun Solaris 8 (sparc) / gcc 3.4 / 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 [3]
Darwin811	Mac OS X 10.4.11 (ppc) [Tiger] / Darwin Kernel 8.11 / gcc 4.0 / Open Motif 2.1
Darwin811x86	Mac OS X 10.4.11 (x86) [Tiger] / Darwin Kernel 8.11 / gcc 4.0 / Open Motif 2.1
Darwin94x86	Mac OS X 10.5.4 (x86) [Leopard] / Darwin Kernel 9.4 / gcc 4.0 / Open Motif 2.1
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/ [1]
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/ [1]
Win32_vc9	Microsoft Visual C++ 9.0 (2008) (static RTL/multithread) Also compatible with Microsoft Visual C++ 2008 Express Edition - http://msdn.microsoft.com/vstudio/express/visualc/ [1]
Win32_dll6	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).
Win32_dll9	built using vc9, creates dependency with MSVCR90.DLL (either other libraries or new applications).
Win64_vc8	Same as **Win32_vc8** but for 64-bits systems using x64 standard.
Win64_vc9	Same as **Win32_vc9** but for 64-bits systems using x64 standard.
Win64_dll8	Same as **Win32_dll8** but for 64-bits systems using x64 standard.
Win64_dll9	Same as **Win32_dll9** but for 64-bits systems using x64 standard.
Win32_gcc3	Cygwin gcc 3.4 (Depends on Cygwin DLL 1.5) - http://www.cygwin.com/ [1]
Win32_cygw15	Same as **Win32_gcc3**, but using the Cygwin Posix system
Win32_mingw3	MingW gcc 3.4 - http://www.mingw.org/ [1] Also compatible with Dev-C++ - http://www.bloodshed.net/devcpp.html and with Code Blocks - http://www.codeblocks.org/ [1]
Win32_mingw4	MingW gcc 4.4 - http://www.mingw.org/ [1]
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 [1,2]

	(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

[1] - Notice that all the Windows compilers with links here are free to download and use.
[2] - 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.
[3] - OpenMotif 2.2 is classified as 'experimental' by the Open Group.

CVS

The CVS repository is at **SourceForge**. It can also be interactively browsed at:

http://iup.cvs.sourceforge.net/iup/

To checkout use the module name "iup" and the CVSROOT:

```
:pserver:anonymous@iup.cvs.sourceforge.net:/cvsroot/iup
```

To checkout the IUP 2.x source code use the module name "iup2".

History of Changes

Version 3.0

See Version 3.0 History.

Version 2.x

See Version 2.x History.

Version 1.x

See Version 1.x History.

History of Changes in Version 3.x

Check the Migration Guide for a summary of the important changes and how to proceed when migrating from version 2.x to version 3.x.

Version 3.0 RC 4a (18/Dec/2009)

- Fixed: VISIBLE attribute management. **IupZbox** now will respect if a child has a VISIBLE attribute set, and it will not change it. **IupTabs** now does not depends on the VISIBLE attribute anymore.
- Fixed: VALUE attribute return in **IupItem** on GTK.
- Fixed: **IupList** with DROPDOWN=Yes and the last item is removed.

Version 3.0 RC 4 (14/Dec/2009)

- New: NMARGIN and NGAP non-inheritable attributes for **IupHbox** and **IupVbox**.
- New: "OTHER" status code for FILE_CB when selecting an invalid file name or a directory in **IupFileDlg**.
- New: DLL_HINSTANCE global attribute in Windows.
- Changed: Added a workaround for TITLEFONTid for **IupTree** when changing only the Bold style in Windows.
- Changed: the RENAMENODE_CB callback in **IupTree** is not supported anymore.
- Changed: improved compatibility of **IupFileDlg** when DIALOGTYPE=DIR and CoInitializeEx was initialized with COINIT_MULTITHREADED prior to **IupOpen** in Windows.
- Changed: **IupFrame** can now has a color background when not using TITLE, and BGCOLOR is set before map.
- Fixed: memory leak in **IupPPlot**.
- Fixed: invalid memory access in set ALIGNMENT attribute for **IupLabel**, **IupButton** and **IupToggle**, and in set MARK for **IupTree**.
- Fixed: invalid layout computation when using the old **IupSpin** element.
- Fixed: STATE attribute for **IupTree** in Windows when branch has no child.
- Fixed: invalid redraw of some controls when dialog is resized in Windows.
- Fixed: invalid memory access for SYSTEMVERSION global attribute in Linux. Thanks to David Given.
- Fixed: missing convertion to UTF-8 in **IupButton** when handling TITLE at map in the GTK driver.
- Fixed: image branch update when branch STATE is changed in **IupTree** in Windows.
- Fixed: SHOWRENAME_CB callback when renaming is started clicking twice in **IupTree**.
- Fixed: invalid limit check in VALUE attribute of **IupList** in the GTK driver. Thanks to Paul Gregory.
- Fixed: invalid memory access when setting VALUE to NULL in **IupTree**.
- Fixed: ACTION callback called when an item is set on a **IupList** when DROPDOWN=Yes.
- Fixed: dialog decoration size when menu is associated during the map process.
- Fixed: K_ANY callback called twice for **IupTabs** in the GTK driver.
- Fixed: invalid memory access when destroying some of the additional controls that use CD.
- Fixed: incomplete redraw of the **IupCanvas** in Windows XP when a window moves over the canvas.
- Fixed: missing call to ACTION when an item that was replaced is clicked in **IupList**.
- Fixed: switch of a complete menu in **IupDialog** was not working.
- Fixed: button press feedback when FOCUSONCLICK=NO in **IupButton** on Windows.
- Fixed: VISIBLE attribute for non native containers. It affected **IupZbox**.
- Fixed: **IupMatrix** with EXPAND=NO was behaving as EXPAND=YES.

Version 3.0 RC 3 (02/Oct/2009)

- New: MOVE_CB callback for **IupDialog** in Windows and GTK.
- New: SPINNING attribute for **IupGetParam** when the callback is activated by a spin.
- New: KEYPRESS, KEYRELEASE and KEY global attributes.
- New: MAXSIZE and MINSIZE attributes for all controls.
- New: NODEREMOVED_CB callback for **IupTree**.
- New: SORT attribute for **IupList**.
- New: function **IupSaveImageAsText**.

- New: function **IupLoadBuffer**.
- New: parameter in the EDITION_CB callback of **IupMatrix** to indicate if the value will be updated.
- New: auxiliary functions **IupGLUseFont** and **IupGLWait** for the **IupGLCanvas**. attribute REFRESHCONTEXT in Windows.
- **New:** VALUECHANGED_CB callback for **IupVal**, **IupDial**, **IupColorBrowser**, **IupToggle**, **IupText** and **IupList**.
- **New:** element **IupClipboard**.
- **New:** functions **IupGetNativeHandleImage** and **IupGetImageNativeHandle** for the Iup-IM library.
- Changed: now the **iup.image** constructor also accepts parameters in the same format as **iup.imagergb** and **iup.imagergba**.
- Changed: return value to boolean of **iup.GLIsCurrent, iup.GetParam, iup.SaveImage, iup.isshift, iup.iscontrol, iup.isbutton1, iup.isbutton2, iup.isbutton3, iup.isbutton4, iup.isbutton5, iup.isdouble, iup.issys, iup.isalt, iup.isSysXkey, iup.isAltXkey, iup.isCtrlXkey, iup.isShiftXkey** and **iup.isXkey** in Lua.
- Changed: the function **iup.key_open** is now obsolete and not necessary anymore.
- Changed: improved transparency for 8bpp images in Windows.
- Changed: in **IupMatrix** since the selection is made only using the mouse, by pressing a key will NOT clear the selection anymore. You can still do that setting MARKED=NULL in the K_ANY callback. Improved MARKL:C to be more flexible for other MARKMODE options.
- Changed: updated the **IupTreeUtil** contributed utility.
- Changed: CHANGEVALUE_CB callback renamed to VALUECHANGED_CB in **IupVal**.
- Changed: internal reorganization of the abstract layout methods of the Ihandle class to allow more flexibility and control of the layout process.
- Changed: LAYERED and LAYERALPHA attributes are now condensed in the OPACITY attribute. The OPACITY is available in Windows and GTK.
- Fixed: the functions **IupPreviousField** and **IupNextField** to respect the dialog hierarchy order.
- Fixed: NUMCOL and NUMLIN when set to 0 in **IupMatrix**. Double click in a title cell was entering in edit mode at the focus cell. Marks were processed after ENTERCELL_CB when the user single click a cell. Enter key processed also for the next cell when MULTIPLE=YES after editing ended.
- Fixed: STARTFOCUS in Motif and Win32 for **IupDialog** where not working. Now STARTFOCUS is set only if SHOW_CB did not changed the current focus.
- Fixed: DLGBGCOLOR in Motif where incorrectly set.
- Fixed: **IupToggle** redraw inside an **IupFrame** in Windows XP where disapearing.
- Fixed: background color of the edit box of **IupTree** in Windows XP where black.
- Fixed: release of stock images in **IupClose** caused the application to crash.
- Fixed: auxiliar function **iup.TreeSetUserId** in Lua when releasing the previous reference.
- Fixed: ACTION callback of **IupButton** in Windows when FOCUSONCLICK=NO was not being called.
- Fixed: return value of **IupSaveImage** was inverted.
- Fixed: export of image in Lua at the **IupView** application.
- Fixed: **IupGetParam** when specifying full intervals without the step parameter.
- Fixed: DEFAULTENTER and DEFAULTESC in Windows when focus is inside an **IupTabs**. Also in Windows they were processed before K_ANY, so K_ANY could not abort them by returning IUP_IGNORE.
- Fixed: K_ANY called twice for K_CR when **IupText** has multiple lines in Windows.
- Fixed: in Windows when a pre-defined system dialog was closed with Enter or Esc, the key was propagated to the dialog that open it.
- Fixed: keyboard navigation in the dialog now respects the order of **IupNextField** and **IupPreviousField** for all drivers. Those functions were also improved.
- Fixed: in GTK the VISIBLE attribute returned invalid result when child is hidden by its parent.
- Fixed: in Windows the text color of a selected item of an **IupTree** was not inverted.
- Fixed: in Windows the VALUE attribute of a inactive **IupItem** was always OFF.
- Fixed: ENTERWINDOW_CB and LEAVEWINDOW_CB for **IupCanvas** in Windows were not being called.
- Fixed: HELP_CB was not working for **IupVal**, **IupTabs** and **IupTree** in Motif.
- Fixed: USETITLESIZE attribute logic in **IupMatrix**.
- Fixed: DELNODE attribute when value is CHILDREN in **IupTree**. It was not working for the root node.

Version 3.0 RC 2 (18/Jul/2009)

- New: MONITORSINFO and VIRTUALSCREEN global attributes now also available in GTK.
- New: USETITLESIZE attribute for **IupMatrix**.
- New: DEFAULTFONTSIZE global attribute.
- New: **IupSetAtt** auxiliar function.
- Changed: the default alignment for **IupButton** (Text and Image) and IupToggle (Image) to "ACENTER:ACENTER".
- Changed: improved decoration size computation for **IupDialog** in GTK.
- Fixed: **IupItem** in GTK when compiled in versions older than 2.14, but run in newer versions.
- **Fixed:** alignment of buttons in **IupAlarm**.
- **Fixed: IupZbox** visible child management and VISILBE attribute update after mapping an element.
- **Fixed:** X and Y attributes for GTK.
- **Fixed: IupTree** TITLE with non UTF-8 characters.
- **Fixed: IupClose** in loop when removing names.
- **Fixed:** CONTEXT and VISUAL in **IupGLCanvas**.
- **Fixed:** SHOWTICKS in **IupVal**.
- **Fixed:** in **IupMatrix.** default cell alignment. BGCOLOR and FGCOLOR to use the global default colors instead of "255 255 255" and "0 0 0". drawing details. misbehavior of the scrollbar in GTK. improved IUP 2 compatibility when calling VALUE_CB and when consulting titles to compute cell size.
- **Fixed:** VALUE management in **IupZbox**.
- **Fixed:** removed "cannot add non scrollable widget" warning message when creating a **IupCanvas** in GTK.
- **Fixed:** ADDEXPANDED in **IupTree**.
- **Fixed:** SIZE consideration in layout computation for **IupDialog**.
- **Fixed:** DIALOGTYPE=MESSAGE for **IupMessageDlg** in GTK.
- **Fixed: IupButton** with no text and no image, but with BGCOLOR defined will properly show the color.

Version 3.0 RC 1 (26/Jun/2009)

General

- New: checked for memory leaks using VLD in Windows and Valgrind in Linux.
- New: PREVIEWGLCANVAS attribute for **IupFileDlg**.
- New: auxiliary functions **IupTextConvertLinColToPos** and **IupTextConvertPosToLinCol** for **IupText**.
- New: basic tutorial for IupLua. (Thanks to Steve Donovan)
- New: **IupTree** now uses native controls and was moved to the standard controls. The old implementation is not available. Images for nodes are not limited to 16x16 anymore. BGCOLOR now follows the same default as **IupText** and **IupList**, and can be changed. New TITLEFONT, FGCOLOR, USERDATA, FINDUSERID, COUNT, CHILDCOUNT, EXPANDALL, INDENTATION, HIDEBUTTONS, HIDELINES, COPYNODE, MOVENODE, SPACING, TOPITEM, INSERTLEAF and INSERTBRANCH attributes. New BUTTON_CB, MOTION_CB and DROPFILES_CB callbacks. Attributes SCROLLBAR and REDRAW are not supported anymore. VALUE attribute split in VALUE and MARK attributes, set MARK using VALUE is still possible fro backward compatibility. STARTING renamed to MARKSTART, and CTRL/SHIFT attributes replaced by MARKMODE (old names kept working for compatibility). Now if DRAGDROP_CB returns IUP_CONTINUE or if it is not defined but SHOWDRAGDROP=Yes then the node will be automatically moved to the new position. **ATTENTION -** DEPTH is now a read-only attribute, use the INSERT* attributes to properly add nodes. NAMEid attribute renamed to TITLE, old attribute still works but will be removed in future versions since it conflicts with the common NAME attribute. The SELECTION_CB and MULTISELECTION_CB callbacks now ignore their return value. The rename action is now activated by two clicks instead of a double click.
- **Changed**: removed "lua5.1.so" dependency in UNIX.
- Changed: In IupLua the Lua function **iup.TreeSetValue** now also accepts node decoration in the initialization table and can add a subtree to any node. (Thanks to Tomas Gorham)
- Changed: In IupLua attributes that are pointers to Ihandle are now returned as ihandle instead of userdata.
- Changed: replaced "[]" in function declarations by a simple "*". None of those functions needed it.

- Changed: the default value of the Windows attribute COMPOSITED is back to NO to improve backward compatibility and to avoid side effects of the attribute.
- Changed: the auxiliary functions **IupTextConvertXYToChar** and **IupListConvertXYToItem** where replaced by **IupConvertXYToPos**, that also works for **IupTree**.
- Changed: added support for WHEEL_CB in GTK for **IupCanvas**.
- Fixed: IupLua initialization when retrieving the argc/argv arguments for **IupOpen**. (Thanks to Ross Berteig)
- Fixed: Arg initialization for all controls in Motif driver.
- Fixed: update of the POSX and POSY attributes for the **IupCanvas**.
- Fixed: FONT size round when converting from pixels to points in Windows. (Thanks to Devin Smith)
- Fixed: button disappearing after mouse over in Windows XP.
- Fixed: **IupMatrix** when NUMCOL/NUMLIN were less than NUMCOL_VISIBLE/NUMLIN_VISIBLE. Also fixed when NUMCOL/NUMLIN were 0 and changed to 1, and when removed 1. CURSOR attribute when RESIZEMATRIX=Yes. (Thanks to Jeremy Cowgar)
- Fixed: action callback return value in Lua for the **IupGetParam** dialog. (Thanks to Zhiwei)
- Fixed: EXPAND attribute for **IupCanvas**.

Version 3.0 BETA 3 (04/Apr/2009)

- New: MARKL:C, READONLY, NUMLIN_VISIBLE_LAST, NUMCOL_VISIBLE_LAST, and SHOW attributes for **IupMatrix**. When scrolling the matrix using the scrollbar the focus is not changed anymore. The last cells at right and bottom are now drawn as incomplete cells if they do not fit in the visible area. New FONT_CB callback. CHECKFRAMECOLOR is not necessary anymore, just set FRAMEVERTCOLOR or FRAMEHORIZCOLOR. Internal code reorganization. AREA and MULTIPLE renamed to MARKAREA and MARKMULTIPLE, old names as still supported. New MULTILINE attribute to edit text in multiple lines, valid only before mapped.
- New: **IupRedraw** and **IupSetClassDefaultAttribute** functions.
- Changed: Added package registration code to IupLua that allows it to be statically linked and require"iuplua" does not abort if the iuplua_open function was called.
- Changed: the **IupOleControl** in Lua will not automatically initialize LuaCOM anymore. The application must manually call "elem:CreateLuaCOM()". The previous initialization was incorrect (thanks to Ross Berteig).
- Changed: the declaration of function **IupGetClassAttributes** to use the class name instead of a control handle.
- Fixed: Fixed button, toggle and list sizes for GTK driver when using the Hildon Framework. Thanks to Otfried Cheong.
- Fixed: some IupLua dynamic libraries in Linux where incorrectly linking with Motif (libiuplua + pplot, cd, controls, gl, im and imglib + 51.so)
- Fixed: HOMOGENEOUS attribute for **IupVbox** and **IupHbox**.
- Fixed: CARET attribute in GTK driver was not correctly scrolling the multiline text when not visible.
- Fixed: parameter checking and the return value in Lua for **IupListDialog** when type=2.
- Fixed: the return value for **IupGetText** when the user canceled. (Thanks to Xu Wang)
- Fixed: **IupGetClassAttributes** and **IupGetAllAttributes** were not implemented in IupLua.
- Fixed: The 32 bits version of the IupLua console in Windows XP64 was not working.
- Fixed: CARET_CB and **IupTextConvertXYToChar** in **IupText** when MULTILINE=YES and FORMATTING=NO.

Version 3.0 BETA 2 (26/Dec/2008)

- Changed: **ATTENTION -** the following headers were deprecated iupcb.h, iupcells.h, iupcolorbar.h, iupdial.h, iupgauge.h, iupmatrix.h, iuptree.h - use iupcontrols.h only
- Changed: **ATTENTION -** the following headers were deprecated iupgetparam.h, iupspin.h, iuptabs.h, iupval.h - use iup.h only
- Fixed: set VALUE attribute for IupText in Windows when formatting is used.
- Fixed: **IupHide** when dialog was maximized in Windows.
- Fixed: get VALUE attribute for **IupText** in all drivers, after the element is mapped it must return the empty string "" when there is no text.
- Fixed: **IupGetParam** when specifying partial intervals.
- Fixed: K_Esc key callback processing in Windows.
- Fixed: PLACEMENT and FULLSCREEN for IupPopup.

Version 3.0 BETA 1 (15/Dec/2008)

General

- New: GTK driver, available in UNIX and Windows.
- New: internal code reorganization. More clear and simple to create controls and drivers. All comments are now in English.
- New: internal documentation and Guide to create new controls. Now all the controls use the same architecture using the same base class.
- New: IUP_ASSERT compile flag.
- New: **IupMainLoopLevel** function.
- New: support for the HILDON framework that runs on top of GTK on the Maemo platform used by the Nokia Internet Tablets. Thanks to Otfried Cheong.
- Changed: all dialogs, and all elements that have names, are now automatically destroyed in **IupClose**.
- Changed: **ATTENTION -** the following headers were deprecated iupcbox.h, iupsbox.h - use iup.h only
- Changed: **ATTENTION -** the headers iupcompat.h and iupcpi.h were removed. They are not supported anymore.

Common Attributes

- New: CHARSIZE convertion factor used by the SIZE attribute.
- New: NAME used by **IupGetDialogChild.**
- New: font face name mappings for Courier, Times and Helvetica.
- New: functions **IupGetClassAttributes, IupGetIntInt**.
- New: CLIENTSIZE returns the size of containers excluding their decoration.
- New: TIP additional attributes (Motif and Windows): TIPFONT, TIPDELAY, TIPBGCOLOR, TIPFGCOLOR, TIPBALLON (Windows Only), TIPBALLONTITLE (Windows Only), TIPBALLONTITLEICON (Windows Only), TIPVISIBLE. Not available in GTK.
- New: TIPRECT auxiliary attribute for the TIP common attribute.
- Changed: attribute FONT now uses a common a more flexible definition for all drivers, old format is still supported. The default FONT in Motif is now "Fixed, 10".
- Changed: **ATTENTION -** Now attributes are stored in the internal hash table only if not processed or allowed by the element class 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: **ATTENTION -** the BGCOLOR is now ignored in **IupLabel**, **IupFrame**, **IupToggle** (for the text) and **IupVal**. They will use the background color of the native parent.

Global Attributes

- New: APPSHELL, XDISPLAY, XSCREEN, XSERVERVENDOR, XVENDORRELEASE in Motif.
- New: VIRTUALSCREEN and MONITORSINFO in Windows.
- Changed: LANGUAGE default from PORTUGUESE to ENGLISH.
- Changed: TRUECOLORCANVAS and SYSTEMLANGUAGE are now available in all drivers.

Common Callbacks

- New: IUP_IGNORE return code accepted for **IDLE_ACTION** callback to automatically remove the callback.
- New: UNMAP_CB for all controls
- Changed: MAP_CB, ENTERWINDOW_CB, LEAVEWINDOW_CB for all controls.

Layout

- New: functions **IupGetDialogChild, IupUnmap, IupReparent, IupInsert, IupUpdateChildren**, **IupGetClassType, IupGetChildPos** and **IupGetChildCount.**.
- New: FLOATING attribute to control the inclusion of the element in layout processing for **IupHbox**, **IupVbox** and **IupZbox**.
- New: HOMOGENEOUS attribute to control the spacing in layout processing for **IupHbox** and **IupVbox**.
- New: EXPANDCHILDREN attribute to control the expansion in layout processing for **IupHbox** and **IupVbox**.
- New: NORMALIZESIZE attribute to control the natural size in layout processing for **IupHbox** and **IupVbox**.
- New: element **IupNormalize**r.
- New: CGAP and CMARGIN for **IupVbox** and **IupHbox** that use SIZE units.
- New: VALUEPOS and VALUE_HANDLE attributes for **IupZbox**.
- Changed: default value for ALIGNMENT in **IupZbox** is now "NW".
- Changed: **IupAppend** and **IupDetach** can now be used for dynamic creation of menus or containers, even after the element is mapped.
- Changed: **IupDetach** will now automatically unmap the element.
- Changed: **IupAppend** will now return the actual parent.
- Changed: **IupUpdate** now only mark the control to be redraw instead of redrawing at the function call.

Dialogs

- New: MINSIZE and MAXSIZE attributes. In Windows 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: DROPFILES_CB and RESIZE_CB callbacks.
- 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 to check if the dialog was shown with **IupShow** or **IupPopup**.
- New: **IupColorDlg, IupFontDlg** and **IupMessageDlg** native pre-defined dialog as elements.
- New: SHOWHIDDEN attribute for **IupFileDLg**. Preview canvas support for the Motif driver.
- New: tip string for each param in **IupGetParam**. And a new "c" param to show a RGB color string with extra controls to show the color and open the color selection dialog.
- Changed: SAVEUNDER dialog attribute now is also available in Motif.
- Changed: DROPFILES_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: the default value of the Windows attribute COMPOSITED is now YES, except in Windows Vista.
- Changed: **IupDestroy** is now automatically called for child dialogs when the parent is destroyed.

Canvas

- New: LINEX, LINEY, XAUTOHIDE and YAUTOHIDE attributes for the scrollbar.
- New: CLIPRECT attribute, a rectangle that has its region invalidated for paiting.
- Changed: if ACTION is defined nothing is painted in the canvas, now also in Motif.
- Changed: BORDER is now also supported in Motif.
- Changed: **ATTENTION -** 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.

Label, Button and Toggle

- New: attributes PADDING, ELLIPSIS, WORDWRAP and MARKUP for **IupLabel**.
- New: IMPRESSBORDER, PADDING, MARKUP, FOCUSONCLICK and ALIGNMENT attributes for **IupButton**.
- New: support for image and text simultaneous in **IupButton**.
- New: support for mnemonics in **IupLabel**, **IupButton** and **IupToggle**.
- New: RADIO attribute for **IupToggle**.
- Changed: ALIGNMENT attribute now includes vertical alignment values.
- Changed: **IupButton** now supports text with more than one line.

Text and Multiline

- New: APPENDNEWLINE and PADDING attributes. CUEBANNER and FILTER attributes in Windows.
- New: MASK attribute for **IupText**, **IupMultiline**, **IupList** and **IupMatrix**. The iupmask functions are now obsolete, autofill option and MATCH_CB callback are not supported anymore.
- New: text formatting using FORMATTING and ADDFORMATTAG attributes in Windows and GTK. New attribute OVERWRITE when using text formatting.
- New: ALL and NONE values for SELECTION attribute.
- New: SCROLLTO attribute. New attributes SCROLLTOPOS, CARETPOS and SELECTIONPOS using 0 based character position. New function **IupTextConvertXYToChar** to convert (x,y) coordinates in (lin, col, pos) character positioning.
- New: SPIN, SPINVALUE, SPINMIN, SPINMAX, SPININC, SPINALIGN and SPINWRAP attributes. New SPIN_CB callback. The **IupSpin** control is now obsolete.
- New: VISIBLECOLUMNS, VISIBLELINES attributes gives much better control over size than the SIZE attribute.
- Changed: **IupMultiline** is now implemented as **IupText** with MULTILINE=YES.
- Changed: **ATTENTION - VERY IMPORTANT -** the ACTION callback in **IupText** now does NOT process extended keys anymore. It is called only if the text is edited, and key=0 if it is not a valid character. The callback now is called before the text is updated on screen.
- Changed: the SELECTION and CARET attribute in Windows do NOT change the focus anymore. The NC attribute now only restricts keyboard input.
- Changed: added support for BUTTON_CB and MOTION_CB callbacks. BUTON_CB can return IUP_IGNORE so the default processing will be ignored.
- Changed: CARET_CB now includes 0 based character position.
- Changed: **ATTENTION -** the Natural Size does not uses the text contents anymore. To control the Natural Size use the SIZE/RASTERSIZE attributes, or VISIBLECOLUMNS/VISIBLELINES attributes, or EXPAND.

List

- New: APPENDVALUE, CANFOCUS, COUNT, DRAGDROP, INSERTITEMn, REMOVEITEM, TOPITEM, SPACING, VISIBLECOLUMNS, VISIBLELINES attributes.
- New: BUTTON_CB, DBLCLICK_CB, DROPDOWN_CB, DROPFILES_CB, MOTION_CB callbacks.
- New: **IupListConvertXYToItem** function.

Other Standard Controls

- New: INVERTED and TICKSPOS attributes for **IupVal**.
- New: PADDING, VALUE_HANDLE, VALUEPOS, MULTILINE and TABIMAGE attributes for **IupTabs**.
- New: control **IupProgressBar**, similar to **IupGauge** but with the text.

- Changed: **IupFrames** now are native parents of their children.
- Changed: **IupVal** implemented as a native control. Attributes HANDLER_IMAGE and HANDLER_IMAGE_INACTIVE are not supported anymore.
- Changed: **IupCbox** is not based on **IupCanvas** anymore.
- Changed: **IupTabs** implemented as a native control. Attributes ALIGNMENT, FONT_ACTIVE, FONT_INACTIVE, TABSIZE and REPAINT are not supported anymore.

Additional Controls

- New: focus feedback and keyboard control for **IupColorbar**.
- Changed: **IupControlsClose** is now deprecated. Declaration still remains for compatibility, actual function does nothing.
- Changed: the NO_COLOR attribute is deprecated, now it simply sets the BGCOLOR attribute in **IupCells**.
- Changed: in **IupColorBrowser** moved from HLS to HSI, added support for resize, anti-aliasing, support for BGCOLOR attribute, feedback for ACTIVE attribute, and feedback for focus. New HSI attribute. New support for mouse wheel to change Hue. New support for PgDn and PgUp keys to change Hue.
- Changed: **IupTabs** and **IupVal** are NOT part of the additional controls anymore. They are now standard controls using native elements.
- Changed: renamed MARGIN attribute to PADDING in **IupGauge**. **IupGauge** is deprecated in favor of **IupProgressBar**.
- Changed: An **IupGLCanvas** when inside an **IupFrame** in Win32 will now work normally. But the dialog COMPOSITE attribute must be NO for hardware acceleration in Windows.

Menus

- New: HIDEMARK, AUTOTOGGLE and TITLEIMAGE attributes for **IupItem**.
- New: BGCOLOR support for **IupMenu**.
- New: Submenu now supports the IMAGE attribute.
- New: RADIO attribute for **IupMenu**.
- Changed: In GTK to have a menu item that can be marked you must set the VALUE attribute to ON or OFF, or set HIDEMARK=NO, before mapping the control.
- Changed: The HIGHLIGHT_CB, OPEN_CB and MENUCLOSE_CB callbacks now work normally for popup menus. HIGHLIGHT_CB is called for items and submenus.
- Changed: OPEN_CB and MENUCLOSE_CB are defined for menus, but it is checked at the parent submenu for backward compatibility with IUP 2.x.
- Changed: TITLE for submenus can now be changed after the element is mapped.
- Changed: Children can be added or removed from menus even after the menu is mapped.
- Changed: menus can now be dynamically changed even after mapped.

Images

- New: support for 24 and 32 bpp images using **IupImageRGB** and **IupImageRGBA** constructors.
- New: "UPARROW" cursor in Motif. New cursors "RESIZE_NS" and "RESIZE_WE". Updated cursor documentation with pictures of all pre-defined cursors.
- Changed: the automatic generation of inactive images for a more smooth one, still using a modified version of the background color to create the disabled effect.
- 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. **IupImageLibClose** removed, loaded images will now be automatically unloaded.

Keyboard

- 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: CANFOCUS attribute for **IupButton**, **IupTogle**, **IupText**, **IupCanvas** and **IupVal**.
- Changed: SHIFTKEY and CONTROLKEY are now available in all drivers.
- Changed: Removed the conflicts: K_BS=K_cH, K_TAB=K_cI and K_CR=K_cM. New key code macros **iup_isShiftXkey**, **iup_isCtrlXkey**, **iup_isAltXkey** and **iup_isSysXkey**.

History of Version 2.x

Migration Guide
IUP 2.x to IUP 3.x

Critical Changes (from 2.x to 2.7/3.0)

All critical changes were packed in version 2.7 so you can prepare your code to work with both 2.7 and 3.0 versions. And you will be able to alternate between both versions without having to add "ifdef"s to your code. The differences in the "iup.h" header file from 2.7 to 3.0 should contains only the new features introduced in 3.0.

IupOpen function declaration now include command line arguments used by X-Windows and GTK - The most important change is the signature of the **IupOpen** function. It was changed to include the main function arguments. The GTK and Motif toolkits use them. In IUP prior to version 2.7 they were ignored for Motif. In Windows they are always ignored. If for some reason you do not have access to the main function arguments you can use NULL in IupOpen. As a general rule the change is:

```
IupOpen()  >>   IupOpen(&argc,&argv)
```

You will also have to search&replace a few things in your source code:

```
the attribute "WIN_SAVEBITS"  >>  "SAVEUNDER"
the function IupGetType  >>  IupGetClassName
```

Although the following were considered obsolete in IUP 2.6, their backward compatibility code were removed in 2.7. So you may have to search&replace for:

```
the attributes "MOTIF_FONT" and "WINFONT"  >>  "FONT"
the value of the attribute "CURSOR" =  "IUP"   >>   "HELP"
the definition IUP_ANYWHERE  >>   IUP_CURRENT
the constructor IupColor (removed)  >>  use the color value
```

The "cdiup" and "cdluaiup" libraries moved from CD to IUP under the name "iupcd" and "iupluacd" - Also you will have to change your makefile or IDE project because we changed some library names to solve the cross dependencies between IUP, CD and IM libraries.

Strategic Changes (from 2.7 to 3.0)

All the changes described here are backward compatible with 2.7. So after doing them you will still be able to go back to 2.7.

Some global attributes like DEFAULTFONT, *BGCOLOR and *FGCOOR are now obtained from the system instead of hardcoded, this affects mainly applications in Windows were the hardcoded DEFAULTFONT was "Tahoma, 8" and the user changed the default font or used the Large Fonts option. If your dialog is too big in the new font then you can simply set DEFAULTFONTSIZE to force a smaller value.

The following headers were deprecated iupcb.h, iupcells.h, iupcolorbar.h, iupdial.h, iupgauge.h, iupmatrix.h, iuptree.h - they now simply include iupcontrols.h. You can replace them by iupcontrols.h in your code.

The following headers were deprecated iupcbox.h, iupsbox.h, iupgetparam.h, iupspin.h, iuptabs.h, iupval.h - they now simply include iup.h. You can remove them from your code.

The ACTION callback in **IupText/IupMultiline** now does NOT process extended keys anymore - the callback is called only if the text is edited, and key=0 if it is not a valid character. In 2.x the key parameter were used for some navigation keys, but now is used only for keys associated with characters. This is the most impacting change from 2.7 to 3.0, because some functionality in your application could stop working. Use the K_ANY or K_* callbacks instead to process navigation keys.

The Natural Size of **IupText/IupMultiline** does not uses the text contents anymore - to control the Natural Size use the SIZE/RASTERSIZE attributes, or the VISIBLECOLUMNS/VISIBLELINES attributes, or the EXPAND attribute. This will avoid the automatic resize of the **IupText/IupMultiline** if its content is changed by the user and the size of the dialog is changed so the layout is recalculated.

Now in **IupCanvas** the scrollbar parameters X/YMIN, X/YMAX and X/YLINE are updated only when DX/Y are updated. POSX/Y will only update the position of the scrollbar. In version 2.x was necessary to set POSX/Y to update those parameters.

The BGCOLOR attribute is now ignored in **IupLabel**, **IupFrame**, **IupToggle** (for the text background) and **IupVal**. They will use the background color of the native parent.

The **IupItem** in GTK must have its VALUE attribute defined (ON or OFF) before mapping - so it can have the check mark, or define HIDEMARK=NO. If not done the item will not be checkable.

The new **IupTabs** does not supports the inactive tab feedback. So the tabs will be always active, although its children will be sucessfully disabled. The return value of the TABCHANGE_CB callback is not processed anymore.

In **IupMatrix** if the last column is not completely visible, then adjust the matrix size manually or let the natural size adjust it for you using NUMCOL_VISIBLE_LAST=YES and NUMLIN_VISIBLE_LAST=YES. Since the selection is made only using the mouse pressing a key will NOT clear the selection anymore. You can still do that setting MARKED=NULL in the K_ANY callback.

In **IupTree** DEPTH is now a read-only attribute, use the INSERT* attributes to properly add nodes. The SELECTION_CB and MULTISELECTION_CB callbacks now ignore their return value. Now you can only add nodes to the tree after it has been mapped to hte native system. NAMEid attribute renamed to TITLE, old attribute still works but will be removed in future versions.

Deprecated Controls (will be removed in a future version)

The controls **IupGauge**, **IupTabs** and **IupVal** based on the CD library were replaced by native controls in IUP 3.0. But not all features of the old controls are available in the new controls, so the old ones are still available until the applications have the chance to migrate them to the new ones. In a future version the old ones will be completely removed from IUP.

The **IupGauge** was superseded by **IupProgressBar**. The new control does NOT supports the display of a text inside the progress bar area. If you use the text in the **IupGauge** you will have to add a label on top or bottom of the progress bar to obtain a similar result. A less critical change is the DASHED attribute which has limited support in the **IupProgressBar**.

The **IupTabs** was moved to the main library under the same name because the new one almost have all the features of the old one. The most impacting feature is the TABORIENTATION attribute that has limited support in the native controls.

The **IupVal** follows the same idea. The only feature that it is not available is the use of a custom handler image.

History of Changes in Version 2.x

CVS (17/Jun/2009)

General

- Changed: the **IupOleControl** in Lua will not automatically initialize LuaCOM anymore. The application must manually call "elem:CreateLuaCOM()". The previous initialization was incorrect (thanks to Ross Berteig).
- Fixed: parameter checking and the return value in Lua for **IupListDialog** when type=2.
- Fixed: the return value for **IupGetText** when the user canceled.
- Fixed: The 32 bits version of the IupLua console in Windows XP64 was not working.
- Fixed: IupLua initialization when retrieving the argc/argv arguments for **IupOpen**. (Thanks to Ross Berteig)
- Fixed: action callback return value in Lua for the **IupGetParam** dialog. (Thanks to Zhiwei)

Version 2.7.1 (15/Dec/2008)

General

- Fixed: the iuplua51 makefile where not using the g++ linker, so require"imlua" failed.
- Changed: removed csh dependency from make_uname scripts.
- Changed: removed IupSpeech from source and documentation.

Motif

- Fixed: IupOpen was crashing if used with NULL parameters.
- Fixed: DIRECTORY attribute in IupFileDlg when set to NULL did an invalid memory access.
- Fixed: invalid default value for SCROLLBAR attribute in IupMultiline and IupList.
- Fixed: size of tips window when displaying a multiline string.

IupControls

- Fixed: some frame lines where not drawn in IupMatrix.

Version 2.7 (14/Oct/2008)

General

- Changed: **INCOMPATIBILITY** - **IupOpen** function declaration now include command line arguments used by X-Windows and GTK.
- Changed: **INCOMPATIBILITY** - "IUP" cursor in Windows renamed to "HELP" cursor.
- Changed: **INCOMPATIBILITY** - WIN_SAVEBITS renamed to SAVEUNDER.

- Changed: **INCOMPATIBILITY -** removed old "MOTIF_FONT" and "WINFONT" attributes. Use only the "FONT" attribute.
- Changed: **INCOMPATIBILITY** - removed old IUP_ANYWHERE and IupColor definitions.
- Changed: **INCOMPATIBILITY - IupGetType** renamed to **IupGetClassName**.
- Changed: **IMPORTANT** - all functions that receive a constant string now has the "const" modifier for the string parameter declaration.
- Changed: **IMPORTANT** - Copyright notice modified to reflect the registration at INPI (National Institute of Intellectual Property in Brazil). License continues under the same terms.
- Changed: **IMPORTANT** - the support services (Downloads, Mailing List and CVS) moved from LuaForge to SourceForge.
- Changed: All dll8 and dll9 DLLs now have a Manifest file that specifies the correct MSVCR*.DLL.
- Changed: Makefiles for UNIX now uses a compact version of Tecmake that does not need any installation, just type "make".
- Changed: removed "INCLUDE" parameter for FILE_CB callback in IupFileDlg.
- Changed: improved automatic inactive image generation.
- Changed: premake files are used now only internally and were removed from the distribution.
- Changed: IupLua3 libraries are not included in the distribution anymore. They are only available in source code or internally at Tecgraf.
- Changed: All Lua samples now have the extension .wlua, and contains require"iuplua" and iup.MainLoop() in the code. Thanks to Ryan Pusztai.
- Changed: added traceback information to error message dialog in IupLua. Thanks to Fred Abraham.
- Changed: The IupLua Console now must include require commands for any additional library.
- Fixed: IupView image export in C format.
- Fixed: removed MARGIN from IupFrame documentation. IupFrame does not have a MARGIN attribute.
- Fixed: removed MARGIN from IupZbox documentation. IupZbox does not have a MARGIN attribute.
- Fixed: SYSTEM global attribute in Windows, when running Windows Vista.
- Fixed: Improved visual appearance and ticks of bar mode in IupPPlot.
- Fixed: missing IupMessagef export in the DLL.
- Fixed: LEDC generated code for 64-bits.
- New: **IMPORTANT** - the "cdiup" and "cdluaiup" libraries moved from CD to IUP under the name "iupcd" and "iupluacd". But headers and documentation remains on the CD package. Function names were NOT changed. This change eliminates a cross-dependency of IUP and CD, now only IUP depends on CD.
- New: "iupluaimglib" library so require"iupluaimglib" can be used to dinamically load the image library.

Windows

- Fixed: invalid memory access when set FONT to NULL.
- Fixed: CARET position when a selection is interactively changed or when the caret is at the begining of the selection in IupText, IupMultiline and IupList.
- Fixed: TABSIZE IupMultiline attribute scale convertion.
- Fixed: invalid character inserted in IupMultiline when opening a dialog from a Ctrl+key combination.

Motif

- Fixed: Removed X run time warning when creating a list.

IupControls

- New: FRAMEVERTCOLORL:C, FRAMEHORIZCOLORL:C and CHECKFRAMECOLOR attributes for IupMatrix.
- Fixed: EXTENDED_CB callback was never called in IupColorbar.
- Fixed: invalid memory access in IupTree when using images with a color index greater than 128.]
- Fixed: invalid memory access in IupTabs when all tabs are disabled and a next or previous tab button is pushed.
- Fixed: invalid memory access in IupColorbar.
- Fixed: invalid call to CLICK_CB when resizing column in IupMatrix.

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 specifiy 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: IupFileDlg 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: convertion 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_dostring.
- Fixed: the second ihandle parameter inside the callabacks 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 collum and the scroolbar 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 execption 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 calback management in C and in Lua. With the new IupGetCallback we were able to remove the inheritance mecanism 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 preffix "_IUP".
- Changed the default limit for text in IupText and IupMultiline to be 2^{31}.
- 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 colum 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 DROPDOW=Yes after inserting elements in the list.
- New BACKINGSTORE IupCanvas attribute so the backing store can be disabled.
- Changed IupToggle with IMAGE and IMPRESS to behave like in Windows, where the button border is always shown.
- Fixed error in menu item initialization.

IupControls

- IMPORTANT: for best results CD version 4.4 should be used.
- Fixed IupSpin keyboard response and mouse press & hold response.
- New MULTISELECTION_CB callback for IupTree.
- New IupCells control. It is an application controlled matrix. More simple and faster than IupMatrix. Can also span cells. Thanks to 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 SORTSIGN*C* attribute to show a sort sign (up or down arrow) in the column C title.
- New drawing in double buffer mode to minimize flicker.
- Fixed dropdown feedback drawing.
- Fixed focus feedback after double click editing.
- The alignment of the text in a cell with a dropdown feedback now considers the horizontal space occupied by the feedback.
- The DRAW_CB callback drawing area now does not includes the focus feedback area if HIDEFOCUS=NO (the default).
- NUMCOL_VISIBLE and NUMLIN_VISIBLE now when retrieved returns the current number of visible lines.
- Fixed problem after trying to edit a non editable cell the focus gets lost.
- Reviewed documentation and behavior of marks.

IupLua

- IupLua5 source code is now 100% compatible with Lua 5.1.
- The iuplua binding and all its libraries can now be dinamically loaded in Lua 5. IupOpen will be automatically called.
- iupkey_open can now be called from Lua 5, using iup.key_open.
- New IupGetParam binding.
- Changed the keys definitions (K_*) in Lua so now they are exactly the same as the definitions in C.
- Fixed invalid IupGetAllNames in IupLua5. Fixed missing IupGetAllNames binding in IupLua5.
- Fixed IupTree EXECUTELEAF_CB callback in IupLua5. It was expecting an invalid extra parameter.
- Fixed error in IupTabs memory initialization in IupLua5.
- Fixed missing IupGetText binding.
- Fixed missing pre-defined masks for iupMask.
- Fixed missing isxkey macro binding.
- Fixed missing callback scroll_cb in IupLua3.
- Fixed missing IupVersion documentation and binding.
- Fixed IupSetGlobal and IupStoreGlobal in IupLua5.

Version 2.3.1 (18/Apr/2005)

General

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

Windows

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

Motif

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

IupControls

- Changed IupGetParam now uses only the number of lines to determine the number of parameters. The last 0 is not necessary anymore.
- Fixed bug in IupColorBrowser destroy.
- Fixed IupTree initialization for LED usage.
- New IupTree feature to rename a node in place.
- New IupColorbar control. It is a palette of colors to allow the selection of primary and secondary colors. Thanks to 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 retreiving the FONT attribute.
- Added backward compatibility code for Motif 1.2. Must edit makefile to add the file "src/mot/ComboBox1.c".

IupControls

- Missing support for IupList with EDITBOX=YES in iupMask.
- BGCOLOR for images were ignored in the IupTree.
- Now some matrix cell attributes are not inherited from parent. Like "L:C", "ALIGNMENT*", "FGCOLOR*", "BGCOLOR*", "FONT*", "WIDTH*" and "HEIGHT*", for optimization reasons.
- IupTree now uses double buffer for optimal drawing.
  To avoid flicker during resize in Windows, do not use it inside a IupFrame, and use CLIPCHILDREN=YES.
- New utility functions: IupTreeSetAttribute, IupTreeStoreAttribute IupTreeGetFloat, IupTreeSetfAttribute, IupTreeGetAttribute, IupTreeGetInt.
- New IupMatrix callback DRAW_CB to allow a custom drawing of the cell contents.
- New IupTree DRAGDROP_CB callback.
- New IupSpin and IupSpinbox utility functions.

IupLua

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

Version 2.2.2 (07/Oct/2004)

General

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

Windows

- New ENTERWINDOW_CB and LEAVEWINDOW_CB for buttons.
- Fixed double click for button, toggle and list were not being considered as two clicks.
- removed FLAT style from toggles with IMPRESS image. Fixed size of toggle with image.
- New attribute SHOWDROPDOWN to open the dropdown list programmatically.
- Removed a black border around IupMultiline and IupText.
- Removed the TABSTOP for non marked Toggles inside a Radio.
- Fixed invalid memory access when menu item is activated and all dialog controls are disabled.
- Fixed IupFileDlg ignored the x,y parameters of IupPopup.

Motif

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

IupControls

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

IupMatrix

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

IupLua

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

Version 2.2.1 (25/Aug/2004)

General

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

Windows

- Fixed dynamic cursor creation.
- Toggle with inactive image could be enabled/disabled only once.

- Fixed toggle in Radio behavior.
- Some keys were not being treated correctly.
- Improved key codes management.

Motif

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

Controls

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

IupLua

- Wrong implementation of DROPCHECK_CB.

Version 2.2 (11/Aug/2004)

INCOMPATIBILITIES

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

General

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

Windows

- Menus for notification icons (system tray) were not working correctly.
- Cursors in Windows now accept more than 2 colors and can have size different from 32x32.
- IupImage was rewritten in Windows to be more simple and flexible. This also solved some weird button backgrounds in gcc3.
- New global attributes "SHIFTKEY" and "CONTROLKEY" can be "ON" or "OFF", return the the key state (windows only).
- The default size for buttons in Windows was increased by 2 characters.
- Returning IUP_CLOSE in a SHOW_CB of an IupPopup wasn't closing dialog.
- IupOpen instead of initializing OLE, now only initializes COM (CoInitialize).
- The border of buttons are now drawn by a system function instead of simulated.
- New attribute "PLACEMENT" to show the dialog maximized or minimized.
- In IupFileDlg when browsing for folder it will use a new interface, with a resizable dialog and other features.
  Also in IupFileDlg fixed start position for IupPopup. New file selection callback and preview area. IupFileDlg was not using the IUP_PARENTDIALOG attribute. Default value for IUP_NOOVERWRITEPROMPT was wrong. ALLOW_NEW was inconsistent with the documentation.
- The button callback now is called only when the button is released inside the button area.
- WOM callback renamed to WOM_CB.
- New "HELPBUTTON" attribute for the dialog.
- The menu item now accepts auxiliary bitmaps.
- When the dialog has a multiline and the user press ESC the window was improperly closed.
- Fixed comboox resize feedback. When resizing the dialog the combobox was temporarily opened.
- IupCanvas was not receiving arrow keys events correctly in keypress_cb.
- IupHide now can close popup dialogs.
- Attribute TABSIZE for IupMultiline in Windows was not documented.
- Default value for attribute BGCOLOR for IupCanvas in Windows was not documented.
- Direction keys now are processed by the ACTION callback for IupText.
- The GETFOCUS_CB and KILLFOCUS_CB management for the controls was reviewed and optimized.
  GETFOCUS_CB now works for toggle and button.
- First RESIZE_CB of the canvas received a wrong canvas size.
- Label alignment for images was always center.

Motif

- New global attribute: "MOTIFVERSION".

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

Controls

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

Matrix

- Documentation reviewed and reorganized.
- Returning IUP_CLOSE in CLICK_CB was not closing application.
- The scrollbar drag will now simultaneously scroll the matrix.
- New callback "DROPCHECK_CB" to aid the dropdown feedback in the cell.
- New utility functions: IupMatSetAttribute, IupMatStoreAttribute IupMatGetFloat, IupMatSetfAttribute, IupMatGetAttribute, IupMatGetInt.
- Fixed some display erros in Windows because of an error in the size of the scrollbar.
- In Windows pressing a key in a menu activates the k_any of the last active element. In the matrix this turns into an infinit loop. The matrix now uses the keypress_cb instead of the k_any callback.
- Fixed empty selection in the dropdown list if the user press a regular key to start editing the cell.
- Fixed invalid dropdown value if the user changed focus to the scrollbars.
- CLICK_CB was called twice in a double click (press+release).
- In Motif, the textbox and the dropdown did not open when you double click a cell. But now the user still needs to click again in the control to put it into focus.
- After editing the cell in the last line, now the focus goes to the column on the right at the last line, instead of the first line.
- BGCOLOR now works also for titles.
- FONT attribute now can be set/get just like BGCOLOR and FGCOLOR. But the cell size is calculated always from the matrix attribute IUP_FONT.

Tree

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

IupLua

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

LEDC

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

Version 2.1 (18/Feb/2004)

General

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

IupTree

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

IupMatrix

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

Windows

- IupOpen/IupClose now initializes OLE (OleInitialize/OleUninitialize)
- ENTERWINDOW/LEAVEWINDOW reimplementation. LEAVEWINDOW does not fail anymore
- Mouse hook removed. Better performace
- New attributes TRAY, TRAYTIP and TRAYIMAGE and new callback TRAYCLICK_CB which allows a dialog to be put in the tray
- Action in IupText now responds to the [ENTER] key
  Some keys were not working with keypress callback: \ ] [' ; / . ,
- New attribute NATIVEPARENT, which makes any dialog in Windows able to be parent of a IUP dialog (even from other toolkits)
- Better protection dealing with other processes messages
- IupFileDialog when used to get directory was not updating STATUS attribute correctly
- IUP_APPEND small memory problem fix
- atexit removed
- KILLFOCUS_CB and GETFOCUS_CB were not being called when focus goes to the menu
- MAP_CB in a canvas is now called before RESIZE_CB (like the Motif driver)
- ALT-F4 was not working to close application
- Images sometimes show black using Visual C: do not use option in Visual C 6.0 /NODEFAULTLIB:libcd
- IUP_TIP does not show when the fade effect is on: MS fixed the problem, use autoupdate

IupLua 3.2, 4.0, 5.0

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

IupVal

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

Manual

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

Distribution

- README on how to compile IUP with tecmake

Version 2.0.1 (31/Jul/2003)

General

- Attribute IUP_TYPENAME replaced by IupGetType function
- minor bugs introduced in 2.0 because of internal old misuse of the hash table.
- Following controls were not working with LED: val, dial, gl, matrix, tree.
- New canvas attribute "DRAWSIZE" that returns the drawing area of the canvas (in Windows we may have an addicional border included in "RASTERSIZE").

Windows

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

Motif

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

LEDC

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

IupLua

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

Version 2.0 (23/Jun/2003)

General

- IUP has undergone a large internal reorganization, but no structural or algorithmic changes have occurred. The purpose of this reorganization was to standardize function, variable and module nomenclature. This process is not yet complete, but the few remaining details will be solved in the next version.
- Table Hash was completely replaced with a modified version of Lua 4. This version is internal of IUP and does not affect applications. This has brought us a better management of the memory used by attributes.
- The CPI was changed to allow the creation of native controls, as well as controls based on IupCanvas. The internal controls were not yet rewritten over the new CPI - this will be done progressively in the next versions.
- The Ihandle definition changed from "void" to "typedef struct Ihandle_ Ihandle;". This has direct implications on C++ applications that did not do pointer typecast. In C++, code errors might occur and, in C, there might be warnings.
- New control IupTimer. Allows creating timers in Windows and Motif.
- New callback "KEYPRESS_CB". Allows intercepting any key and replacing all callbacks "K_xxx".
- IupHelp was rewritten in a simpler way. In Windows, it simply uses the system's configuration to open a URL and, in UNIX, it directly runs Netscape or another executable configured by an environment variable.
- New attribute "FULLSCREEN", allows creating a dialog that occupies exactly the whole screen.
- Dialog IupGetFile was rewritten using IupFileDlg.

Windows

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

Motif

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

IupLua

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

IupControls

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

IupMatrix

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

IupTree

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

IupGL

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

History of Version 1.x

History of Changes in 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 iuptree control, the callback BRANCHOPEN_CB was not passing the node parameter.
- In the iuptree control, new functions were implemented to associate and retrieve a Lua Table from a node or leaf.
- IupGLCanvas binding.

IupTree

- Expand and collapse no more alters selection of elements.
- When all nodes were deleted using "DELNODE0", "CHILDREN" inside a tree_selection callback, the program crashed.
- BRANCH_OPEN now passes parameter node.
- IUP_DEPTH now works for folders and leaves. Attention: the depth works only with the appointed element, not with its children.
- Some conditions necessary for a DEPTH change were wrong.
- Redraw optimization.
- When a tree was big, the scrollback 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.**

IupMatrx

- 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 immediatly after it is set though programming.
- Tip now can be removed immediatly.
- In a SubMenu, the attribute ACTIVE was not working propertly.
- 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 unecessary 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 propertly. When the user clicked a node, while the timer was still waiting for the second click, it was impossible to click a nother node.
- Hitting the space button with CTRL pressed now marks the element immediatly.
- SELECTION_CB callback was created. This callback is called when any type of mark is made on the Tree. The return value blocks this action.
- Removed callbacks BEFORESELECTION_CB and AFTERSELECTION_CB.
- Setting IUP_VALUE though programming does not activate callbacks anymore.
- Keyboard control, including arrow keys, PGUP, PGDOWN, HOME e END were not working propertly.
- 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 FGCOLOR 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 IUPLUA_QUIET attribute.
- The callbacks in IupLua are now inherited (eg.: k_any from a dialog is called when IupCanvas does not have k_any).
- The 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 [hvz]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 FGCOLOR, BGCOLOR, and fonts FONT, WINFONT, MOTIFFONT.

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 (plic).
- Keyboard accelerators for menus now work, since the focus is no longer on the dialog. When a dialog receives the focus back, it sets the focus to the last control inside it that had the focus.
- IUP_K_ANY no longer issues beeps when keys are pressed on the canvas.
- When the IUP_STARTFOCUS attribute is not defined, the focus is set for the first control in the dialog that accepts it, thus preventing the dialog from keeping the focus and allowing the menus to be called via accelerator.
- Attribute IUP_SELECTION was implemented.

Motif

- 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.
- IupFileDlg:
  - The default value for the DIALOGTYPE attribute was not being recognized (the program aborted when there was no defined value).

    - When ALLOWNEW = NO, the dialog informs if the user is specifying a non-existing file (instead of simply returning, as was happening).
    - When the dialog type was OPEN, the returned value was 1 (Cancel) even when the user confirmed the operation.
    - If DIALOGTYPE is SAVE, a confirmation is required if the file already exists.
    - A new dialog was created for each popup without destroying the previous dialog.
    - The NOCHANGEDIR attribute was implemented.
    - The dialog does not return if the user specifies a new file when attribute ALLOWNEW = NO. The same happens when attribute ALLOWDIR = NO and a directory is specified. In these cases, alerts are shown.
- The iupGetColor function for CPI controls was replaced in functionality by the iupGetRGB function (iupGetColor is maintained for compatibility purposes, but it should no longer be used).
- TRUECOLORCANVAS was created. It indicates if the display allows the creation of TrueColor windows (> 8bpp), even if the default is PseudoColor.
- Tabs: a problem was fixed concerning the use of the VISIBLE attribute for elements belonging to a non-selected tab.
- IupHelp: allows using a browser (default = Netscape) for viewing HTML pages.
- The ACTION_CB callback, from IupText, now receives, apart from Ihandle* and int, a char* pointing to the new text value in case the key is confirmed.
- Dropdown lists were not correctly processing the VISIBLE attribute.
- A problem with the initialization of multiple-selection lists was solved: the VALUE attribute was not being respected in some cases.
- Attributes FGCOLOR and BGCOLOR from the dropdown list were not being correctly updated.
- IupLoopStep was re-implemented: now it no longer blocks when there are no events to be processed (it simply returns DEFAULT).
- The dropdown list is closed when the associated textbox is totally or partially darkened.
- The dropdown list was not being closed when the dialog lost the focus if IupIdle was registered.
- A problem in the exhibition of CPI controls was fixed.
- New return code (CONTINUE) was created, specific for key callbacks, to be used when the event is to be propagated to the parent of the element receiving it.
- In some situations, elements destroyed by means of IupDestroy were receiving events, making the application abort.
- The redefinition of items in the main menu was making the dialog return to its original size.
- Consulting attribute BGCOLOR in a dropdown list was aborting the application.
- Consulting attributes BGCOLOR and FGCOLOR of a canvas with a different visual from the default was generating an X-Windows error message.
- The problem with IupFileDlg was fixed (the application was aborting).
- IupDestroy in a bar menu was inducting 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 IupFileDlg control, allowing the use of more than one filter.
- Changes were made to allow the creation of CPI elements other than CANVASes or dialogs.
- The IUP_ACTIVE attribute of a dialog can now be changed after it was mapped.
- List callback correction: the callback is now called both for selected and not selected items.
- New function void IupHelp(char *url) shows a URL in a Netscape window.
- The treatment of the new return value for keyboard callbacks, IUP_CONTINUE, was implemented.
- IUP_CURSOR attribute was implemented.
- A code was added to treat the case of toggle de-selection via IupSetAttribute.
- IUP_CARET now uses , as a separator instead of old :.
- A restriction was eliminated that prevented the function iupGetTextSize from being called passing a dialog or frame as a parameter.
- New text callback was implemented; it receives the text both before and after the change, and receives the code of the typed key.
- It was possible to set two activated radio toggles by selecting VALUE for one of them on the radio and VALUE = ON on the other toggle.
- Attributes IUP_STARTFOCUS, IUP_DEFAULTENTER and IUP_DEFAULTESC were implemented.
- The IUP_VALUE of a IupRadio was not allowing to be changed if it was not visible.
- A problem was corrected for the lists, which were being reset between a IupShow/IupPopup and another.
- Attribute IUP_SELECTEDTEXT was implemented. It returns the selected text (if there is any), with the \r already filtered.
- A bug was corrected; it caused and 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)

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

Version 1.8.1

Windows 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: IupSetfAttribute function.
- NEW: IupSetGlobal, IupGetGlobal and IupStoreGlobal functions for global attributes.
- NEW: K_sCR key; shift-enter combination is now treated by IUP (callback: IUP_K_sCR, code: K_sCR).
- NEW: IUP_TYPENAME attribute returns the name of the element type.
- NEW: CPI popup method.
- NEW: Definition of global attributes (verification only) IUP_VERSION, IUP_DRIVER, IUP_SYSTEM and IUP_SCREENSIZE.
- NEW: Attributes IUP_X and IUP_Y were implemented, for dialogs only. They provide the 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 IupFileDlg control was changed into IUP_DIALOGTYPE, which must contain OPEN, SAVE or NULL.
- CHANGE: The IupSetAttributes function now returns the Ihandle*.
- CHANGE: The IupSetAttribute function no longer returns the old value.
- CHANGE: 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 BGCOLOR 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

Roadmap for the Next Version

1. MacOSX using GTK
2. **IupSplitbox** (native IupSbox)
3. Other New Controls: **DropDownButton, GridBox, ScrollBox** and **LayoutBuilder**.
4. Drag&Drop
5. Tutorial and Demo
6. [Sugestions?]

General

- Priority: Build a MacOS X version using the new GTK Darwin native framework. This implicates in a CD using GDK/Cairo or using Quartz 2D.
- Priority: Drag&Drop between controls.
- Priority: A tutorial section in the documentation. One big Controls Demo just like GTK, wxWidgets and Qt have.
- A MacOS X native driver using Carbon.
- The actual model for control data structure in the internal SDK is limited and hard to use for derived classes.
- Unicode Support (UTF-8) in Windows and Motif.
- Simplify IupLua implementation. More similar to a regular binding implementation like CDLua and IMLua.
- Add support for loadind and saving RTF files in **IupText**. Add support for images inside the text.
- Support for text over image in **IupLabel**, **IupButton** and **IupToggle**. Or a way to composte text on **IupImage**.
- Support for changing the system menu in Windows. Support for cascading **IupPopup** for menus.
- Support for Portuguese in the **IupView** application.

Windows

- Known Issue: In Windows, when using a font for an node with TITLEFONTid in **IupTree** that is larger than the element FONT the item text will be cropped at right and bottom because the system uses the element font to calculate the item size. The only exception is when you just change the font to add a Bold style.
- Known Issue: The **IupImgLib** takes an very long time to compile under Visual C++. In a 2.5GHz CPU tooks about 5min! (since 3.0)
- Known Issue: In Windows Vista the COMPOSITE=YES attribute of the **IupDialog** is not working as expected. There is still flicker when the dialog is resized. IupTabs in Windows Vista when COMPOSITE=YES works only if MULTILINE=YES. (since 3.0)
- Known Issue: The MDI Window menu is not properly updated when the child mdi is closed. (since 3.0)
- Known Issue: When building with Open Watcom the additional controls crash. When you add debug information to the main IUP library the problem solves. We tried to track down this error but it does not occurs with debug information and our attempts without debug does not gives any results. So the IUP main library for Watcom is now distributed with debug information. (since 3.0)

Motif

- Known Issue: when the **IupList** has DROPDOWN=Yes in Motif, and the list has items with the same string, the ACTION callback will return the index of the item with the first instance of the string only. This seems to be a bug in Motif.

IupMatrix

- Change the style of the grid lines.
- Use an image as a cell content.
- Mark using the keyboard.
- Scroll with position free (not fixed to cell boundaries).
- Merge cells.
- Check mark for cells.
- Other columns/lines fixed at start like column 0 and line 0.
- Copy&paste from clipboard
- New PADDINGL:C, TIPL:C and ACTIVEL:C attributes.

IupTree

- Define minimum size based on tree nodes.
- Check mark for nodes.
- TIP attribute for nodes
- drag&drop of multiple selecteded nodes.
- Old NAMEid attribute conflicts with the common attribute NAME. Should be replaced by the new TITLEid. NAMEid will be removed in future versions. (since 3.0)
- Optimize internal node find from id and vice-versa.
- Add new nomenclature option for id, for example ":2:1:4"
- RENAMEEDIT_CB callback and RENAMEMASKid attribute.

IupPPlot

- Add support of legend text near each dataset plot.
- Adjust AutoScale to start and end at major axis ticks.
- Improve the display of values near each sample.
- PPlot force the definitions of the margins. It should have a way to automatic calculate the margins when doing automatic scaling.
- PPlot has callback mode feature that is not available in IupPPlot.
- 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 compiles under pure Visual C++ 6 (must use STL Port). Use of STL is simple and can be removed.

New Controls

- **Priority: DropDownButton** - Mix between a drop down list and a button
- **Priority: ScrollBox** - a box that can scroll the elements inside
- **Priority: GridBox** - container to distribute elements in a grid.
- **Priority: LayoutBuilder** - a dialog that shows the layout of another dialog and allows the user to reposition the elements interactively.
- ----------
- **Scrollbar** - just the scrollbar as a control.
- **CanvasCD** - a IupCanvas with a CD canvas associated.
- **UrlButton** - Button that activates a URL (looks like a flat label, but gives visual feedback when mouse is hover)
- **Toobar -** button/toggle/list, detachable
- **ImageList** - A list that contains images
- **Table** - similar to IupMatrix but using native controls
- ----------
- Project Manager/Dialog Editor/Integrated Debugger for IupLua
- FontDlg for X fonts - Motif driver
- Expandable/Closeable Frames
- PropertyGrid - a 2 column matrix with expandable/closeable items
- LUT - Lookup Table, maps a set of values in another set using a curve.
- HTML viewer
- Statusbar
- Calendar

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 was 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	1997-2009/03	1.6.36	C++	Win, X	own	3	great look, license restrictions
FLTK	LGPL*	1998-2008/04	1.1.9	C++	Win, X, Mac	own	4	was from Digital Domain. Easy to learn.
GTK+	LGPL*	1997-2009/06	2.16.2	C	Win, X, Mac	own	10	target for X-Windows, basis of GNOME, Windows is apart, Mac using X
Qt	GPL	1994-2009/03	4.5.1	C++	Win, X, Mac	own	(many)	Is free for Non Commercial, basis of KDE, Emulates the native look and feel
wxWidgets	LGPL*	1992-2009/03	2.8.10	C++	Win, X, Mac	native	6	X can use Motif or GTK
IUP	MIT	1994-2008/12	2.7.1	C	Win, X, Mac	native	2	X can use Motif or GTK, Mac using X

Table Last Update: June 2009

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.

Discussion

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

FLTK promises a new version with a better look and new features, but there are no concrete release dates. 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 was originally target for X-Windows. Nowadays GTK+ 2 is a great free C toolkit. But some predefined dialogs could be the native ones, like the File Selection, specially in Windows. The Windows port has a look and feel very similar to the Windows native look and feel, but it is different from a native application. A MacOS X port without using X-Windows is on the way, so the future is promising.

Qt had several license limitations, but since mid 2009 a new license model take place and it became more attractive. It is a very stable and powerful toolkit.

wxWidgets is an excellent choice because of the native controls and its portability.

It is hard to compare IUP with wxWidgets and Qt since they are much more than an User 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 with Motif or GTK. We plan for the future to build a native Carbon driver.

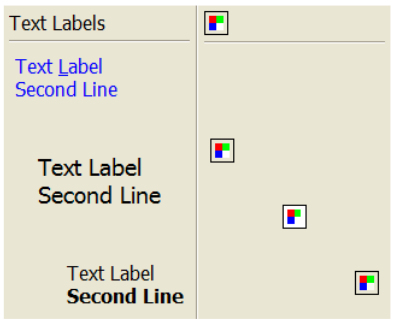
Why Still Motif? Motif is very important for non Linux systems, some Tecgraf applications run on old AIX, SGI and Sun systems, that only have Motif installed and we can not force the installation of other toolkits like GTK.

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

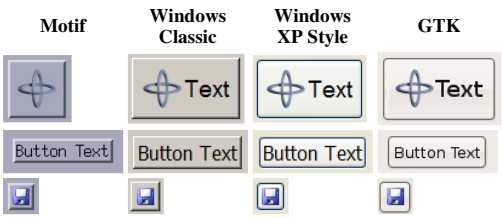
Gallery

Standard Controls

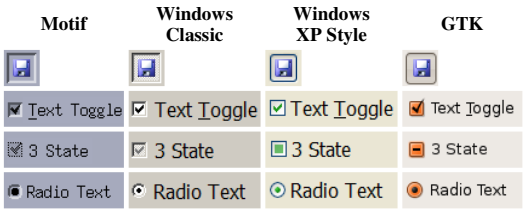
IupLabel



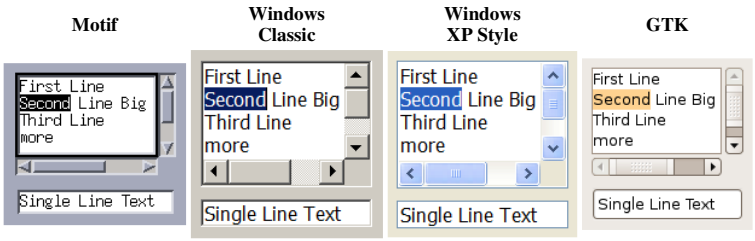
IupButton



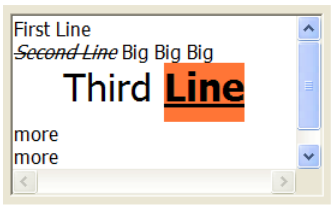
IupToggle



IupText



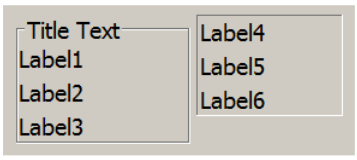
Using FORMATTING:



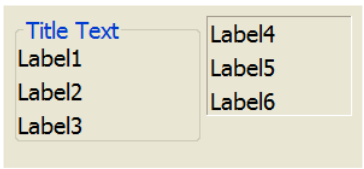
When SPIN=YES:

IupFrame

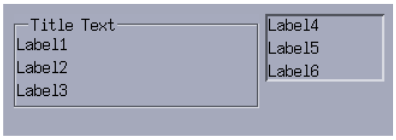
Windows 2000



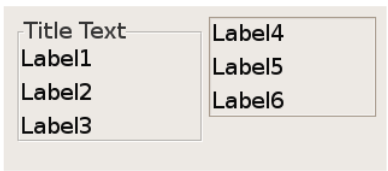
Windows XP



Motif

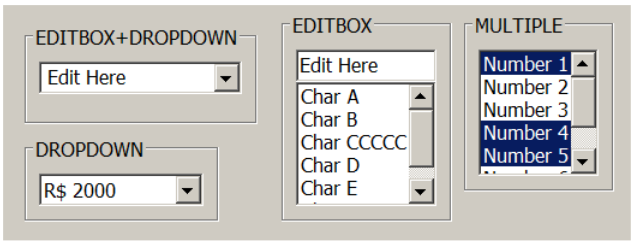


GTK

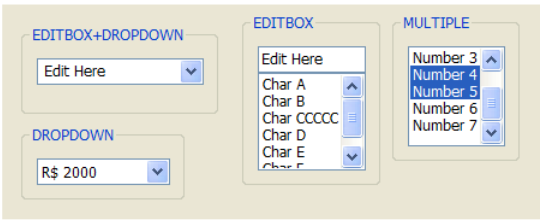


IupList

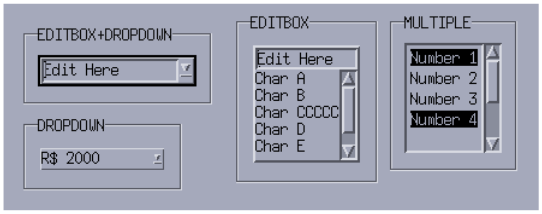
Windows 2000



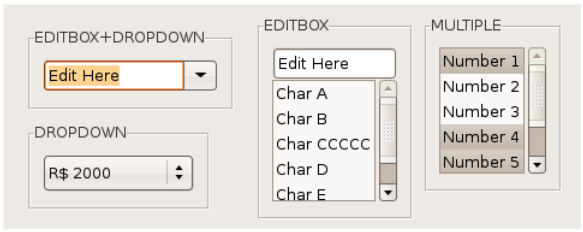
Windows XP



Motif

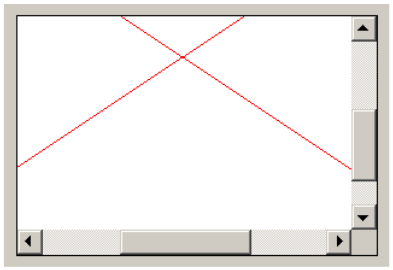


GTK

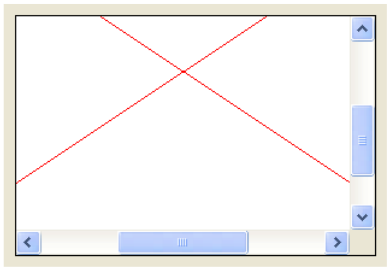


[IupCanvas]

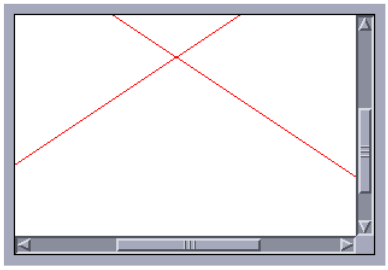
Windows Classic



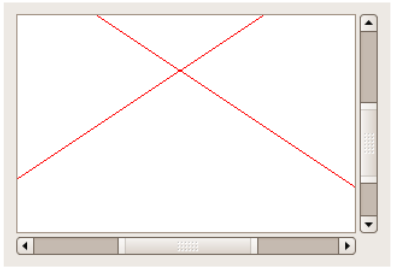
Windows XP Style



Motif



GTK

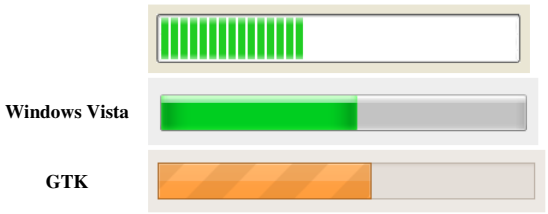


[IupProgressBar]

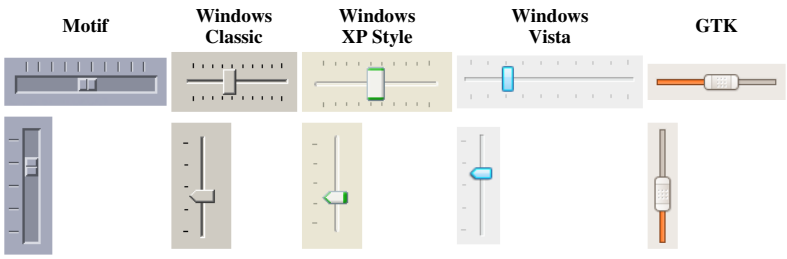
Motif

Windows Classic

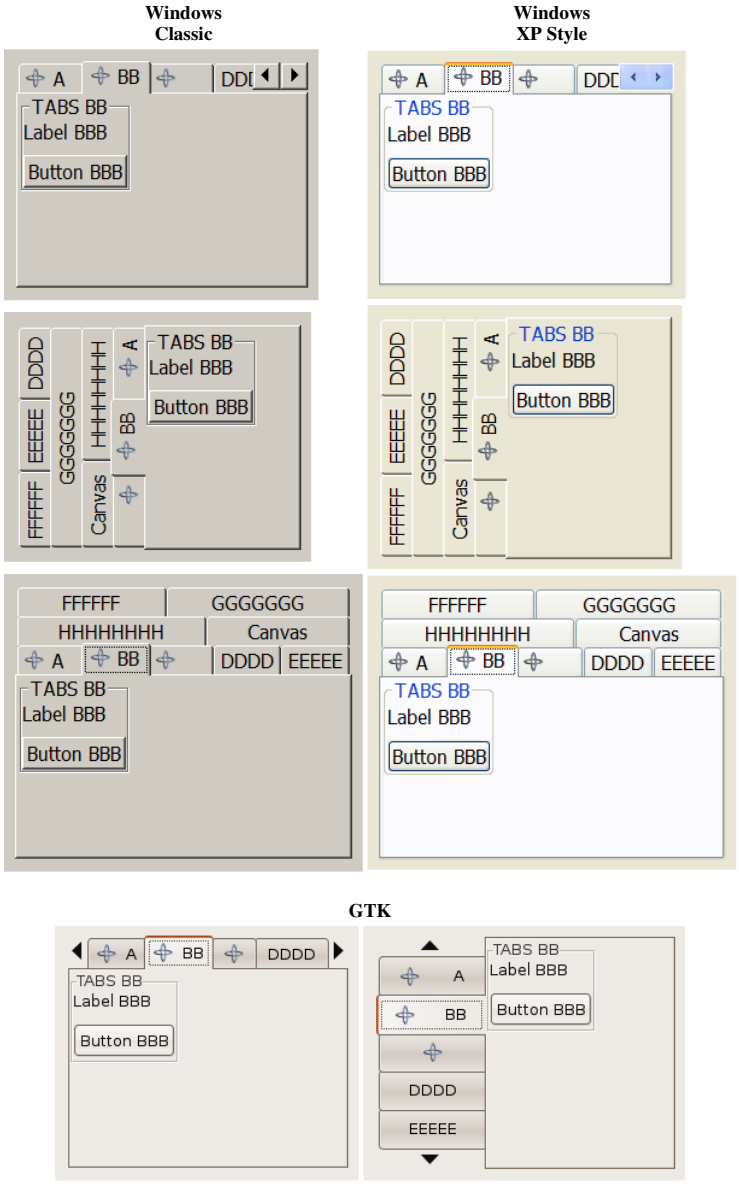
Windows XP Style

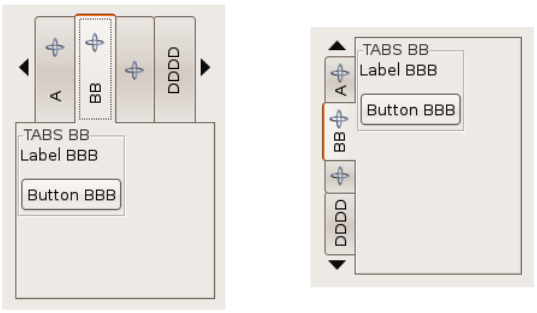


IupVal

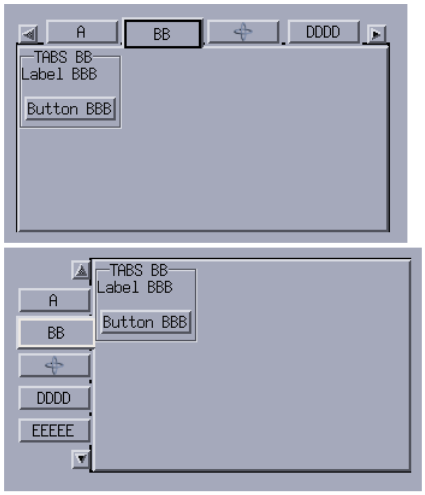


IupTabs

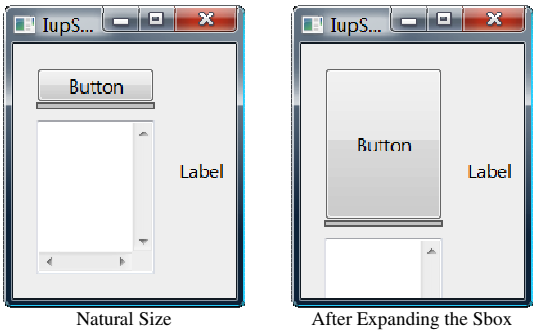




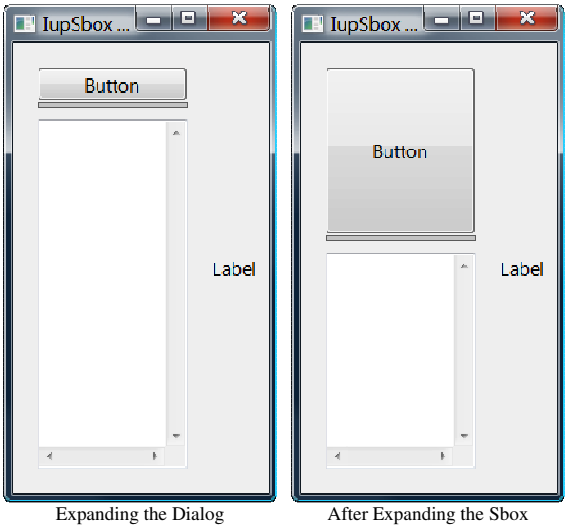
Motif



IupSbox



Natural Size | After Expanding the Sbox

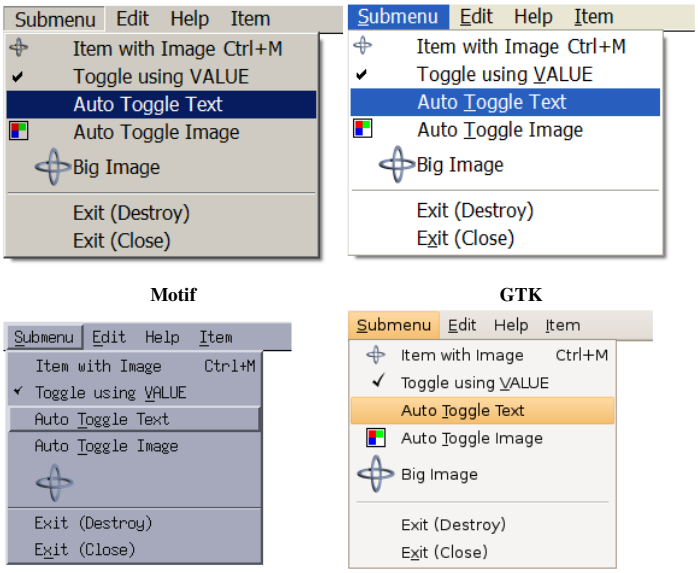


Expanding the Dialog | After Expanding the Sbox

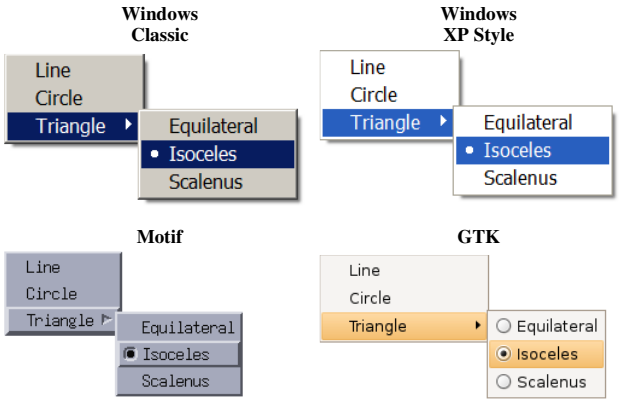
Resources

IupMenu, IupSubmenu and IupItem

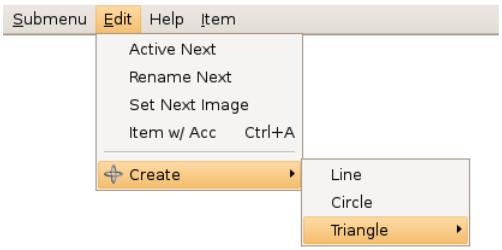
Windows Classic	Windows XP Style



The **IupItem** check is affected by the RADIO attribute in **IupMenu**:



Several Submenus:



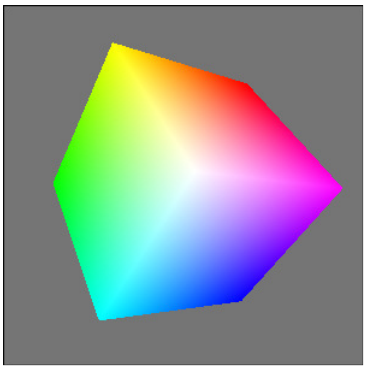
IupImage

See also the IupImgLib, a library of pre-defined images.

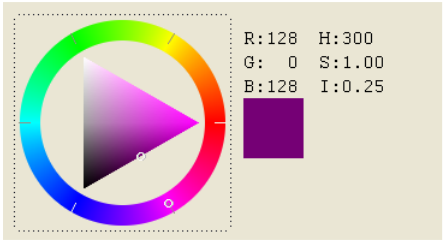
Gallery

Additional Controls

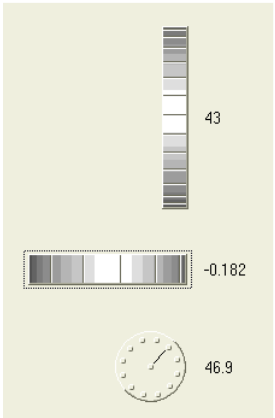
IupGLCanvas



IupColorBrowser



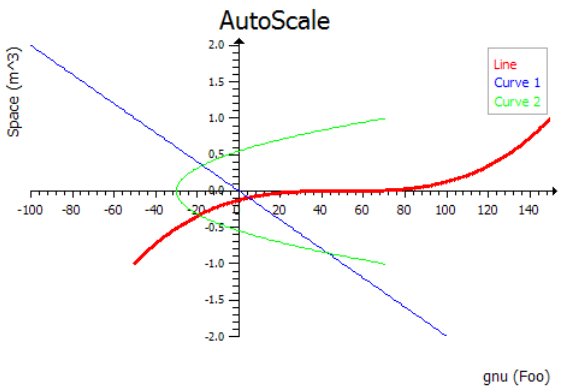
IupDial

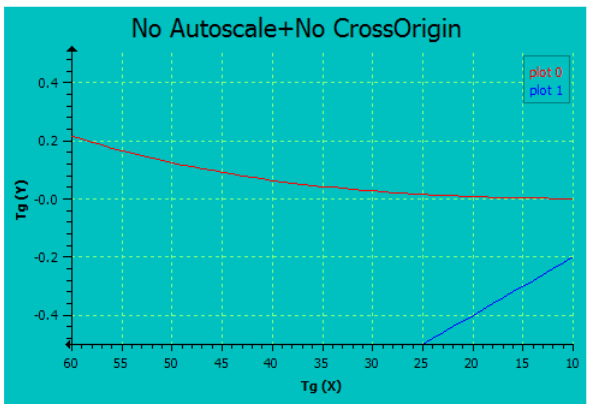


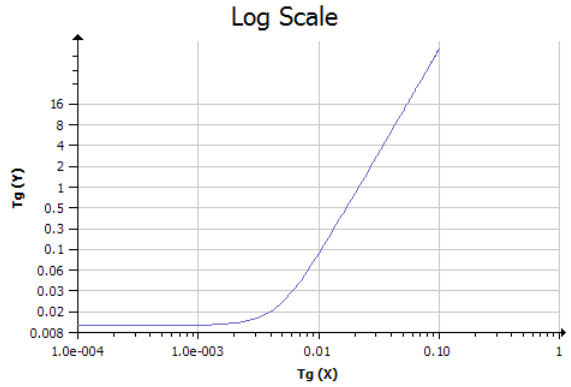
IupMatrix

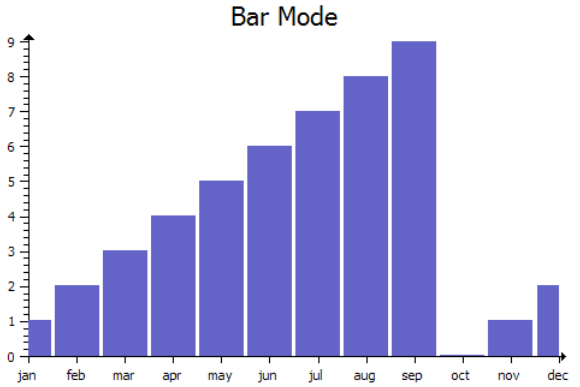
Inflation	January 2000	February 2000
Medicine Pharma	5.6 3.33	4.5
Food	2.2	8.1
Energy	7.2	3.4

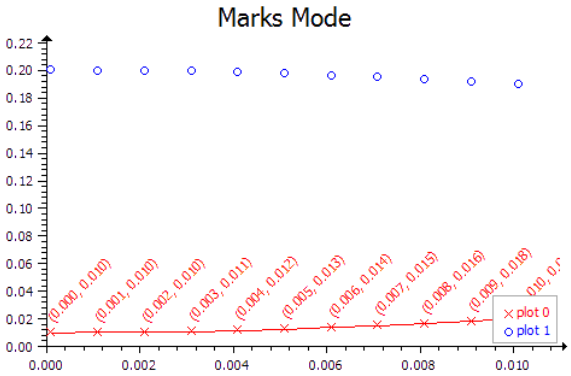
IupPPlot

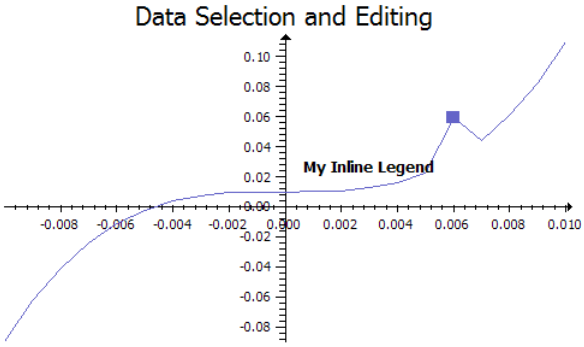






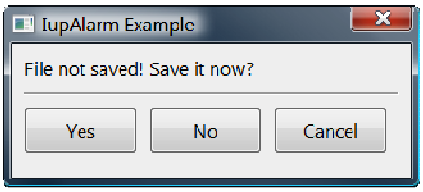






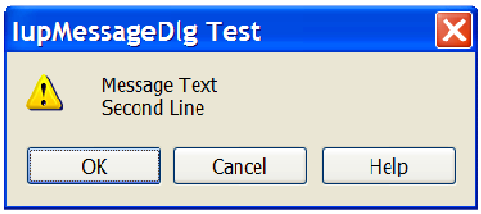
Pre-defined Dialogs

IupAlarm



IupMessageDlg

Windows XP

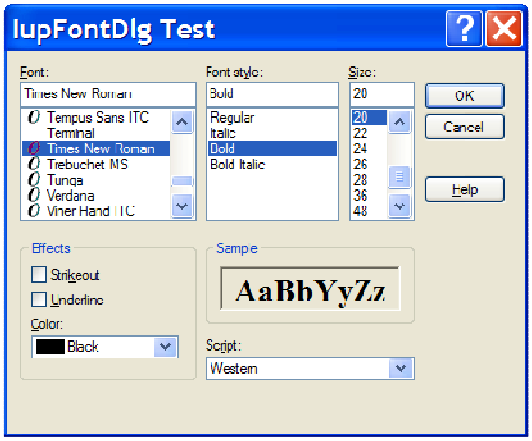


Motif/Mwm

GTK/GNOME

IupFontDlg

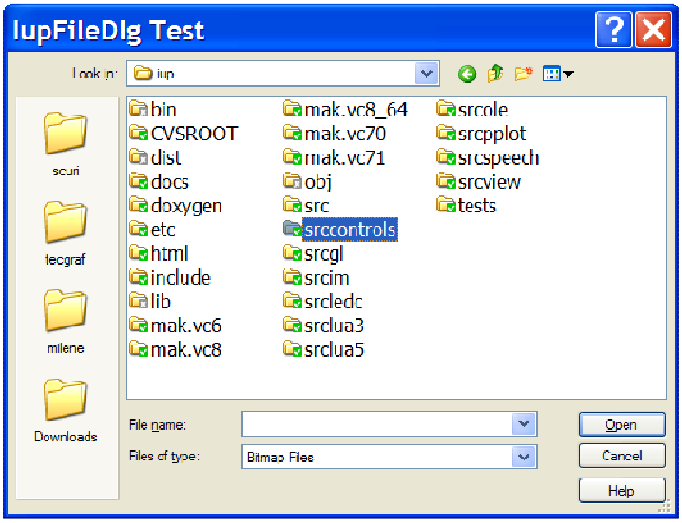
Windows XP



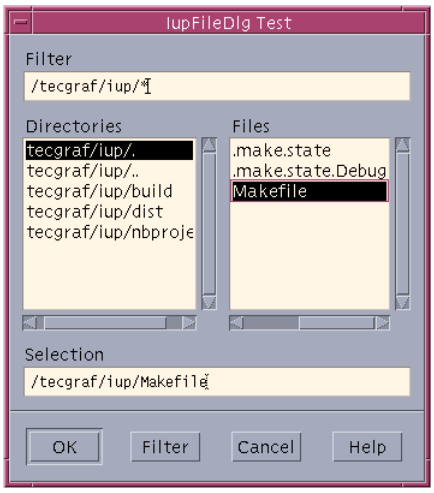
GTK/GNOME

IupFileDlg

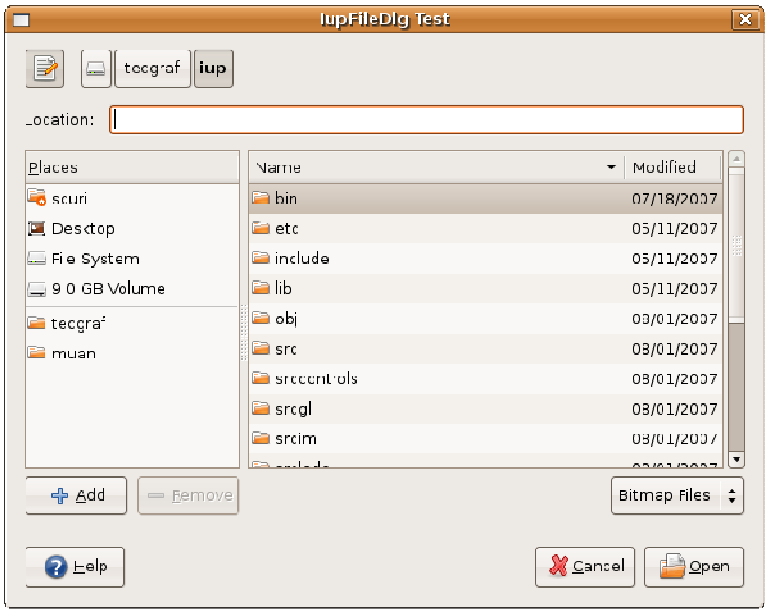
Windows XP



Motif/Mwm

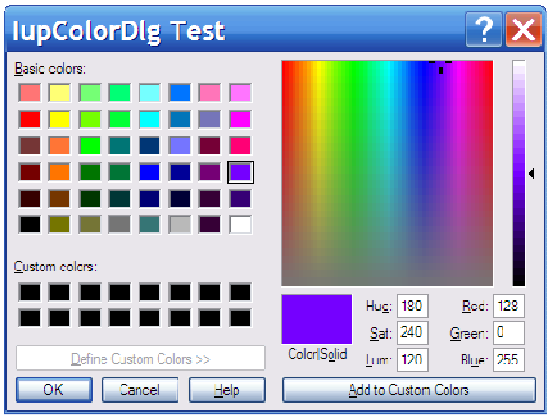


GTK/GNOME

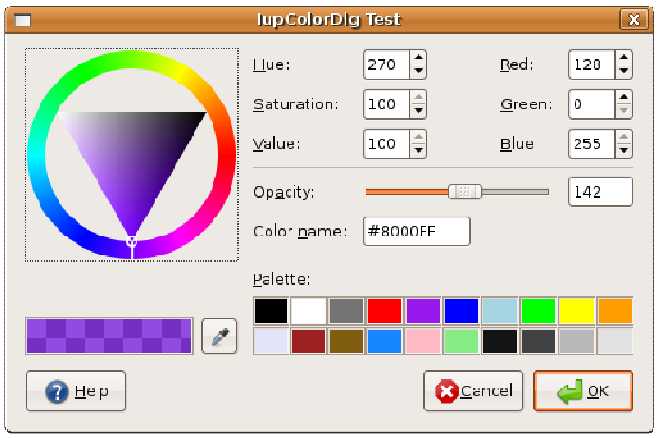


IupColorDlg

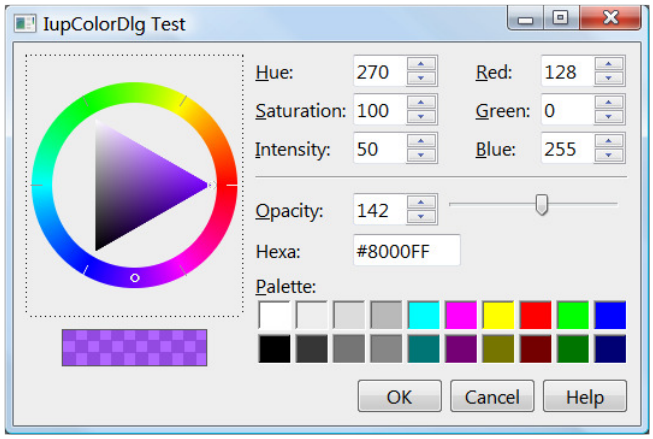
Windows XP



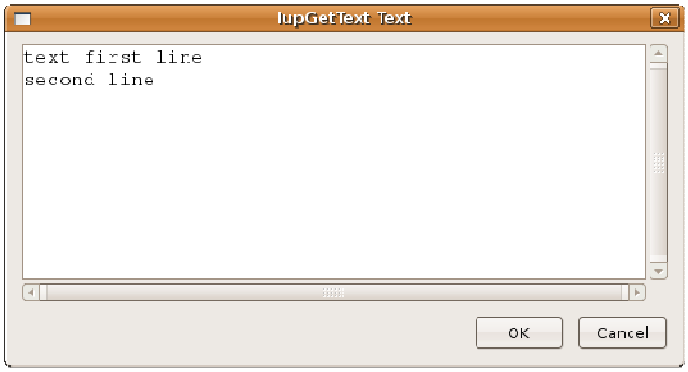
GTK/GNOME



ColorBrowser Based

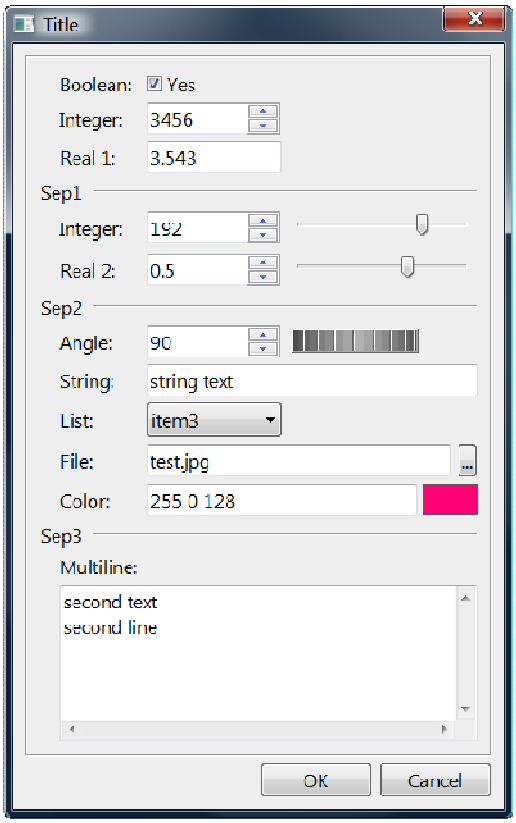


IupGetText



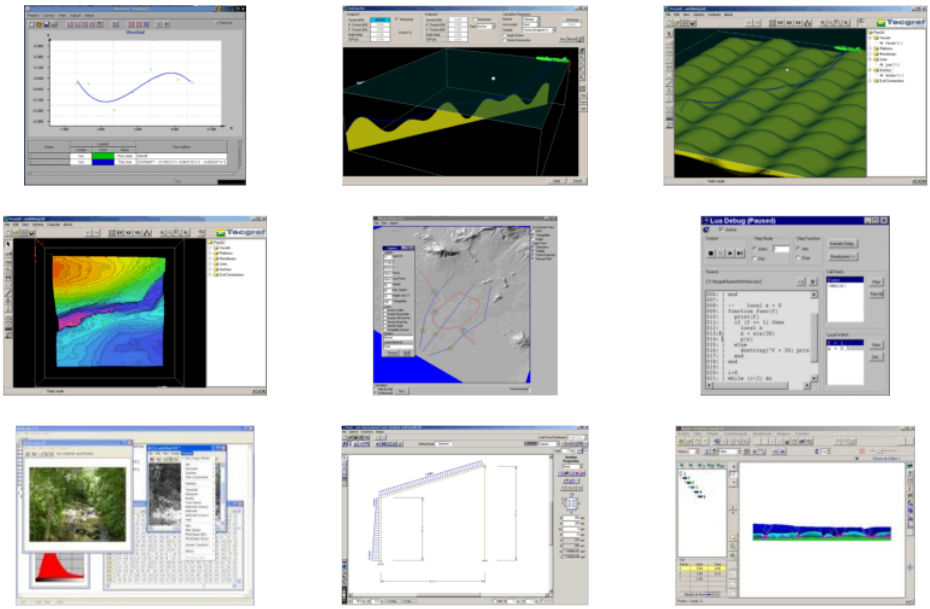
IupListDialog

IupGetParam

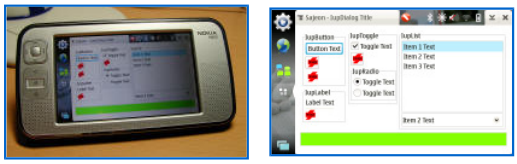


Screenshots

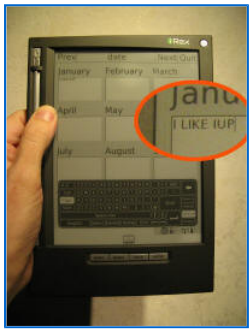
Click on the picture to enlarge image.

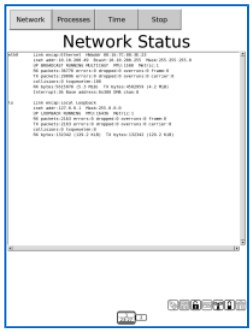


IUP 3.0 beta running on a Nokia N800 Internet Tablet using the GTK driver (contribution by Otfried Cheong):



IUP 3.0 beta running on a Irex Iliad Book Reader using the GTK driver (contribution by Hans Elbers):





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** library must be linked.

To use the additional controls you will need the **iupcontrols** and **iupcd** libraries and the CD library **cd**.

Other controls are available in secondary libraries, they also may have other external dependencies, check the documentation of the control.

To use the Lua Binding, you need to link the program with the **iuplua** library and with the **lua** library. The other secondary libraries also have their Lua binding libraries that must be linked to use the control in Lua.

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 and to enable Windows XP Visual Styles. **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), Code Blocks 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, but you may require additional options:

Linux	-L/usr/X11R6/lib -I/usr/X11R6/include
IRIX	-L/usr/lib32 (X11) -L/usr/Motif-2.1/lib32 -I/usr/Motif-2.1/include (Motif)

SunOS	-L/usr/openwin/lib -I/usr/openwin/share/include (X11) -L/usr/dt/lib -I/usr/dt/share/include (Motif)

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
```

In UNIX the following INCLUDES paths are necessary:

```
/usr/include/atk-1.0 /usr/include/gtk-2.0 /usr/include/cairo /usr/include/pango-1.0 /usr/include/glib-2.0
and eventually:   /usr/lib/glib-2.0/include /usr/lib/gtk-2.0/include
            or    /usr/lib64/glib-2.0/include /usr/lib64/gtk-2.0/include
```

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 regarless of the system.

Building The Library

In the Downloads you will ne able to find pre-compiled binaries for many platforms, all those binaries were built using Tecmake. Tecmake is a command line multi compiler build tool based on GNU make, available at http://www.tecgraf.puc-rio.br/tecmake. Tecmake is used by all the Tecgraf libraries and many applications.

In UNIX, you do not need to install Tecmake, a compact version of Tecmake for UNIX is already included in the source code package. Just type "make" in the command line on the main IUP folder and all libraries and executables will be build. Set the TECTOOLS_HOME environment variable to the folder were the CD, IM and Lua libraries are installed, by default it will assume "TECTOOLS_HOME=../..".

In Windows, the easiest way to build everything 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*" folders are very easy to understand also. In IUP's main directory, and in each source directory, there are files named *make_uname.bat* that build the libraries using **Tecmake**. To build for Windows using Visual C 7.0 (2005) for example, just execute *"make_uname vc7"* in the iup root folder, or the DLLs with Visual C++ 9 (2008) type *"make_uname dll9"*. The Visual Studio workspaces with the respective projects available in the source package is for debugging purposes only.

IUP runs on many different systems and interact with many different libraries such as Motif, OpenGL, Canvas Draw (CD) and Lua. You have to install some these libraries to build the secondary IUP libraries. Make sure you have all the dependencies for the library you want installed, see the documentation bellow.

If you are going to build all the libraries, the makefiles and projects expect the following directory tree:

```
/mylibs/
        iup/
        cd/
        im/
        lua5.1/
```

IUP_ASSERT can be defined to enable some runtime checks for the main API.

Libraries Dependencies

```
iupwin* -> gdi32 user32 comdlg32 comctl32 ole32 (system - Windows)
iupmot* -> [Xpm Xmu Xext] Xt X11 (system - UNIX)
iupgtk* -> gtk-win32-2.0 gdk-win32-2.0 pangowin32-1.0(system - Windows)
        -> gtk-x11-2.0 gdk-x11-2.0 pangox-1.0 (system - UNIX)
        -> gdk_pixbuf-2.0 pango-1.0 gobject-2.0 gmodule-2.0 glib-2.0 (system - Windows/UNIX)
iupgl -> iup
      -> opengl32 glu32 glaux (system - Windows)
      -> GLU GL (system - UNIX)
iupcd -> iup
      -> cd
iupcontrols -> iupcd
iup_pplot -> iupcd
          -> PPlot (included)
iupim -> iup
      -> im
iupimglib -> iup
iuplua51 -> iup
         -> lua5.1
iupluacd51 -> iuplua51
           -> cdlua51
           -> iupcd
iupluacontrols51 -> iuplua51
                 -> iupcontrols
iupluagl51 -> iuplua51
           -> iupgl
iupluaim51 -> iuplua51
           -> imlua51
           -> iupim
iupluaole51 -> iuplua51
            -> iupole
iuplua_pplot51 -> iuplua51
               -> iup_pplot
iupole -> iup

(*) In Windows, "iupwin" is called "iup".
    In Linux, Darwin and FreeBSD "iupgtk" is called "iup".
```

```
    In IRIX, AIX and SunOS "iupmot" is called "iup".
```

As a general rule (excluding system dependencies): IUP depends on CD and IM, and IUPLua depends on Lua, CDLua and IMLua. Notice that not all IUP libraries depend on CD and IM.

Instead of building all the libraries, try building only the libraries you are going to use. The Makefile on the root folder will build all the libraries, but in each source folder there are secondary Makefiles. We use the following source code structure:

```
iup/
    src/         - The core library. Motif, GTK and Windows code
    srccd/       - CD_IUP canvas driver for the CD library
    srcconsole/  - Lua interpreter executable with pre-loaded IUP, CD and IM libraries
    srcgl/       - IupGLCanvas
    srcim/       - IUP/IM utilities
    srcimglib/   - Image Libraries with Icons, Logos and Bitmaps
    srcledc/     - ledc executable
    srclua3      - Lua3 binding, used at Tecgraf by some old applications
    srclua5/     - Lua 5 binding
    srcole/      - IupOleControl
    srcpplot/    - IupPPlot
    srcview/     - IupView executable
```

The Lua bindings for IUP, CD and IM (Makfiles and Pre-compiled binaries) depend on the LuaBinaries distribution. So if you are going to build all use the **LuaBinaries** source package also, not the **Lua.org** original source package. If you like to use another location for the Lua files overwrite the LUAINC, LUALIB and LUABINDIR definitions before using Tecmake or Tecmake Compact.

In Ubuntu you will need to install the following packages and their dependencies (they are not installed by default):

```
libgtk2.0-dev (for the GTK driver)
libmotif-dev (for the Motif driver, if used)
libgl1-mesa-dev (for the IupGLCanvas)
libgul-mesa-dev (for the IupGLCanvas)
```

Using IUP in C++

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

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

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

public:
  iupDialog(Ihandle* child)
  {
    hDlg = IupDialog(child);
    IupSetAttribute(hDlg, "iupDialog", (char*)this);
    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, see next on **Contributions**.

Contributions

All the contributions use the same license terms of the IUP license.

RSSGui by Danny Reinhold. (RSS_GUI_0_5.zip)

> 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.*

IupTreeUtil by Sergio Maffra and Frederico Abraham. (IupTreeUtil3.zip or IupTreeUtil3.tar.gz)

It is an utility wrapper for the **IupTree** control. It is limited to add leaves after all branches inside a branch. It uses the STL.

IupAsync by Ross Berteig

Described by his words:
IUP is not designed to be accessed from multiple threads, but occasionally there is a need (especially in a multi-threaded application) for the UI to update a display or dispatch an action in response to messages from other threads or from an OS component. To address this need, we designed an IUP control that translates calls from any application thread into a callback function guaranteed to be running in IUP's thread.
The IupAsync control is presently an alpha release proving the concept for the Windows platform only. It is intended that it be ported to the other platforms supported by IUP (GTK and Motif for Linux and OSX).

Ruby-IUP by Heesob Park

ruby-iup is an extension module for Ruby that provides an interface to the IUP GUI toolkit.

IupEu by Jeremy Cowgar

Iup wrapped for Euphoria.

A Basic Guide to using IupLua by Steve Donovan

A very nice introductory tutorial for IupLua.

FreeBasic Binding by AGS

The first release of FreeBASIC bindings for IUP version 3.0 (RC2). See the Forum post Portable GUI toolkit (IUP) version 3.0 (RC2)

C++ BuilderX IDE Project Options Guide

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

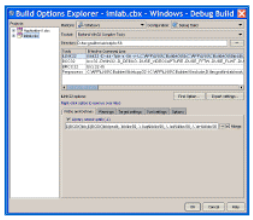
Borland C++ Builder X is an Integrated Development Environment (IDE) for Java and C/C++ languages. It can use several sets of compilers, including the Borland command line compilers version 5.6.

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

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

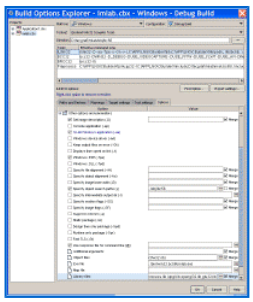
- In the Tools list, click on ILINK32. Then bellow select the Path and Defines tab - there you are going to add the path of the libraries you use, for example:

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



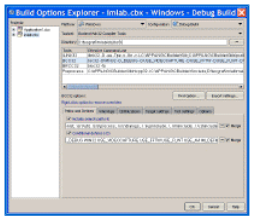
- In the same ILINK32 options, in the tab Options, select Other Options and Parameters, then Library files - there you are going to list the libraries, for example:

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

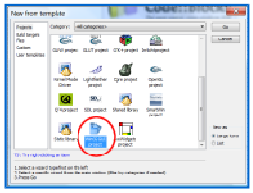


Code Blocks Project Properties Guide

http://www.codeblocks.org/

This guide was built using Code Blocks 8.02 IDE in Windows (but similar configuration can be applied for Linux).

To create a new project go to the menu "File / New / Project" and select "Win32 GUI project":

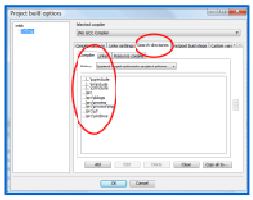


You can use several compilers, for this tutorial we will choose the MingW3 compiler. Just use the respective IUP binaries package: "mingw3".

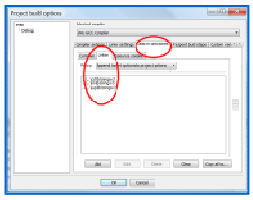
Then remove the automatically added files and add your files to the projet workspace.

After creating the project you must configure it to find the IUP includes and libraries. Go the menu "Project / Build Options".

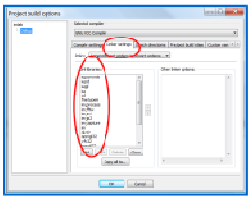
To configure the include files location go to "Search Directories" then in Compiler add the paths you need:



To configure the library files location go to "Search Directories" then in Compiler add the paths you need:



To add the library files go to "Linker Settings" then in "Link libraries" add the files you need:



Dev-C++ IDE Project Options Guide

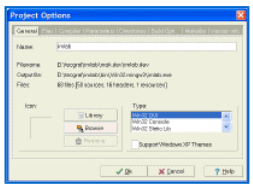
http://www.bloodshed.net/devcpp.html

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

It has many features, and integrated debug and it is free! To use IUP with Dev-C++ you will need to download the "mingw3" binaries in the download page.

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

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

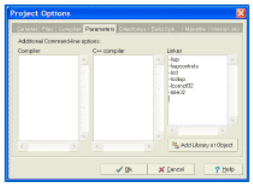


- Parameters / Linker - where you are going to list the libraries you use, for example:

```
-liup
-liupcontrols
-lcd
-liupcd
-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.

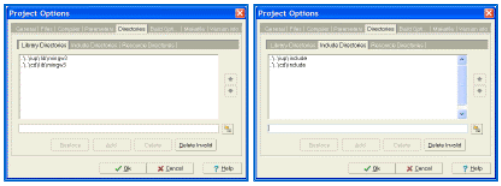


- Directories / Library Directories and Include Directories - where you are going to list the include path, for example:

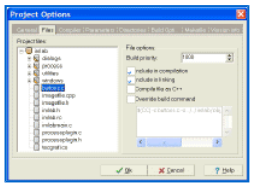
```
..\..\iup\lib\mingw3
..\..\cd\lib\mingw3
or
c:\tecgraf\iup\lib\mingw3
c:\tecgraf\cd\lib\mingw3
```

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++".

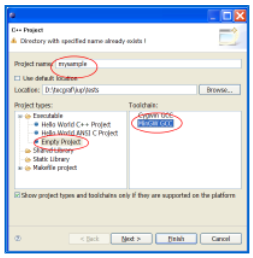


Eclipse for C++ Project Properties Guide

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":

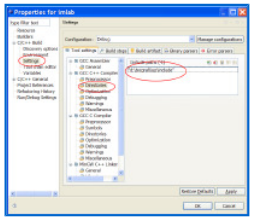


You can use the MingW3 or Cygwin compiler. Just use the respective IUP binaries package: "mingw3" or "gcc3".

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

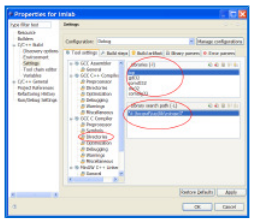
After creating the project you must configure it to find the IUP includes and libraries.

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".



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.

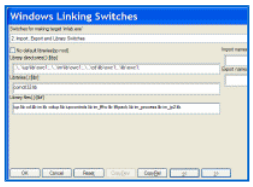
"It is a joint effort between SciTech Software Inc, Sybase and the Open Source development community to maintain and enhance the Watcom C/C++ and Fortran cross compilers and tools. An Open Source license from Sybase allows free commercial and non-commercial use of the Open Watcom tools."

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

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

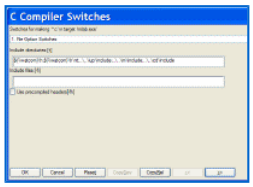
- In the Windows Linking Switches dialog, select option 2. Import, Export and Library Switches. Then enter the Library directories and Library files. For example:

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

comctl32.lib iup.lib iupcontrols.lib cd.lib iupcd.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
```

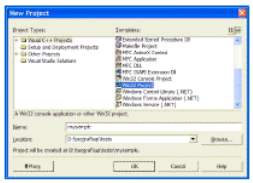


Visual C++ 7 IDE Project Properties Guide

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":



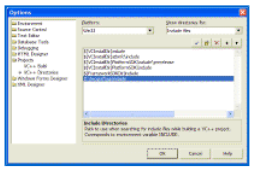
Select "Win32 Project" on the Templates. Before finishing the Wizard, select "Application Settings". Mark "Windows application" and "Empty project".

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 the use standard C printf function to display textual information for debugging purposes.

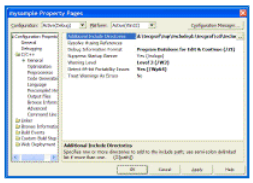
Then add your files in the menu "Project / Add New Item" or "Project / Add Existing Item".

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

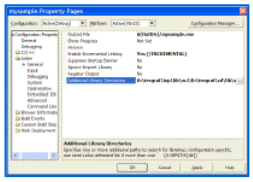
One is in the menu "Tools / Options", then select "Project / Visual C++ Directories". Select "Include Files" or "Library Files" in "Show directories for:". In this dialog you will configure parameters that will affect all the projects you open.



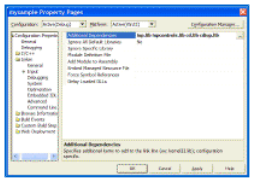
Or you can configure the parameters only for the project you created. In this case go the menu "Project / Properties". To configure the include files location select "C/C++ / General" in the left tree, then write the list of folders separated by ";" in "Additional Include Directories".



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



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

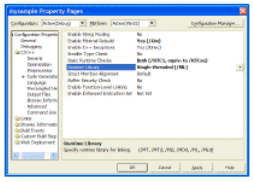


In this sample configuration the project is using the additional library of Controls that uses the CD library, also available at the download page.

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

```
LINK : warning LNK4098: defaultlib 'LIBC' conflicts with use of other libs; use /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)".



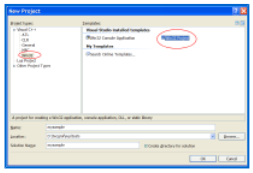
If you want to use multithreading then you must use the DLL version of the IUP libraries. They are built with the "Multi-threaded DLL (/MD)" option. Or you must rebuild the libraries with your own parameters.

Visual C++ 8 IDE Project Properties Guide

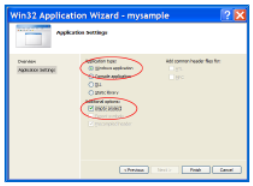
http://msdn2.microsoft.com/en-us/vstudio/default.aspx
http://msdn.microsoft.com/vstudio/express/downloads/ (free version)

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":



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

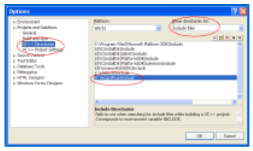


You can also create a "Console application", and whenever you execute your application a text console will also be displayed. This is a very useful situation so you can the use standard C printf functions to display textual information for debugging purposes.

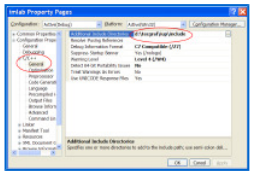
Then add your files in the menu "Project / Add New Item" or "Project / Add Existing Item".

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.

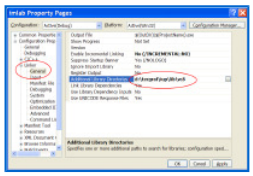
One is in the menu "Tools / Options", then select "Project and Solutions / Visual C++ Directories". Select "Include Files" or "Library Files" in "Show directories for:". In this dialog you will configure parameters that will affect all the projects you open.



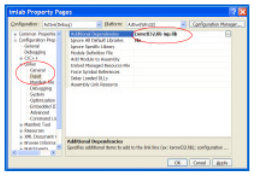
Or you can configure the parameters only for the project you created. In this case go the menu "Project / Properties". To configure the include files location select "Configuration Properties / C/C++ / General" in the left tree, then write the list of folders separated by ";" in "Additional Include Directories".



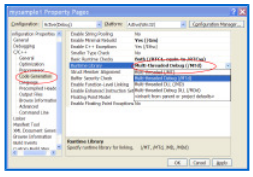
To configure the library files location select "Configuration Properties / Linker / General" in the left tree, then write the list of folders separated by ";" in "Additional Library Directories".



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".



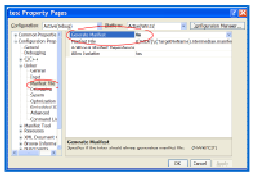
The default configuration use the C run time library with debug information and in a DLL. The standard IUP binary distribution has two packages for Visual Studio 2005 (or Visual C++ 8). Both do not have debug information, but this could be ignored even if a warning appears in the Output log. To change 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)".

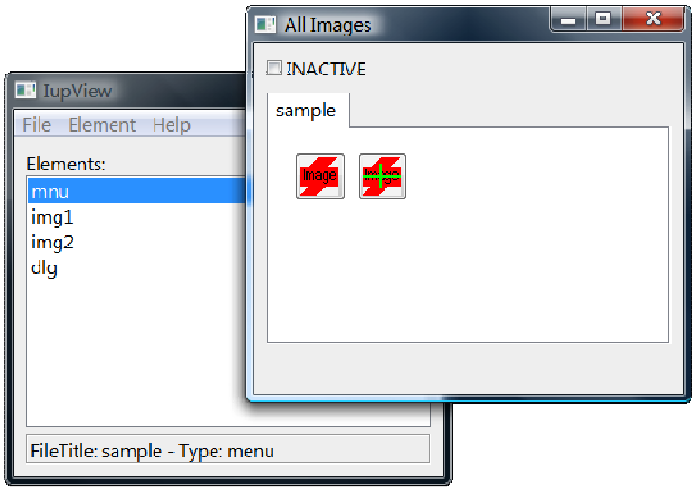
When using the "iup.manifest" from "iup.rc", configure the linker properties of your project to do NOT generate a manifest file or the Windows XP Visual Styles won't work.



Tools Executables

IupView

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 **IupView** application is available in the distribution files source code and pre-compiled binaries at the Download pages.



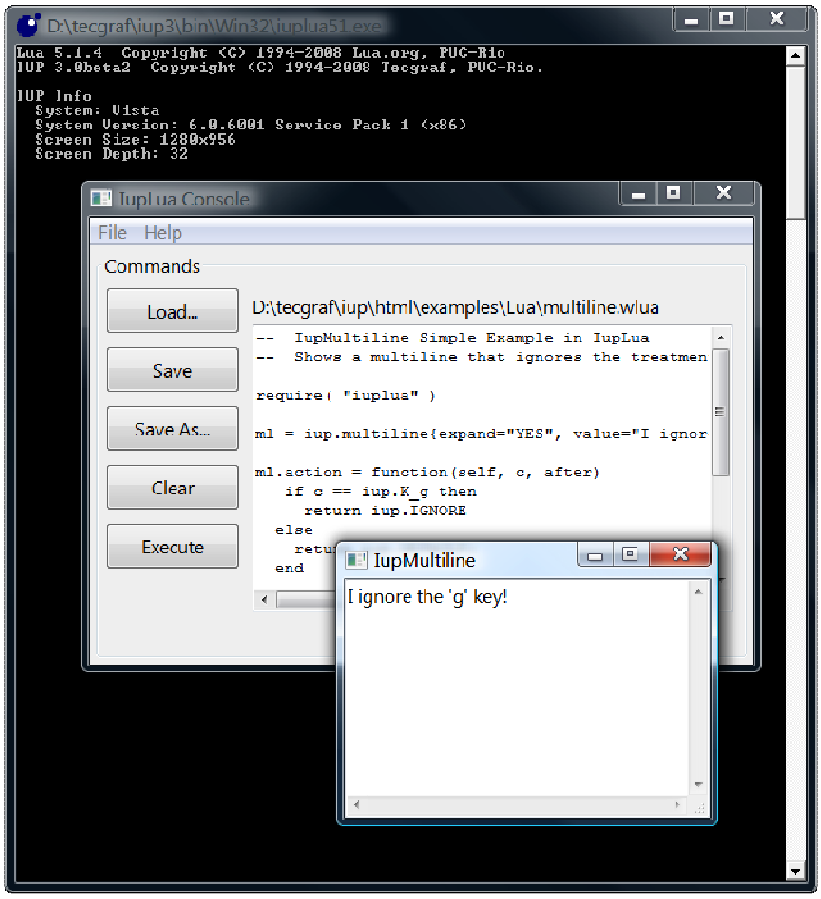
IupLua Console

The **IupLua Console** can load and execute Lua scripts using the IupLua binding. Lua print calls are output in the standard output. The executable package in Windows also includes the CD and IM libraries, and a modified version of the LuaGL library.

The **IupLua Console** is available in the distribution files source code and pre-compiled binaries at the Download pages. The packages are almost ready to be run, but it needs the Visual C++ 2005 Run Time (x86 or x64) installed on the system in Windows, and in UNIX it needs that the LD_LIBRARY_PATH environment variable contains the executable folder, for example:

```
export LD_LIBRARY_PATH=$LD_LIBRARY_PATH:/home/tecgraf/iup3/bin/$TEC_UNAME
```

If you build the executable from sources then you will need to do an addtional step before running it. The easiest one is to copy the dinamic libraries of IUP, CD, IM and Lua to the executable folder, in Windows or in UNIX.



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*)
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.

Complete Samples

Standard Controls

The following example creates a dialog with virtually all of IUP standard elements as well as some variations of them, with some attributes changed. The same example is implemented in C, LED and Lua. 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.wlua

You can see the results in Windows, Motif and GTK on the Sample Results.

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.

Browse for Example Files

External Samples

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

Some freely available applications also use IUP:

IMLAB - Image Processing Laboratory

EdPatt - Pattern Editor

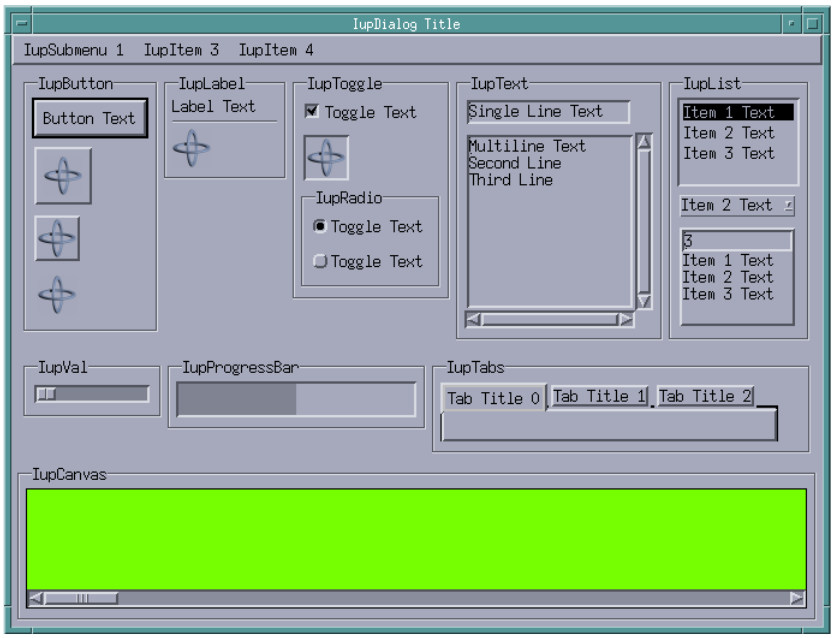
Ftool - Two-dimensional Frame Analysis Tool

The Lua for Windows distribution is a 'batteries included environment' for the Lua scripting language on Windows, that also includes LuaGL, IUP, CD and IM.

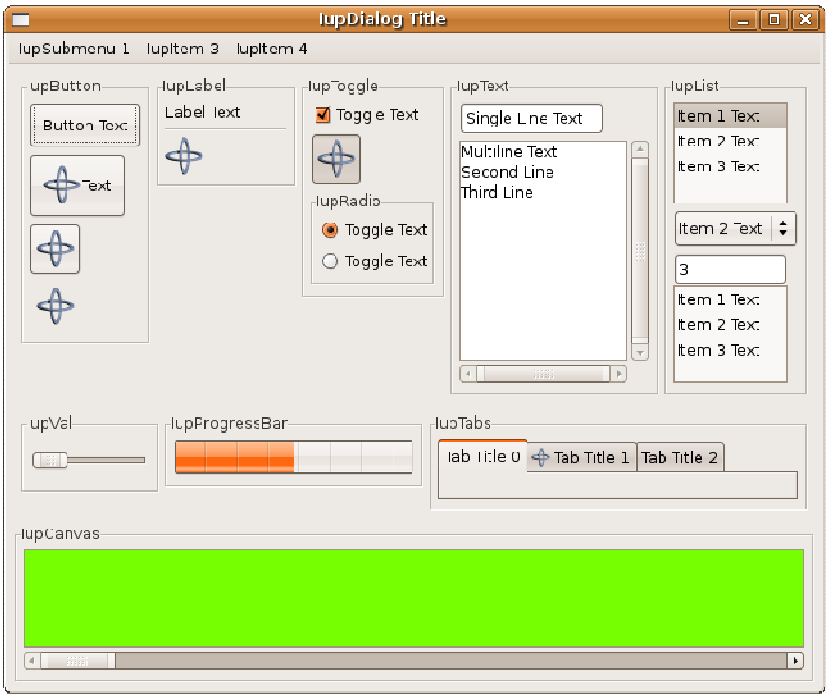
Sample Results (1)

The following screenshots shows the sample.c results. See also the same sample changing the dialog BACKGROUND, the dialog BGCOLOR and the children BGCOLOR.

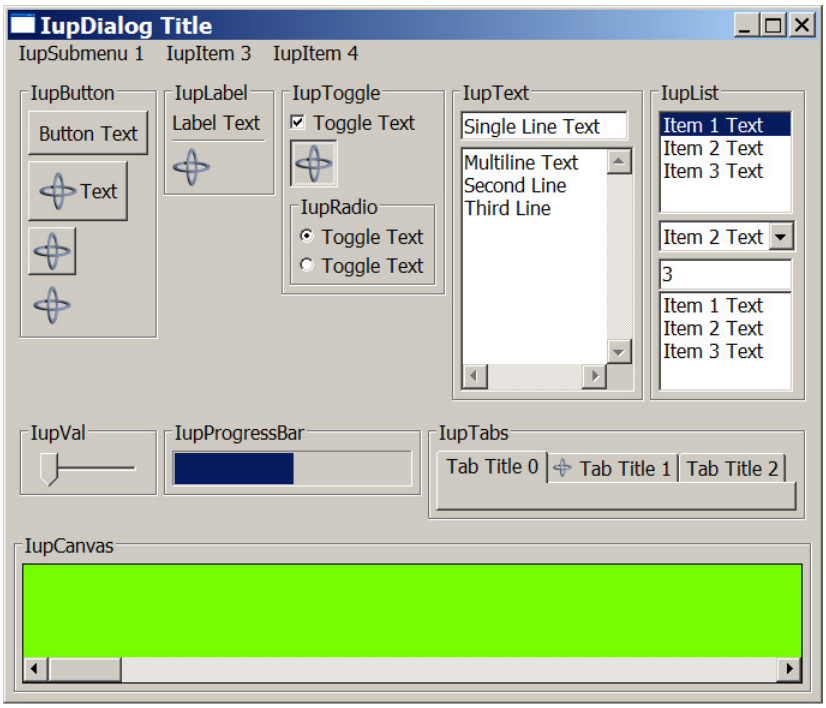
Motif in MWM



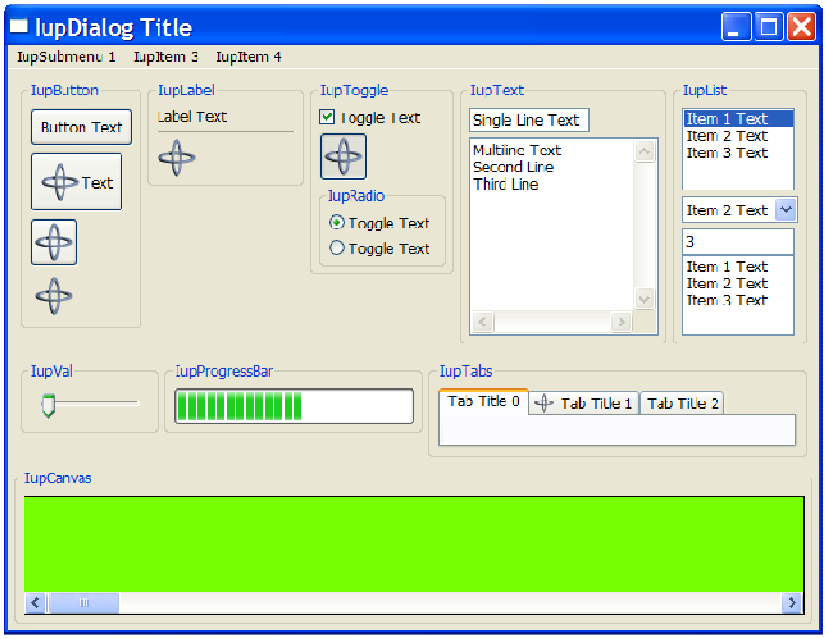
GTK in Gnome



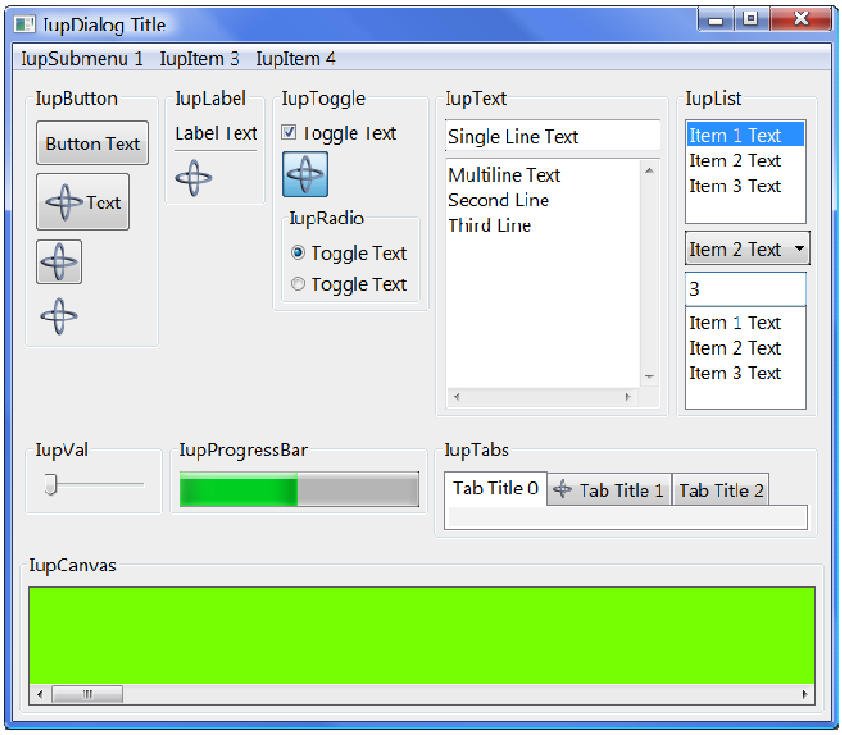
Windows Classic



Windows XP Style



Windows Vista

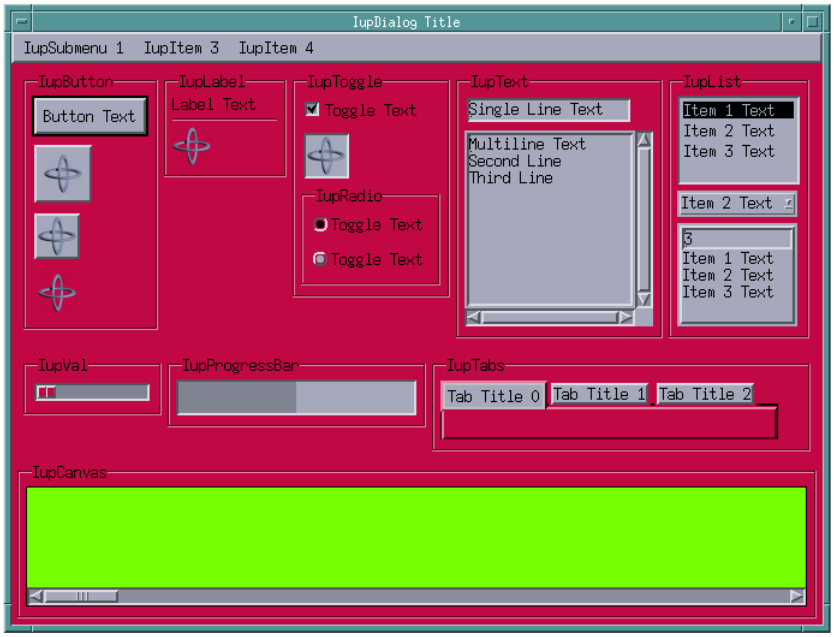


Sample Results (2)

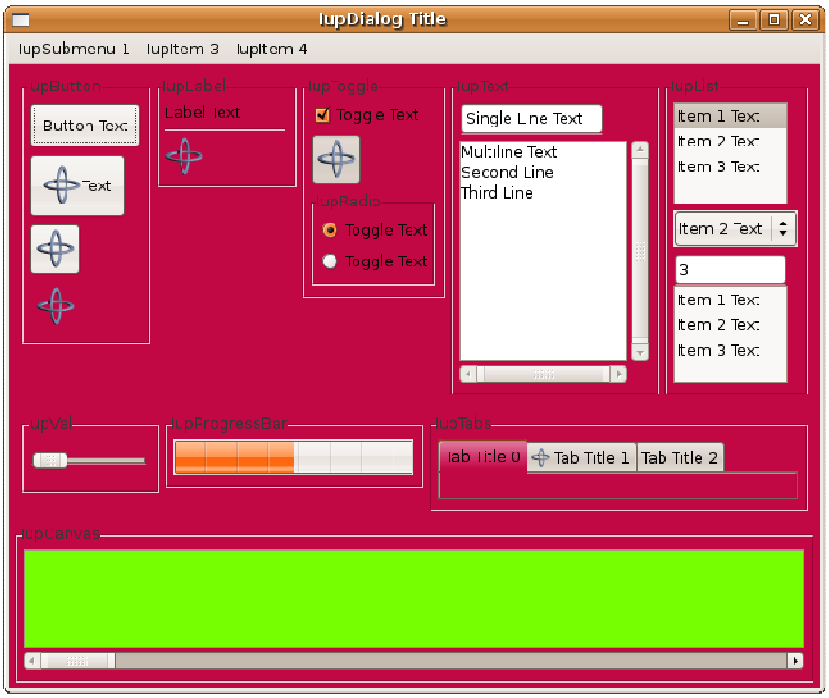
The following screenshots shows the sample.c results when the dialog BACKGROUND attribute is changed. See also the same sample with normal background, the dialog BGCOLOR and the children BGCOLOR.

Notice that the dialog BACKGROUND attribute affects only the background of the dialog, the background of each control is preserved except for the ones that the background is transparent.

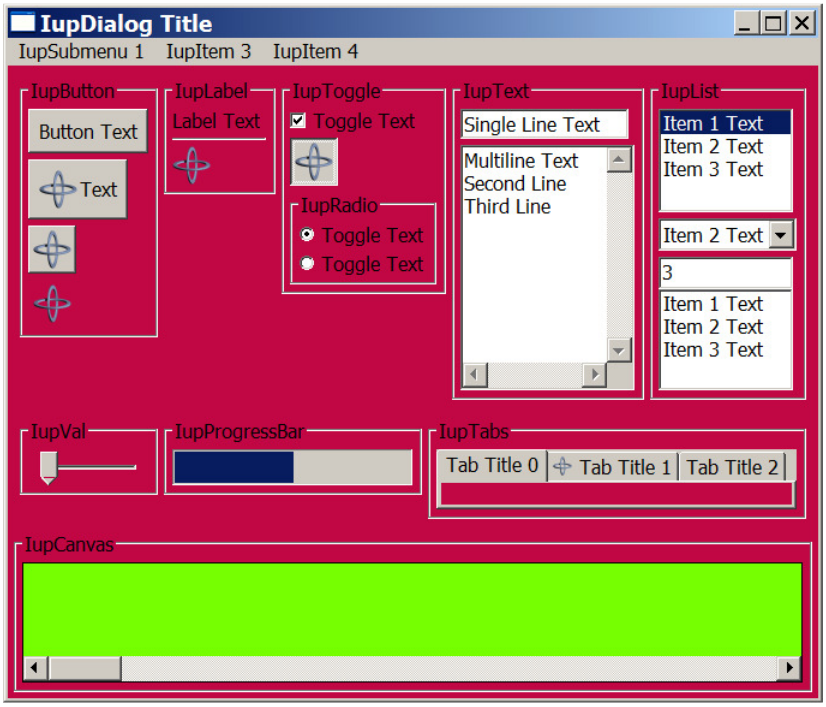
Motif in MWM



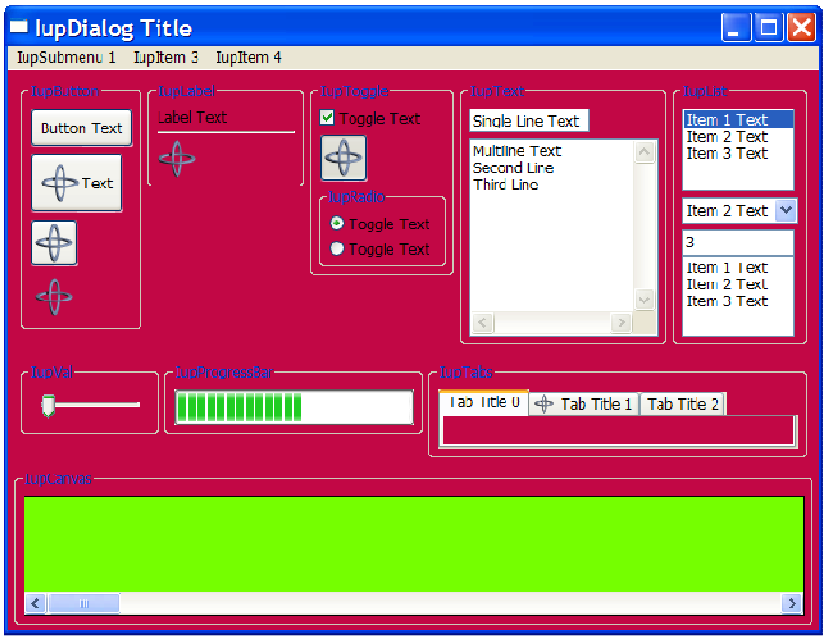
GTK in Gnome



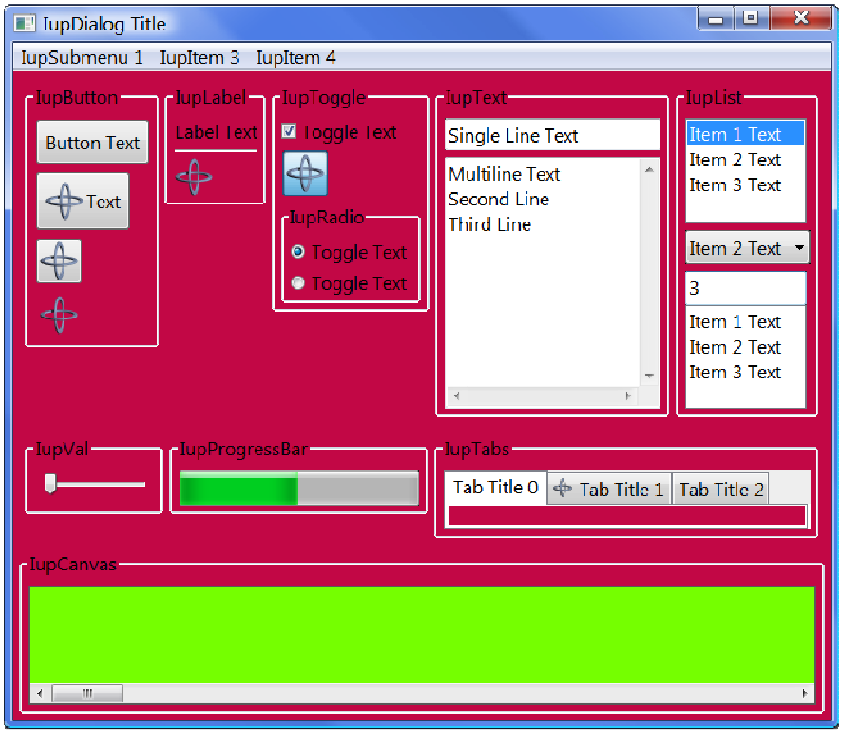
Windows Classic



Windows XP Style



Windows Vista

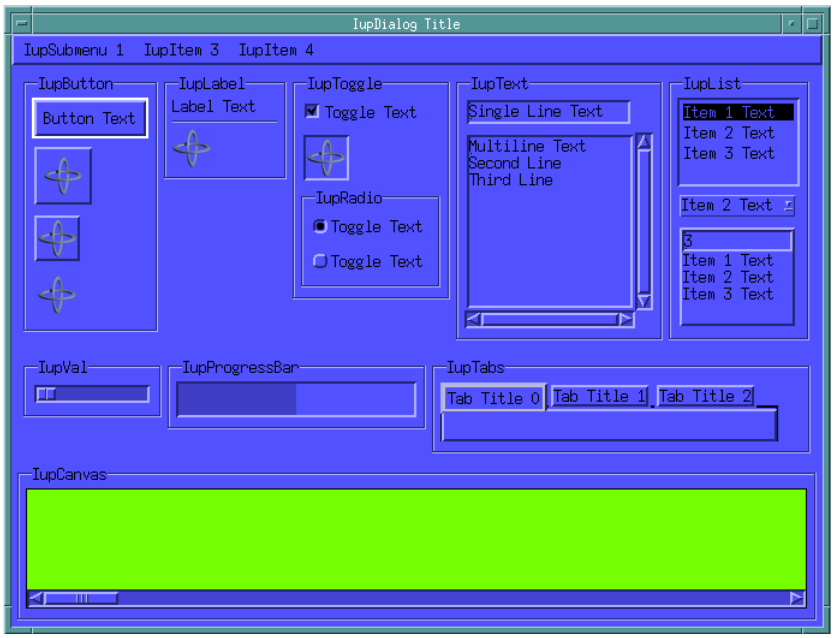


Sample Results (3)

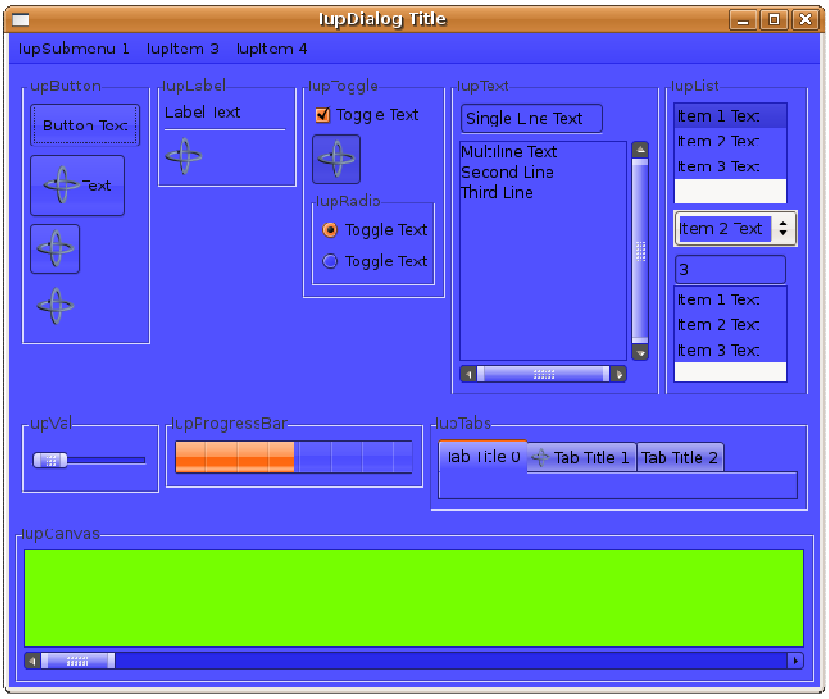
The following screenshots shows the sample.c results when the dialog BGCOLOR attribute is changed. See also the same sample with normal background, changing the dialog BACKGROUND and the children BGCOLOR.

Since BGCOLOR is an inheritable attribute changing it at the dialog affects all controls. And notice that on Windows the BGCOLOR is ignored for several controls.

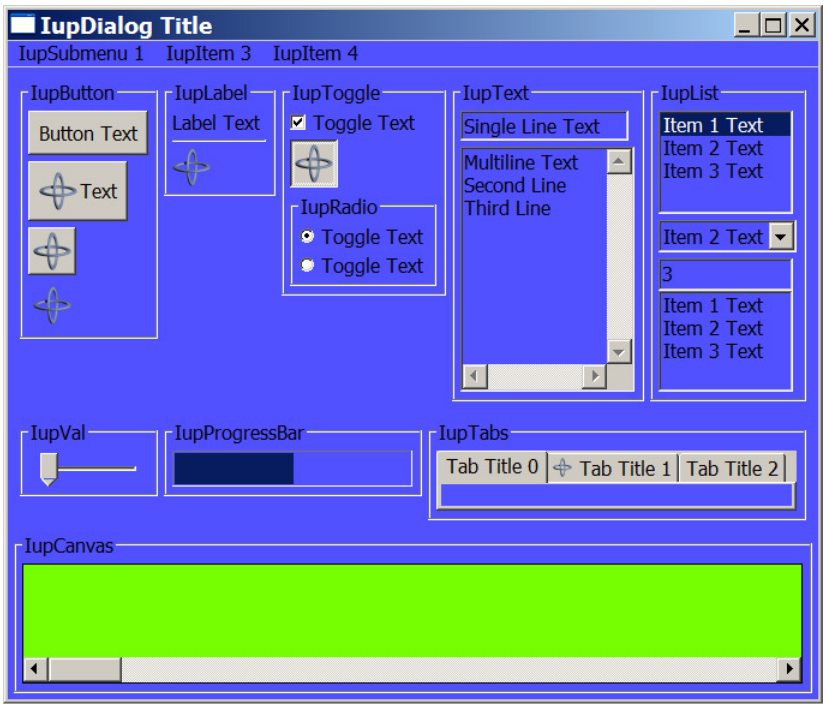
Motif in MWM



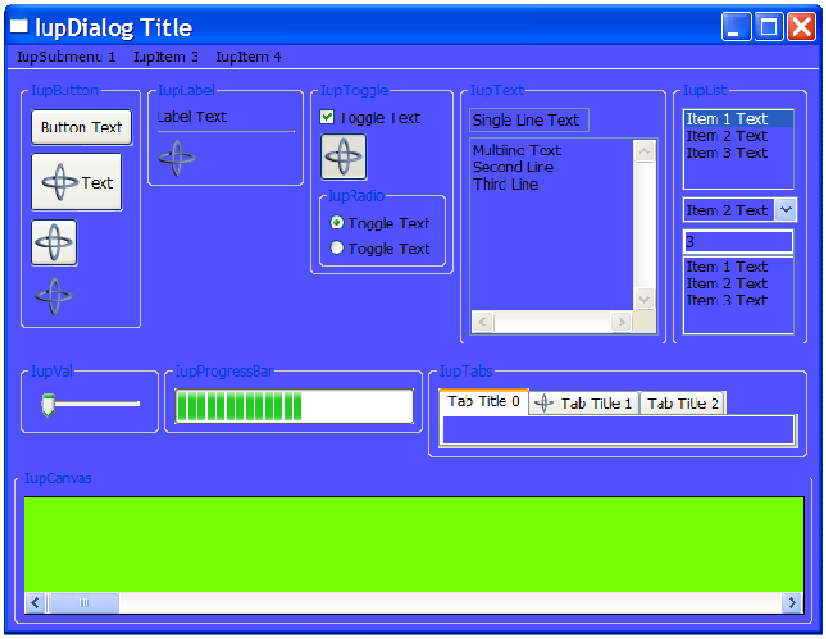
GTK in Gnome



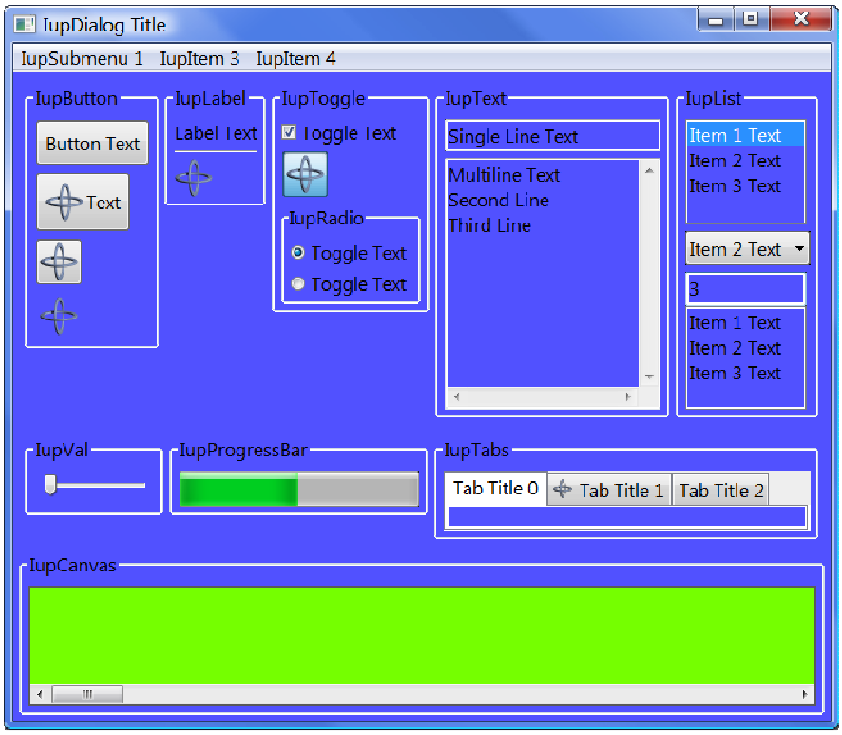
Windows Classic



Windows XP Style



Windows Vista

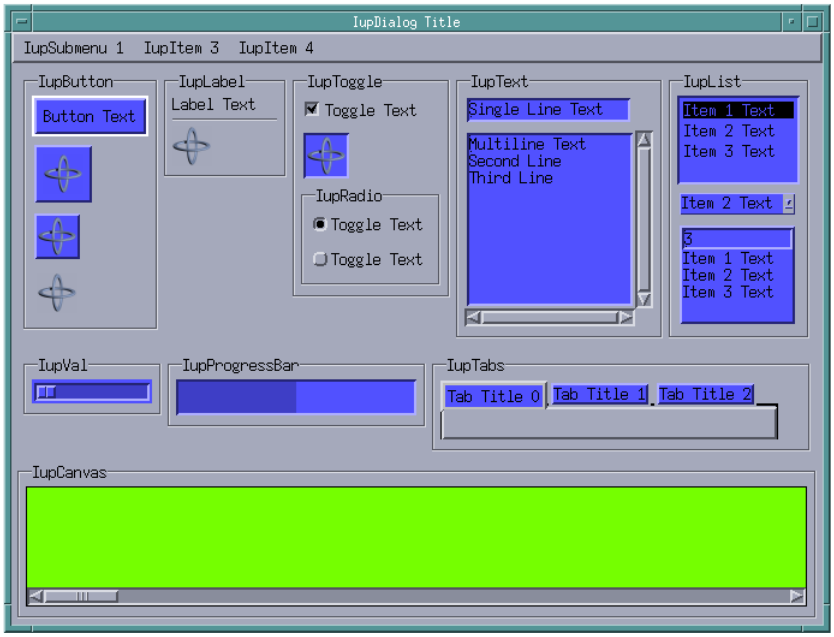


Sample Results (4)

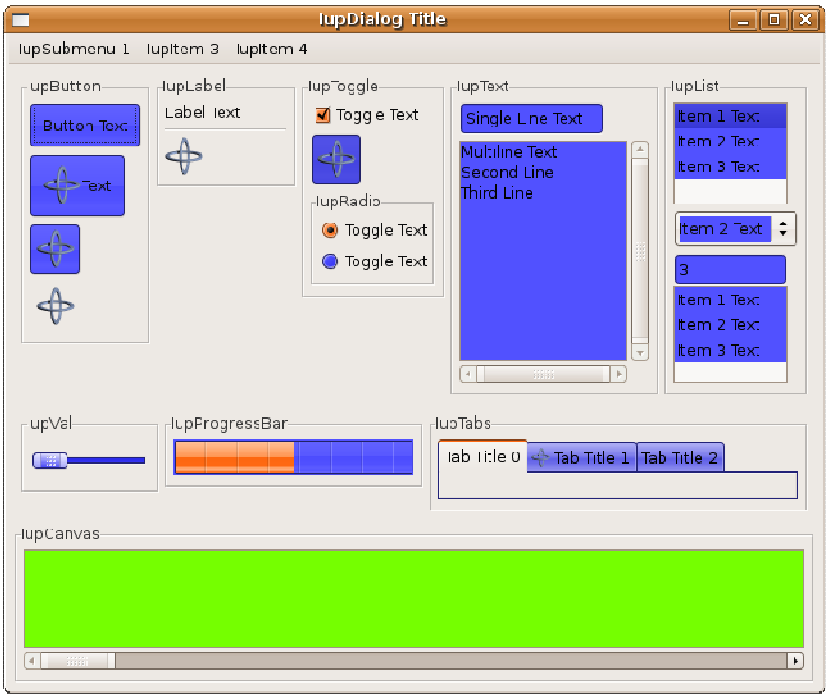
The following screenshots shows the sample.c results when the BGCOLOR attribute of the dialog children is changed, but NOT the dialog. See also the same sample with normal background, changing the dialog BACKGROUND and the dialog BGCOLOR.

In this case, the BGCOLOR attribute affects only the controls. Also notice that the transparent area of the controls are not affected. And notice that on Windows the BGCOLOR is ignored for several controls.

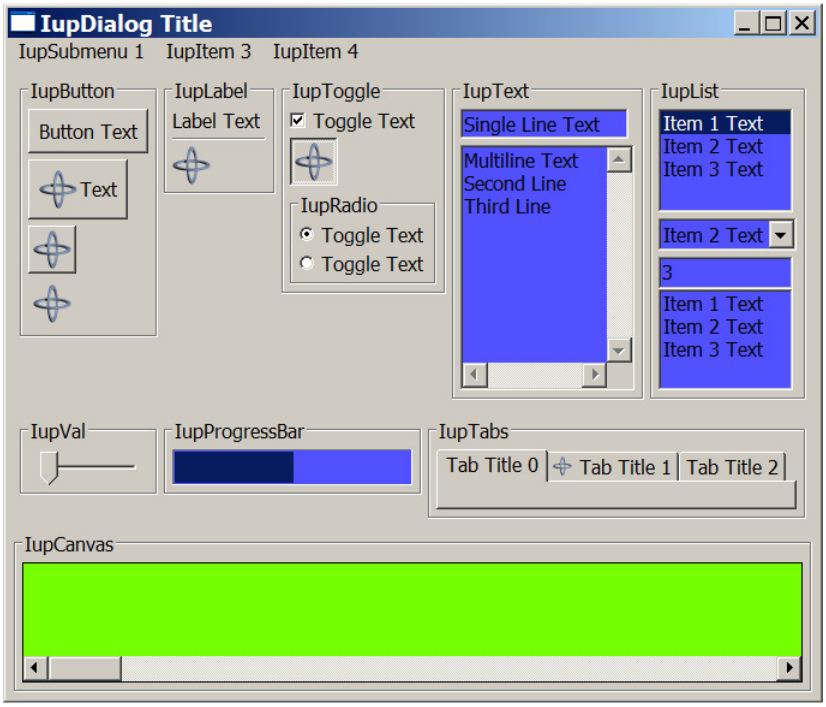
Motif in MWM



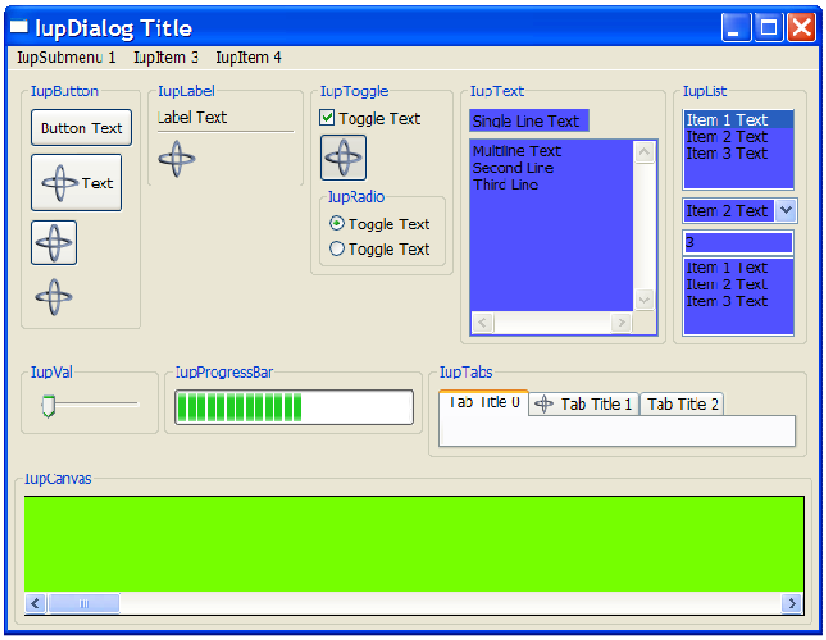
GTK in Gnome



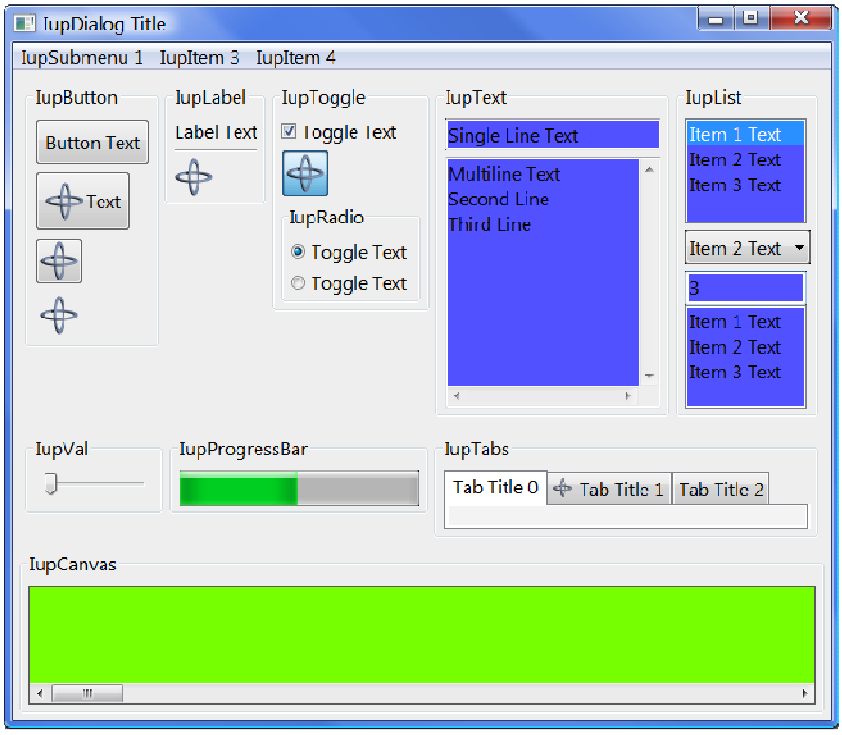
Windows Classic



Windows XP Style



Windows Vista



Lua Binding

Overview

All the IUP functions are available in Lua, with a few exceptions. We call it **IUPLua**. To use them the general application will do require"iuplua", and require"iupluaxxxx" to all other secondary libraries that are needed. The functions and definitions will be available under the table "iup" using the following name rules:

```
iupXxx  -> iup.Xxx    (for functions)
IUP_XXX -> iup.XXX    (for definitions)
```

All the metatables have the "tostring" metamethod implemented to help debuging.

Also the functions which receive values by reference in C were modified. Generally, the values of parameters that would have their values modified are now returned by the function in the same order.

Notice that, as opposed to C, in which enumeration flags are combined with the bitwise operator OR, in Lua the flags are added arithmetically.

In Lua all parameters are checked and a Lua error is emitted when the check fails.

All the objects are NOT garbage collected by the Lua garbage collector, you must manually call **iup.Destroy** or elem:**destroy**, if you would like to destroy an element.

In Iup additional features were created for the Lua Binding using the metamethods. Attributes and callbacks can be set and get in a much more natural way:

```
IupSetAttribute(label, "TITLE", "test")   >>   label.title = "test"            (names are in lower case)
title = IupGetAttribute(label, "TITLE")   >>   title = label.title
IupSetCallback(button, "ACTION", button_action_cb);   >>    function button:action() ... end
```

Also the element constructors were changed so you can use tables to initialize their parameters and attributes:

```
IupButton("test")         >>    iup.button{title = "test", alignment="acenter"}
IupHbox(bt1, bt2, NULL)   >>    iup.hbox{bt1, bt2, margin="10x10"}
```

Lua was created after **LED**, so that's why **LED** exists. Since we have many application still using LED, its support will continue in IUP. Today **IupLua** completely replaces the LED functionality and adds much more.

The distribution files include an executable called **iuplua51**, that you can use to test your Lua code. It has support for all the addicional controls, for IM, CD and OpenGL calls. It is available at the Download.

IupLua Initialization

Lua 5.1 "require" can be used for all the **IupLua** libraries. You can use **require**"**iuplua**" and so on, but the LUA_CPATH must also contains the following:

```
"./lib?51.so;"    [in UNIX]

".\\?51.dll;"     [in Windows]
```

The LuaBinaries distribution already includes these modifications on the default search path.

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.

Additionally you can statically link the **IupLua** libraries, but you must call the initialization functions manually. The **iuplua_open** function is declared in the header file **iuplua.h**, see the example below:

```
#include <lua.h>
#include <lualib.h>
#include <lauxlib.h>
#include <iuplua.h>

void main(void)
{
  lua_State *L = lua_open();

  luaopen_string(L);
  luaopen_math(L);
  luaopen_io(L);

  iuplua_open(L);
  lua_dofile("myprog.lua");

  lua_close(L);
}
```

When using **Lua** the Iup initialization functions, **IupOpen**, **IupControlsOpen** and others, are not necessary. The initialization is automatically done inside the respective **IupLua** initialization function.

Embedding Lua files in the Application Executable

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:

```
luac5.1 -o myfile.lo myfile.lua
bin2c5.1 myfile.lo > myfile.loh
```

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

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

More Information

Steve Donovan wrote a very nice "A Basic Guide to using IupLua" that was included in Lua for Windows. It is now available as part of the IUP documentation.

The slides for **"Tecgraf Development Tools: IUP, CD and IM"** presented at the Lua Workshop 2009 are also available for Download [iupcdim_wlua2009_en.pdf].

A Basic Guide to using IupLua
by Steve Donovan

IupLua is a cross-platform kit for creating GUI applications in Lua. There are particularly powerful facilities for getting user input that don't require complicated coding, so it is particularly good for utility scripts.

Attributes are an important concept in IUP. You set and get them just like table fields, but they are different from fields in several crucial ways. First, case is not significant, `SIZE` is just as good as `size` (but try to be consistent!). Second, writing to a non-existent attribute will *not* give you an error, so proof-read carefully. Third, writing to an attribute often causes some action; e.g the `visible` attribute of controls can be used to hide them. It is best to think of them as a special kind of function call.

Functions which create IupLua objects (i.e. *constructors*) take tables as arguments. Lua allows you to drop the usual parentheses in such a case, but remember that something like `iup.fill{}` is not the same as `iup.fill()`; it is actually short for `iup.fill({})`. A Lua table can contain an array-like part (just items separated by commas) and a map-like part (attribute-value pairs); the convention is to put the array part first, and separate the map part from it with a semicolon. (See Attributes/Guide/IupLua in the Manual for a good discussion.)

Simple Output

Even simple scripts need to give the user some feedback. Otherwise people get anxious and start worrying if their files really have been backed up, for example. This is easy in IUPLua, and takes exactly one line. Note that all IUP scripts must at least have a `require 'iuplua'` statement at the begining:

```
require( "iuplua" )

iup.Message('YourApp','Finished Successfully!')
```

Of course, many operations require confirmation from the user. `iup.Alarm` is designed for this:

```
require( "iuplua" )

b = iup.Alarm("IupAlarm Example", "File not saved! Save it now?" ,"Yes" ,"No" ,"Cancel")

-- Shows a message for each selected button
if b == 1 then
  iup.Message("Save file", "File saved sucessfully - leaving program")
elseif b == 2 then
  iup.Message("Save file", "File not saved - leaving program anyway")
elseif b == 3 then
  iup.Message("Save file", "Operation canceled")
end
```

Like `iup.Message`, the first parameter appears in the title bar of the dialog box, the second parameter appears above the buttons, but `iup.Alarm` allows you to specify a number of buttons. The return code will then tell you which button has been pressed, starting at 1 (which is always the Lua way.)

Simple Input

Asking for a Filename

The most common thing an interactive script will require from a user is a file, or set of files. For simple cases, `iup.GetFile` will do the job:

```
require( "iuplua" )

f, err = iup.GetFile("*.txt")
if err == 1 then
  iup.Message("New file", f)
elseif err == 0 then
  iup.Message("File already exists", f)
elseif err == -1 then
  iup.Message("IupFileDlg", "Operation canceled")
end
```

This will present you with the standard Windows File Open dialog box, and allow you to either choose a filename, or cancel the operation. Notice that this function returns two values, the filename and a code. The code will tell you whether the file does not exist yet (if for instance you typed a new filename into the file dialog box.)

Asking for Multiline Text

The simplest way of getting general text is to use `iup.GetText`:

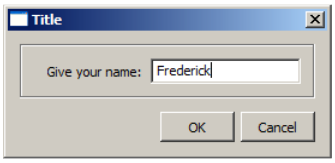
```
require 'iuplua'

res = iup.GetText("Give me your name","")

if res ~= "" then
    iup.Message("Thanks!",res)
end
```

Using this dialog, you can enter as many lines as you like, and press OK.

Asking for a Single String, or Number

A better option for asking for a single string is the very versatile `iup.GetParam`:

```
require( "iuplua" )
require( "iupluacontrols" )

res, name = iup.GetParam("Title", nil,
    "Give your name: %s\n","")

iup.Message("Hello!",name)
```



This has two advantages over plain `GetText`; you can give a prompt line, and you can press Enter after entering text.

The `%s` code requires some explanation. Although you might at first think it is a C-style formating code, as you would use in `string.format`, it actually describes how the value is going to edited; `%s` here merely means that a regular text box is used; if you had used `%m`, then a multiline edit box (like that used by `iup.GetText`) would be used.

If there is a limited set of choices, then the `%l` format is useful:

```
res, prof = iup.GetParam("Title", nil,
    "Give your profession: %l|Teacher|Explorer|Engineer|\n",0)
```

Note the `|item1|item2|...|` list after the `%l` format; these are the choices presented to the user. The initial value you give it, and the value you receive from it, are going to be an index into this list of choices. Somewhat confusingly, they start at 0 (which is not the Lua way!) So in this case, `0` means that 'Teacher' is to be selected,

and if I then selected 'Engineer', the resulting value of `prof` would be 2.

The `%i` code allows you to enter an integer value, with up/down arrows for incrementing/decrementing the value.

```
require( "iuplua" )
require( "iupluacontrols" )

res, age = iup.GetParam("Title", nil,
    "Give your age: %i\n",0)

if res ~= 0 then    -- the user cooperated!
    iup.Message("Really?",age)
end
```

Dialogs

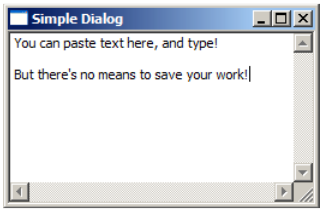
Constructing General Layouts

`GetParam` is a very versatile facility for asking for data, but it is not very interactive. In general, you want to present something back to the user that is more complicated than a simple message. Up to now we have used the predefined dialogs available to IupLua; it is now time to go beyond that and examine custom dialogs. The structure of a simple IupLua program is straightforward:

```
require 'iuplua'

text = iup.multiline{expand = "YES"}

dlg = iup.dialog{text; title="Simple Dialog",size="QUARTERxQUARTER"}

dlg:show()

iup.MainLoop()
```

A multiline edit *control* is created, and put inside a window frame with a given size, which is then made visible with the `show` method. We then enter the main loop of the application with `MainLoop`, which will only finish when the window is closed.

Controls are also windows, but without the frame and decorations of a *top-level* window; they are always meant to be inside some window frame or other *container*. We set the `expand` attribute of `multiline` to force it to use up all the available space in the frame, so that it takes its size from its container. The dialog's `size` attribute is a string of the form "XSIZExYSIZE", where sizes can be expressed as fractions of the desktop window size, in this case a quarter of the width and height. (You can of course also use numerical sizes like "100x301" but these will not always scale well on displays with different resolutions. See Attributes/Common/SIZE in the manual for these units.)



It's good to pause a moment to look at the resulting application in action; it is fully responsive and you can enter text, paste, etc. into the edit control. Common keyboard shortcuts like `Ctrl+V` and `Ctrl+C` work as expected. All this functionality comes with the windowing system you are currently using. On my system, Task Manager shows that this program uses 3.8 Meg of memory, and an instance of Notepad uses 3.3 Meg, which represents all the common code necessary to support a simple GUI application; you are not actually paying much for using IupLua at all. The equivalent C program using the Windows API would be about 150 lines, so the gain in *programmer efficiency* is tremendous!

Of course, there is not much interaction possible with such a simple program. To make a program respond to the user we define *callbacks* which the system calls when some event takes place. For example, we can put a button control in the dialog, and define its `action` callback:

```
require 'iuplua'

btn = iup.button{title = "Click me!"}

function btn:action ()
    iup.Message("Note","I have been clicked!")
    return iup.DEFAULT
end

dlg = iup.dialog{btn; title="Simple Dialog",size="QUARTERxQUARTER"}

dlg:show()

iup.MainLoop()
```

This is perfectly responsive, although not very useful! The button sizes itself to its natural size since `expand` is not set (try setting `expand` to see the button fill the whole window frame.) Callbacks usually return the special value `iup.DEFAULT`, although in IupLua this is not really necessary.

`dialog` takes only one control, so IupLua defines containers in which you can pack as many controls as you like. Here `vbox` is used to pack two buttons into the dialog vertically (To save space I'm leaving out the 'dlg:show...' common code at the bottom)

```
btn1 = iup.button{title = "Click me!"}
btn2 = iup.button{title = "and me!"}

function btn1:action ()
    iup.Message("Note","I have been clicked!")
end

function btn2:action ()
    iup.Message("Note","Me too!")
end

box = iup.vbox {btn1,btn2}

dlg = iup.dialog{box; title="Simple Dialog",size="QUARTERxQUARTER"}
```

This does the job, although the buttons are sized differently according to their contents; this program would not win any design contests! Still, you now have two commands in your application. You can actually get a more pleasing result by using a horizontal packing box (`hbox`) and specifying a non-zero gap between the buttons:

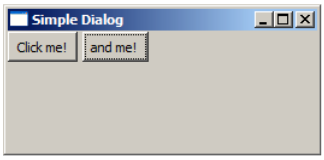
```
box = iup.hbox {btn1,btn2; gap=4}
```

You can nest boxes as much as you like, which is the way to construct more complicated layouts. Here are our horizontal buttons packed vertically with a multiline edit control:

bbox = iup.hbox {btn1,btn2; gap=4} text = iup.multiline{expand = "YES"} vbox = iup.vbox{bbox,text}

dlg = iup.dialog{vbox; title="Simple Dialog",size="QUARTERxQUARTER"}

We have effectively implemented a crude but functional toolbar:



A labeled frame can be put around a control using `iup.frame`:

```
edit1 = iup.multiline{expand="YES",value="Number 1"}
edit2 = iup.multiline{expand="YES",value="Number 2?"}

box = iup.hbox {iup.frame{edit1;Title="First"},iup.frame{edit2;Title="Second"}}
```

A useful way to present various views to a user is to put them in tabs. This places each control in a separate page, accessible through the tabbar at the top. Notice in this example that the titles of the tab pages are actually set as attributes of the *pages* through `tabtitle`. This is not one of the standard IUP controls (see Controls/Additional in the manual) so we also need to bring in the `iupluacontrols` library.

```
require( "iupluacontrols" )

edit1 = iup.multiline{expand="YES",value="Number 1",tabtitle="First"}
edit2 = iup.multiline{expand="YES",value="Number 2?",tabtitle="Second"}

tabs = iup.tabs{edit1,edit2,expand='YES'}
```

Timers and Idle Processing

Sometimes a program needs to wake up and perform some operation, such as a scheduled backup or an autosave operation. IupLua provides *timers* for this purpose. (Note at this point that there is no reason why you can't have `print` in a IupLua application; sometimes there is no better way to track what's going on. But on Windows you do have to run the program using the regular lua.exe, not wlua.exe.)

```
-- timer1.lua

require "iuplua"

timer = iup.timer{time=500}

btn = iup.button {title = "Stop",expand="YES"}

function btn:action ()
    if btn.title == "Stop" then
        timer.run = "NO"
        btn.title = "Start"
    else
        timer.run = "YES"
        btn.title = "Stop"
    end
end

function timer:action_cb()
    print 'timer!'
end

timer.run = "YES"

dlg = iup.dialog{btn; title="Timer!"}
dlg:show()
iup.MainLoop()
```

After a timer has been started by setting its `run` attribute to "YES", it will continue to call `action_cb` using the given time in milliseconds. Notice that it is important to set the timer going only after the callback has been defined. It is perfectly permissable to switch a timer off in the callback, which is how you can perform a single action after waiting for some time.

It is a well-known fact that computers spend most of the time doing very little, waiting for incredibly slow humans to type something new. However, when a computer is actually doing intense processing, users become impatient if not told about progress. If you do your lengthy processing directly, then the windows of the application become unresponsive. The proper way to organize such work is to do it when the system is *idle*.

IupLua provides a gauge control which is intended to show progress; this little program shows that even when the computer is almost completely preoccupied doing work, it is still keeping the user informed and in fact the window remains useable, although a little slow to respond.

```
-- idle1.wlua
require "iuplua"
require "iupluacontrols"

function do_something ()
    for i=1,6e7 do end
end

gauge = iup.gauge{show_text="YES"}

function idle_cb()
    local value = gauge.value
    do_something()
```

```
    value = value + 0.1
    if value > 1.0 then
        value = 0.0
    end
    gauge.value = value
    return iup.DEFAULT
end

dlg = iup.dialog{gauge; title = "IupGauge"}

iup.SetIdle(idle_cb)

dlg:showxy(iup.CENTER, iup.CENTER)

iup.MainLoop()
```

Lists

It is easy to display a list of values in IupLua. The values can be directly specified in the `iup.list` constructor, like so:

```
-- list1.wlua
require "iuplua"

list = iup.list {"Horses","Dogs","Pigs","Humans"; expand="YES"}

dlg = iup.dialog{list; title="Lists"}
dlg:show()
iup.MainLoop()
```

(Remember that the single argument to these constructors is just a Lua table, which you can construct in any way you choose, say by reading the values from a file.)

Tracking selection changes is straightforward:

```
function list:action(t,i,v)
    print(t,i,v)
end
```

Now, as I move the selection through the list, from the start to the finish, the output is:

```
Horses  1       1
Horses  1       0
Dogs    2       1
Dogs    2       0
Pigs    3       1
Pigs    3       0
Humans  4       1
```

So `v` is 1 if we are selecting an item, 0 if we are deselecting it; `i` is the one-based index in the list, and `t` is the actual string value. If you want to associate some other data with each value, then all you need to do is keep a table of that data and look it up using the index `i`.

To register a double-click is a little more involved. There is (as far as I can tell) no way to detect whether a double-click has happened in the `action` callback. So we track the selection manually; if two selection events for a given item happen consecutively, then that is understood to be a double-click. It ain't pretty, but it works (except perhaps for the valid case of a person wanting to double-click the same item repeatedly):

```
local lastidx,doubleclick

function on_double_click (t,i)
    print(t,i)
end

function list:action(t, i, v)
    if v ~= 0 then
        if lastidx == i and doubleclick ~= i then
            on_double_click(t,i)
            doubleclick = i
        end
        lastidx = i
    end
end
```

Once a list has been created, how does one change the contents? The answer is that the list object *behaves* like an array. For example, to fill a list with all the entries in a directory, I can use this function:

```
function fill (path)
    local i = 1
    for f in lfs.dir(path) do
        list[i] = f
        i = i + 1
    end
    list[i] = nil
end
```

Note that this does not mean that a list object *is* a table. In particular, you have to explicitly set the end of the list of elements by setting a nil value just after the end.

Trees

The most flexible way to present a hierarchy of information is a tree. A tree has a single *root*, and several *branches*. Each of these *branches* may have *leaves*, and other branches. All of these are called *nodes*. Thinking of a family tree, a node may have *child* nodes, which all share the same *parent* node.

A good example of this in everyday computer experience is a filesystem, where the leaves are files and the branches are directories. Lua tables naturally express these kind of *nested* structures easily, and in fact it is easy to present a Lua table as a tree, where array items are leaves, and the branches are named with the special field `branchname`:

```
require 'iuplua'
require 'iupluacontrols'

tree = iup.tree{}
tree.addexpanded = "NO"
```

```
list = {
  {
    "Horse",
    "Whale";
    branchname = "Mammals"
  },
  {
    "Shrimp",
    "Lobster";
    branchname = "Crustaceans"
  },
  {
    branchname = "Birds"
  },
  branchname = "Animals"
}

iup.TreeSetValue(tree, list)

f = iup.dialog{tree; title = "Tree Test"}

f:show()

iup.MainLoop()
```

This example begins with the branches 'collapsed', and you will have to explicitly expand them with a mouse click. By default, trees are presented in their fully expanded form; try taking out the fourth line that sets the `addexpanded` attribute of the tree object. Note that branches can be empty!

Tree operations are naturally more complicated than list operations, but there is a callback which happens when a node is selected or unselected. Add this to the example:

```
function tree:selection_cb (id,iselect)
    print(id,iselect,tree.name)
end
```

You will see that `iselect` is 0 for the unselection operation, and 1 for selection; id is a tree node index. These indices are always in order of appearance in a tree, starting at 0 for the root node. The `name` attribute of the tree object is the text of the currently selected node.

A pair of useful callbacks are `branchopencb` *and* `branchclosecb`. If you were displaying a potentially very large tree (like your computer's filesystem) then it would be inefficient to create the whole tree at once, especially considering that you would normally be only interested in a small part of that tree. Trapping `branchopencb` *allows you to add child nodes to your selected node before it is expanded.* `executeleafcb` is called when you double-click on a leaf, as if you were running a program in a file explorer.

In itself, the id is not particularly useful. The id order is always the same in the tree, so as nodes get added and removed, the id of a particular node will change. Generally, there is going to be some deeper data associated with a node. On a filesystem, a node represents a full path to a file or directory, or there may be an ip address associated with a computer name. IUP provides you with a way to associate arbitrary data with nodes even if the id changes. But to use this you will have to understand how to build up a tree from scratch - `TreeSetValue` is very convenient, but won't help you if you have to add nodes later. Replace the definition of `list` and the call to `TreeSetValue` with this code:

```
tree.name = "Animals"
tree.addbranch = "Birds"
tree.addbranch = "Crustaceans"
tree.addbranch = "Mammals"
```

You will get the top level branches of the tree; notice that they are specified in reverse order, since nodes are always added to the top. Also note the curious way in which the `addbranch` attribute is used. For a start, it is *write-only*, and the effect of setting a value to it is to add a new branch to the currently selected node. By default, this starts out as the root (which is set using the `name` attribute) The id of the root is always 0; when we add "Birds", the new branch has id 1, again when we add "Crustaceans" the new branch also has id 1 - by which time "Birds" has moved to id 2, further down the tree.

To add leaves, a similar process:

```
tree.name = "Animals"
tree.addbranch = "Mammals"
tree.addleaf1 = "Whale"
```

The `addleaf` attribute works like `addbranch`, and both of them can take an extra parameter, which is the id of the node to add to. In this case, "Whale" is a child leaf of the "Mammals" branch, which has id 1 *at this stage*. This new leaf gets an id of 2, which is one more than the parent. So this gives us a way to build up arbitrary trees, knowing the id at each point. IUP provides a function `TreeSetTableId` which can associate a Lua table with a node id. We can choose to put a string value inside this table, but it really can contain anything. Here is the first example, using some helper functions to simplify matters:

```
-- testtree2.lua

require 'iuplua'
require 'iupluacontrols'
tree = iup.tree{}

function assoc (idx,value)
    iup.TreeSetTableId(tree,idx,{value})
end

function addbranch(self,label,value)
    self.addbranch = label
    assoc(1,value or label)
end

function addleaf(self,label,value)
    self.addleaf1 = label
    assoc(2,value or label)
end

tree.name = "Animals"
addbranch(tree,"Birds")
addbranch(tree,"Crustaceans")
addleaf(tree,"Shrimp")
addleaf(tree,"Lobster")
addbranch(tree,"Mammals")
addleaf(tree,"Horse")
addleaf(tree,"Whale")

function dump (tp,id)
```

```
    local t = iup.TreeGetTable(tree,id)
    -- our string data is always the first element of this table
    print(tp,id,t and t[1])
end

function tree:branchopen_cb(id)
    dump('open',id)
end

function tree:selection_cb (id,iselect)
    if iselect == 1 then dump('select',id) end
end

f = iup.dialog{tree; title = "Tree Test"}

f:show()

iup.MainLoop()
```

There is a corresponding function `TreeGetTable` which accesses the table associated with the node id. There is also a function `TreeGetTableId` which will return the id, given the *unique* table associated with it. You can use this to programmatically select a tree node given its data by setting the `value` attribute to the returned id.

Now let's do something interesting with a tree control, a simple file browser. It is straightforward to get the files and directories contained within a directory:

```
require 'lfs'

local append = table.insert

function get_dir (path)
    local files = {}
    local dirs = {}
    for f in lfs.dir(path) do
        if f ~= '.' and f ~= '..' then
            if lfs.attributes(path..'/'..f,'mode') == 'file' then
                append(files,f)
            else
                append(dirs,f)
            end
        end
    end
    return files,dirs
end
```

We ignore '.' and '..' (the current and parent directory respectively) and check the mode to see if we have file or a directory; this requires the full path to be passed to `attributes`. This function returns two separate tables containing the *names* of the files and directories.

It is useful to define two helper functions for setting and getting data to be associated with the tree nodes:

```
tree = iup.tree {}

function set (id,value,attrib)
    iup.TreeSetTableId(tree,id,{value,attrib})
end

function get(id)
    return iup.TreeGetTable(tree,id)
end
```

Filling a tree with the contents of a directory is straightforward. We want the directories before the files, so we put them in last; nodes must be added in reverse order! The id of the new nodes will always be `id+1` where `id` is going to be the directory which we are filling. The fullpath plus a field indicating whether we are a directory is associated with each new item:

```
function fill (path,id)
    local files,dirs = get_dir(path)
    id = id + 1
    local state = "STATE"..id
    for i = #files,1,-1 do -- put the files in reverse order!
        tree.addleaf = files[i]
        set(id,path..'/'..files[i])
    end
    for i = #dirs,1,-1 do -- ditto for directories!
        tree.addbranch = dirs[i]
        set(id,path..'/'..dirs[i],'dir')
        tree[state] = "COLLAPSED"
    end
end
```

By default, the directory branches will be created in their expanded form, so we use the `STATE` attribute to force them into their collapsed state. Normally you would say this in Lua like so `state2 = "COLLAPSED"` but here we build up the appropriate attribute string with the given id and use array indexing to set the tree attribute.

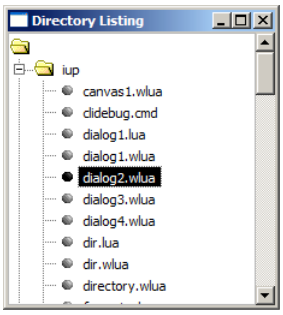
Just calling `fill('.',0)` and putting the tree into a dialog as usual will give you a directory listing of the current directory! But it would be cool if expanding a directory node would automatically fill that node; it would obviously be wasteful to fill the whole tree at startup, since your filesystem contains thousands of files. The `branchopencb` *callback is called when a user tries to expand a directory. We use this to fill the directory with its contents, but only on the _first* time that we expand this node:

```
function tree:branchopen_cb(id)
    tree.value = id
    local t = get(id)
    if t[2] == 'dir' then
        fill(t[1],id)
        set(id,t[1],'xdir')
    end
end
```

This is why directories need to be *specially marked*, so we can tell later whether we have actually generated the contents of that directory!

The first statement of this function makes the node we are opening to be the selected node of the tree. (Although we are passed the correct id of the node, it seems to be necessary to perform this step to make things work correctly.)

See `directory.wlua` in the examples folder.



Menus

Any application that can perform a number of operations needs a menu. These are not difficult to create in Iuplua, although it can be a little tedious to set up. The basic idea is this: create some *items*, make a menu out of these items, and set the `menu` attribute of the dialog. The items have an associated `action` callback, which actually performs the operation.

```
-- simple-menu.wlua

require( "iuplua" )

-- Creates a text, sets its value and turns on text readonly mode
text = iup.text {readonly = "YES", value = "Show or hide this text"}

item_show = iup.item {title = "Show"}
item_hide = iup.item {title = "Hide"}
item_exit = iup.item {title = "Exit"}

function item_show:action()
  text.visible = "YES"
  return iup.DEFAULT
end

function item_hide:action()
  text.visible = "NO"
  return iup.DEFAULT
end

function item_exit:action()
  return iup.CLOSE
end

menu = iup.menu {item_show,item_hide,item_exit}

-- Creates dialog with a text, sets its title and associates a menu to it
dlg = iup.dialog{text; title="Menu Example", menu=menu}

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

iup.MainLoop()
```

A menu may contain items and *submenus*. This example shows a small function which makes creating arbitrarily complicated menus easier:

```
-- menu.wlua

require( "iuplua" )

function default ()
    iup.Message ("Warning", "Only Exit performs an operation")
    return iup.DEFAULT
end

function do_close ()
    return iup.CLOSE
end

mmenu = {
    "File",{
        "New",default,
        "Open",default,
        "Close",default,
        "-",nil,
        "Exit",do_close,
    },
    "Edit",{
        "Copy",default,
        "Paste",default,
        "-",nil,
        "Format",{
            "DOS",default,
            "UNIX",default
        }
    }
}

function create_menu(templ)
    local items = {}
    for i = 1,#templ,2 do
        local label = templ[i]
        local data = templ[i+1]
        if type(data) == 'function' then
            item = iup.item{title = label}
            item.action = data
        elseif type(data) == 'nil' then
            item = iup.separator{}
        else
            item = iup.submenu {create_menu(data); title = label}
```

```
        end
        table.insert(items,item)
    end
    return iup.menu(items)
end

-- Creates a text, sets its value and turns on text readonly mode
text = iup.text {value = "Some text", expand = "YES"}

-- Creates dialog with a text, sets its title and associates a menu to it
dlg = iup.dialog {text; title = "Creating Menus With a Table",
      menu = create_menu(mmenu), size = "QUARTERxEIGHTH"}

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

iup.MainLoop()
```

The function `create_menu` does all the work; we provide it with a Lua table containing pairs of values; the first value of a pair is always a string, and will be the label. The second value can either be a function, in which case it represents an item to be associated with a callback, or `nil`, which means that it's a separator, or otherwise must be a table, which represents a submenu. It is a nice example of how recursion can naturally handle nested structures like menus, and how Lua's flexible table definitions can make specifying such structures easy. This useful function is available in the `iupx` utility library as `iupx.menu`.

Plotting Data

Many kinds of numerical data are best seen as X-Y plots. `iup.pplot` is a control which can show several kinds of plots; you can have lines between points, show them as markers, or both together. Several series (or *datasets*) can be shown on a single plot, and a simple legend can be shown. The plot will automatically scale to view all datasets, but the default minimum and maximum x and y values can be changed. It is even possible to select points and edit them on the plot.

A simple plot is straightforward:

```
-- pplot1.wlua
require( "iuplua" )
require( "iupluacontrols" )
require( "iuplua_pplot51"  )

plot = iup.pplot{TITLE = "A simple XY Plot",
                    MARGINBOTTOM="35",
                    MARGINLEFT="35",
                    AXS_XLABEL="X",
                    AXS_YLABEL="Y"
                    }

iup.PPlotBegin(plot,0)
iup.PPlotAdd(plot,0,0)
iup.PPlotAdd(plot,5,5)
iup.PPlotAdd(plot,10,7)
iup.PPlotEnd(plot)

dlg = iup.dialog{plot; title="Plot Example",size="QUARTERxQUARTER"}

dlg:show()

iup.MainLoop()
```

Creating a dataset involves calling `PPlotBegin`, a number of calls to `PPlotAdd` to add data points, and finally a call to `PPlotEnd`. You can create multiple datasets (or series) using multiple begin/end calls, and can of course use loops to add points:

```
iup.PPlotBegin(plot,0)
for x = -2,2,0.01 do
    iup.PPlotAdd(plot,x,math.sin(x))
end
iup.PPlotEnd(plot)

iup.PPlotBegin(plot,0)
for x = -2,2,0.01 do
    iup.PPlotAdd(plot,x,math.cos(x))
end
iup.PPlotEnd(plot)

plot.DS_LINEWIDTH = 3
```

A limitation of the `pplot` library is that it does not choose appropriate sizes for the plot margins. So I've had to set the bottom and left margins (in pixels) to properly accomodate the axes and their titles. As with all IupLua attributes, you can choose to make them uppercase if you like; a full list is found in the manual under Controls/Additional/IupPPlot. Some of these attributes refer to the plot as a whole, some to the *current dataset*. For instance, setting `GRID` to "YES" will draw gridlines for both axes, but if we set `DS_LINEWIDTH` to 3 after the construction of the cosine dataset, then only that line is affected.

Some attributes affect others. `DS_MODE` is used to specify how to draw the dataset; it can be "LINE", "BAR", (for a bar chart) "MARK" (just for marks) or "MARKLINE" (for lines and marks). But it has to be set before any of the other `DS_` attributes like `DS_MARKSIZE`, etc. In another case, you will often find it useful to set an explicit minimum y value by setting `AXS_YMIN`. But it will only take effect if `AXIS_YAUTOMIN` has been set to "NO" to disable auto scaling.

As with menus, making a Lua-friendly wrapper around an API is not difficult and can be very labour-saving. It would be clearer if we could work with the plot object in a more object-oriented way:

```
    plot:Begin()
    for i = 1,#xvalues do
            plot:Add(xvalues[i],yvalues[i])
    end
    plot:End()
```

And for the common case where you have arrays of values, it would be convenient to be able to say:

```
plot:AddSeries({{0,1.5},{5,4.5},{10,7.6}},{DS_MODE="MARK"})
```

Here is a function which wraps the `PPlot` API:

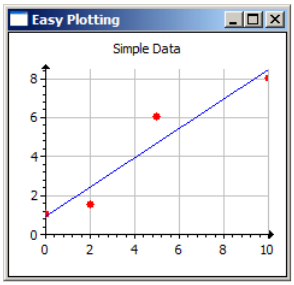
```
function create_pplot (tbl)
    -- don't need to remember this anymore!
    require( "iuplua_pplot51"  )
```

```
    -- the defaults for these values are too small, at least on my system!
    if not tbl.MARGINLEFT then tbl.MARGINLEFT = 30 end
    if not tbl.MARGINBOTTOM then tbl.MARGINBOTTOM = 35 end

    -- if we explicitly supply ranges, then auto must be switched off for that direction.
    if tbl.AXS_YMIN then tbl.AXS_YAUTOMIN = "NO" end
    if tbl.AXS_YMAX then tbl.AXS_YAUTOMAX = "NO" end
    if tbl.AXS_XMIN then tbl.AXS_XAUTOMIN = "NO" end
    if tbl.AXS_XMAX then tbl.AXS_XAUTOMAX = "NO" end

    local plot = iup.pplot(tbl)
    plot.End = iup.PPlotEnd
    plot.Add = iup.PPlotAdd
    function plot.Begin ()
        return iup.PPlotBegin(plot,0)
    end

    function plot:AddSeries(xvalues,yvalues,options)
        plot:Begin()
        -- is xvalues a table of (x,y) pairs?
        if type(xvalues[1]) == "table" then
            -- because there's only one data table, the next must be options
            options = yvalues
            for i,v in ipairs(xvalues) do
                plot:Add(v[1],v[2])
            end
        else
            for i = 1,#xvalues do
                plot:Add(xvalues[i],yvalues[i])
            end
        end
        plot:End()
        -- set any series-specific plot attributes
        if options then
            -- mode must be set before any other attributes!
            if options.DS_MODE then
                plot.DS_MODE = options.DS_MODE
                options.DS_MODE = nil
            end
            for k,v in pairs(options) do
                plot[k] = v
            end
        end
    end

    function plot:Redraw()
        plot.REDRAW='YES'
    end
    return plot
end
```

This function creates a `PPlot` object as usual, but supplies some more sensible defaults for the margins, makes setting things like `AXS_XMAX` also set `AXS_XAUTOMAX`, and adds some new methods to the object. Of these, `AddSeries` is the interesting one. It allows you to specify the data in two forms; either as two arrays of x and y values, or as a single array of x-y pairs. It also allows optionally setting `DS_` attributes, taking care to set the plot mode before any other attributes. In this way, the actual details can be hidden away from the programmer, who has then less things to worry about.

Given this function, we can write a little program which plots some points and draws the linear least-squares fit between them:

```
-- simple-pplot.wlua

local xx = {0,2,5,10}
local yy = {1,1.5,6,8}

function least_squares (xx,yy)
    local xsum = 0.0
    local ysum = 0.0
    local xxsum = 0.0
    local yysum = 0.0
    local xysum = 0.0
    local n = #xx
    for i = 1,n do
        local x,y = xx[i], yy[i]
        xsum = xsum + x
        ysum = ysum + y
        xxsum = xxsum + x*x
        yysum = yysum + y*y
        xysum = xysum + x*y
    end
    local m = (xsum*ysum/n - xysum )/(xsum*xsum/n - xxsum)
    local c = (ysum - m*xsum)/n
    return m,c
end

local m,c = least_squares(xx,yy)

function eval (x) return m*x + c end

local plot = create_pplot {TITLE = "Simple Data",AXS_YMIN=0,GRID="YES"}

-- the original data
plot:AddSeries(xx,yy,{DS_MODE="MARK",DS_MARKSTYLE="CIRCLE"})

-- the least squares fit
local xmin,xmax = xx[1],xx[#xx]
plot:AddSeries({xmin,xmax},{eval(xmin),eval(xmax)})
```



`create_plot` is so useful that I've packaged it as part of the `iupx` library as `iupx.pplot`. A new pseudo-attribute has been introduced, `AXS_BOUNDS`, which is a table of four values `{xmin,ymin,xmax,ymax}`. This example shows that very different ranges can happily exist on the same plot:

```
-- pplot5.wlua

require "iupx"

plot = iupx.pplot {TITLE = "Simple Data", AXS_BOUNDS={0,0,100,100}}

plot:AddSeries ({{0,0},{10,10},{20,30},{30,45}})
plot:AddSeries ({{40,40},{50,55},{60,60},{70,65}})

iupx.show_dialog{plot; title="Easy Plotting",size="QUARTERxQUARTER"}
```

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(lua_State *L, Ihandle *ih);
```

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

```
Ihandle* ih = *(Ihandle**)lua_touserdata(L, pos);
```

or using parameter checking:

```
Ihandle* iuplua_checkihandle(lua_State *L, int pos);
```

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, classname)
```

where "classname" is the string returned in IupGetClassName.

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 5 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 of 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 metable 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
```

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.

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 function. Some parameters processed by the driver can be removed so the address is necessary. They can be NULL. (Since 2.7)

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. It will also automatically destroy all dialogs and all elements that have names.

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.

This function is usually called by the application. 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 is also valid for all the additional controls when IUP is dynamically loaded. To call **IupClose** in this way you must call **iuplua_close**.

This enable you to dynamically load IUP using Lua 5 "require".

Parameters/Return

```
int iuplua_open(lua_State *L); [in C]
[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

IupVersion

Returns a string with the IUP version number.

Parameters/Return

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

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

```
iup.VersionNumber() -> (version: number) [in Lua]
```

Returns: the version number including the bug fix. The defines only includes the major and minor numbers. For example: "2.7.1".

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         "2.7"
IUP_VERSION_DATE    "2008/10/14"
IUP_VERSION_NUMBER  207000

[in Lua]
iup._NAME
iup._DESCRIPTION
iup._COPYRIGHT
iup._VERSION
iup._VERSION_DATE
iup._VERSION_NUMBER
```

IupLoad and IupLoadBuffer

Compiles a LED specification.

Parameters/Return

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

char *IupLoadBuffer(const char *buffer); [in C] (since 3.0)
iup.LoadBuffer(buffer: string) -> error: string [in Lua]
```

filename: name of the file containing the LED specification.
buffer: string with the LED specification.

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]
```

Returns: the language.

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 aplications 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.

Indigo Magic look in Sgi

To turn on the Indigo Magic look for an application, simply set the applications sgiMode resource to TRUE. Typically, you should add this line to the "/usr/lib/X11/app-defaults" file for your application:

appName*sgiMode: TRUE

where appName is the name of your application.

Win32 System Driver

This driver was designed for the modern Microsoft Windows in 32 bits or 64 bits (2000/XP/2003/Vista/7).

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.

Tips

Visual C++ 6

Since 3.0 Visual C++6 is not supported, although we still provide pre-compiled binaries. To compile the IUP 3 code with VC6 you will need to download a new Platform SDK, because the one included in the compiler is too old. But it cannot be a too new one also, because the compiler will report errors in the newest headers. So we recommend you to upgrade your compiler. Visual C++ Express Edition is a free compiler that has everything VC6 had and more.

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 **IupOleControl**.

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/Vista/7 Visual Styles

Windows 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 Visual Styles won't work.

If your Windows is using the Windows Classic theme or any other style, IUP controls appearance will follow the system appearance if you are using a Manifest. If not, it will always look like Windows Classic.

Black Canvas

The IupGLCanvas does not work when inside an IupFrame, the result is a black canvas with no drawing. (Only in IUP 2.x)

Help in CHM format fail to open

When you download a CHM file from the Internet Windows blocks your access to the file. You must unblock it manually. Right click the file in Explorer and select "Unblock" at the bottom of the dialog.

GTK System Driver (since 3.0)

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

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

The oldest GTK version that can be used is 2.4, oldest versions will not compile. But using versions older than 2.12 several features will not work. Critical features need at least version 2.8.

Currently it is not available for IRIX, AIX and SunOS. But is it available for SunOS10 and it is not available for Linux24.

Tips

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

GTK uses UTF-8 as its charset for all displayed text, so IUP will automatically convert all strings to (SetAttribute) and from (GetAttribute) UTF-8. To use UTF-8 strings set the global attribute UTF8AUTOCONVERT to NO.

When using DLLs in Windows, the "iup.dll" uses the Win32 driver. So applications that dynamically load IUP will always use the "iup.dll", for example Lua using require. To use the IUP GTK dll in Windows you must rename the "iupgtk.dll" to "iup.dll", so the GTK driver will be loaded instead of the Win32 driver.

GTK is in fact composed of several libraries. The GTK package contains the GDK library and depends on the ATK, Cairo, Glib and Pango libraries.

Attributes

Attributes are used to change or consult 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 a way to send and obtain information to and from elements. They are used by the application to communicate with the user interface system, on the other hand callbacks are used by the application to receive notifications from the system that the user or the system itself has interacted with the user interface of the application.

There are two functions to change them:

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

With **IupSetAttribute** you can also store application pointers that can be strings or not. This can be very useful, for instance, used 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 and the default value will be used if there is one defined. 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.SetClassAttribute(name, value)==store then
    ih.SetHashTableAttribute(name, value)
  endif
  if (ih.IsInheritable(name))
    -- NotifyChildren
    for each child of ih do
       if not child.GetHashTableAttribute(name) then
        child.SetClassAttribute(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.GetClassAttribute(name)
  if not value then
    value = ih.GetHashTableAttribute(name)
  endif
  if not value and ih.isInheritable(name) then
    parent = ih.parent
    while (parent and not value)
      value = parent.GetHashTableAttribute(name)
      parent = parent.parent
    endwhile
  endif
  if not value then
    value = ih.GetDefaultAttribute(name)
  endif
}
```

Notice that the parent recursion is done only for the parent hash table, the parent class implementation is not consulted.

The class implementation of the control can update or retrieve a value even if the control is not mapped. When the control is not mapped and its implementation can not process the attribute, then the attribute is simply stored in the hash table. After the element is mapped its attributes are re-processed to be updated in the native system and they can be removed from the hash table at that time.

All this flexibility turns the attribute system very complex with several nuances. If the attribute is checked before the control is mapped and just after, its value can be completely different. Depending on how the attribute is stored its inheritance can be completly ignored.

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. In some cases, common attributes behave differently in different elements, but in such cases, there are comments in the documentation of the element explaining the different behavior.

In LED there is no quotation marks for attributes, names or values. In Lua attribute names can be lower case.

Inheritance

Elements included in other elements can inherit their attributes. There is an **inheritance** mechanism inside a given child tree.

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. As general rules the following attributes are **NON** inheritable always:

- Essencial attributes like VALUE, TITLE, SIZE, RASTERSIZE, X and Y
- Id numbered attributes (like "1" or "MARK1:1")
- Handle names (like "CURSOR", "IMAGE" and "MENU")
- Pointers that are not strings (like WID)
- Read-only or write-only attributes
- Internal attributes that starts with "_IUP"

Inheritable attributes are stored in the hash table so the IupGet/SetAttribute logic can work, even if the element implementation store it internally. But when you change an attribute to NULL, then its value is removed from the hash table and the default value if any is passed to the native system. Non inheritable attributes never have default values.

When consulted the attribute is first checked at the element specific implementation. 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's hash table and so forth, until it reaches the root child (usually 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).

When changed the attribute change is propagated to all children except for those who the attribute is already defined.

Noticed that when changed the inheritance check is done for the current control. Since Vbox, Hbox, and other containers have only a few registered attributes, by default an unknown attribute is treated as inheritable. For example, the IMAGE attribute of a Label is not inheritable, so when checked at the Label it will return NULL if not defined, the Label parent tree will not be consulted. But if you change the IMAGE attribute at a Vbox that contains several Labels, the Vbox will not be affected but all child Labels will.

Availability

Although attributes can be changed and retrieved at any time there are exceptions and some rules that must be followed according to the documentation of the attribute:

- **read only**: the attribute can not be changed. Ignored when set.
- **write only**: the attribute can not be retrieved. Normally used for action attributes. Returns NULL or some value set before the element was mapped.
- **creation only**: it will be used only when the element is mapped on the native system. So set it before the element is mapped. Ignored when set after the element is mapped.

IupLua

Each interface element is created as a Lua table, and its attributes are fields in this table**.** Some of these attributes are directly transferred to IUP, so that any changes made to them immediately reflect on the screen. However, not all attributes are transferred to IUP.

Control attributes, such as handle, which stores the handle of the IUP element, and parent, which stores the object immediately above in the class hierarchy, are not transfered. Attributes that receive strings or numbers as values are immediately transferred to IUP. Other values (such as functions or objects) are stored in IupLua and might receive special treatment.

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

```
myButton = 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 paremeters that are necessary for the element to be created in C.

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

```
mybox = 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 but the string is internally duplicated.

Parameters/Return

```
void IupStoreAttribute(Ihandle *ih, const char *name, const 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 is duplicated before it is stored in the hash table, if that happen. Otherwise behaves just like **IupSetAttribute**.

You can NOT use this functions for pointers that are not strings.

See Also

IupGetAttribute, IupSetAttribute

IupSetAttribute

Defines an attribute for an interface element. See also the Attributes Guide section.

Parameters/Return

```
void IupSetAttribute(Ihandle *ih, const char *name, const 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 default value will be used.

Notes

See the Attributes Guide for more details.

Storage and Inheritance

When an attribute is set it is first processed by the element class implementation, if allowed or ignored by the class implementation then it is stored in the internal hash table. If it is an inheritable attribute then the attribute is propagated to the element children class implementation, their hash table remains untouched.

If the value is NULL, the attribute will be removed from the hash table and the default value will be used during the element class implementation.

Since IUP 3.0, if the attribute is marked non inheritable at a child then its propagation can be ignored by the child. This depends on the implementation of the child class. When that happen set the attribute directly at the child.

User Data

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.

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

Examples

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)
```

Defines a radio's initial value.

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

Other usages:

```
1. IupSetAttribute(text, "VALUE", "Hello!");

2. IupSetAttribute(indicator, "VALUE", "ON");

3. struct
   {
     int x;
     int y;
   } myData;
   IupSetAttribute(text, "myData", (char*)&myData);  // correct, BUT myData should be a global variable.
```

See Also

IupGetAttribute, IupSetAttributes, IupGetAttributes, IupStoreAttribute

IupSetfAttribute

Defines an attribute for an interface element.

Parameters/Return

```
void IupSetfAttribute(Ihandle *ih, const char *name, const 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 **sprintf** 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.

Internally will call **IupStoreAttribute**.

See Also

IupGetAttribute, IupSetAttribute, IupSetAttributes, IupGetAttributes, IupStoreAttribute

IupSetAttributes

Defines a set of attributes for an interface element.

Parameters/Return

```
Ihandle *IupSetAttributes(Ihandle *ih, const 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.

Returns: **ih** if all attributes were defined, or NULL (nil in Lua) otherwise.

Notes

Internally will call **IupStoreAttribute**.

Examples

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

```
dialog = IupSetAttributes(IupDialog(
    IupSetAttributes(IupHBox(
       canvas = IupSetAttributes(IupCanvas(NULL), "BORDER=NO, RASTERSIZE=100x100"),
       NULL), "MARGIN=10x10"),
    "TITLE=Test");
```

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

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

See Also

IupGetAttribute, IupSetAttribute, IupGetAttributes, IupStoreAttribute, IupSetAtt

IupSetAtt

Defines a set of attributes for an interface element and optionally sets its handle.

Parameters/Return

```
Ihandle* IupSetAtt(const char* handle_name, Ihandle* ih, const char* name, ...); [in C]
```

handle_name: optional handle name. **IupSetHandle** will be called internally. can be NULL**.**
ih: Identifier of the interface element.
name: name of the first attribute.
...: after **name** a **value** must be set, then a sequence of name and value pairs can follow until a NULL name is found. It must be a constant string because **IupSetAttribute** will be used internally.

Returns: **ih**

Examples

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

```
dialog = IupSetAtt("MainDialog", IupDialog(
    IupSetAtt(NULL, IupHBox(
       IupSetAtt("MainCanvas", IupCanvas(NULL), "BORDER", "NO", "RASTERSIZE", "100x100", NULL),
       NULL), "MARGIN", "10x10", NULL),
    "TITLE", "Test", NULL);
```

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

```
Ihandle *list = IupList(NULL);
IupSetAtt(NULL, list, "VALUE", "3", "1", "Brazil", "2", "USA", "3", "Japan", "4", "France", NULL);
```

See Also

IupGetAttribute, IupSetAttribute, IupGetAttributes, IupStoreAttribute, IupSetAttributes

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, const 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.

Notes

This work is automatically done in Lua when an attribute that is an element name is set to an element handle. In other words, in Lua you can set a string or a handle as the attribute value, when a handle is used a name is automatically created just as the IupSetAttributeHandle.

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, const 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. See also the Attributes Guide section.

Parameters/Return

```
char *IupGetAttribute(Ihandle *ih, const 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.

Returns: the attribute value or NULL (nil in Lua) if the attribute is not defined or does not exist.

Notes

See the Attributes Guide for more details.

This function return value is not necessarily the same pointer used by the application to define the attribute value. The pointers of internal IUP attributes returned by **IupGetAttribute** must **never** be freed or changed.

Storage and Inheritance

When the attribute is retrieved it is first checked at the element class implementation. If not defined then it checks in the hash table. If not defined, the attribute will be inherited from its parent hash table 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). Inheritance can be blocked by the element class implementation.

IupLua

In IupLua, only known internal pointer attributes are returned as user data or as an ihandle, all other attributes are returned as strings. To access attribute data always as user data use iup.GetAttributeData:

```
iup.GetAttributeData(ih: ihandle) -> value: userdata [in Lua]
```

Examples

Browse for Example Files

See Also

IupSetAttribute, IupGetInt, IupGetFloat, IupSetAttributes, IupGetHandle.

IupGetAllAttributes (Since 3.0)

Returns the names of all attributes of an element that are defined in its intenal hash table only. The internal attributes are also returned (attributes prefixed with "_IUP").

Parameters/Return

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

ih: identifier of the interface element.
names: table receiving the names. Only the list of names need to be allocated. Each name will point to an internal string.
max_n: maximum number of names the table can receive.

Returns: the number of names loaded to the table. If names==NULL or max_n==0 then returns the actual number of names.

See Also

IupGetAttribute, IupSetAttribute, IupSetAttributes, IupStoreAttribute

IupGetAttributes

Returns all attributes of a given element that are in the internal hash table. 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 applciation must ensure that there is no pointer attributes set for that element, although some known pointers are handled.

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

Parameters/Return

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

ih: Identifier of the interface element.

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, const char *name) [in C]
[There is no equivalent in Lua]
```

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

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

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 but the string is internally duplicated.

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 betwwen **IupSetGlobal** and **IupStoreGlobal** is the same of **IupSetAttribute** and **IupStoreAttribute**. In the Store functions the string value is duplicated internally.

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 handle to NULL.

See Also

IupStoreAttribute, 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.

Returns: the attribute 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.

Value

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

Default: "YES"

Notes

An interface element is only active if its native parent is also active.

ACTIVE can also be set for controls that do not have user interaction because they may have a visual feedback to indicate the inactive state.

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.

Affects

All controls that have visual representation.

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. On some controls if not defined will inherit the background of the native parent.

Affects

All, but with restrictions. Several controls have transparent parts that are not affected by the BGCOLOR.

See also the screenshots of the sample.c results with normal background, changing the dialog BACKGROUND, the dialog BGCOLOR and the children BGCOLOR.

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 (since 3.0)

Character font of the text shown in the element. See FONT definition up to IUP 2.x. Although it is an inheritable attribute, it is defined only on elements that have a native representation.

Value

Font description containing typeface, style and size. Default: the global attribute DEFAULTFONT.

The common format definition is similar to the the Pango library Font Description, used by GTK+2. It is defined as having 3 parts: "<font family>, <font styles> <font size>".

Font family can be a list of fonts face names, but this is only accepted in GTK. So the font family can be reduced to font face.

The supported font style is a combination of: **Bold**, **Italic**, **Underline** and **Strikeout**. The Pango format include many other definitions not supported by the common format, they are supported only by the GTK driver. Unsupported values are simply ignored.

Font size is in points (1/72 inch) or in pixels (using negative values).

The old Font Names are still supported. The old Windows FONT format is still supported in the Windows driver.

Returned values will be the same value when changing the attribute, except for the old font names that will be converted to the new common format definition.

Windows

The DEFAULTFONT is retrieved from the System Settings (see bellow), if this failed then "Tahoma, 10" is assumed.

The native handle can be obtained using the "**HFONT**" attribute.

In "Control Panel", open the "Display Properties" then click on "Advanced" and select "Message Box" and change its Font to affect the default font for applications. In Vista go to "Window Color and Appearance", then "Open Classic Appearance", then Advanced.

Motif

The DEFAULTFONT is retrieved from the user resource file (see bellow), if failed then "Fixed, 11" is assumed.

The X-Windows Logical Font Description format (XLFD) is also supported.

The native handle can be obtained using the "**XMFONTLIST**" and "**XFONTSTRUCT**" attributes. The selected X Logical Font Description string can be obtained from the attribute "**XLFD**".

You can use the **xfontsel** program to obtain a string in the X-Windows Logical Font Description format (XLFD). Noticed that the first size entry of the X-Windows font string format (**pxlsz**) is in pixels and the next one (**ptSz**) is in deci-points (multiply the value in points by 10).

Be aware that the resource files ".Xdefaults" and "Iup" in the user home folder can affect the default font and many others visual appearance features in Motif.

GTK

The DEFAULTFONT is retrieved from the style defined by GNOME (see bellow), if failed "Sans, 10" is assumed.

The X-Windows Logical Font Description format (XLFD) is also supported.

The native handle can be obtained using the "**PANGOFONTDESC**" attribute.

Style can also be a combination of: Small-Caps, [Ultra-Light|Light|Medium|Semi-Bold|Bold|Ultra-Bold|Heavy], [Ultra-Condensed|Extra-Condensed|Condensed|Semi-Condensed|Semi-Expanded|Expanded|Extra-Expanded|Ultra-Expanded]. Those values can be used only when the string is a full Pango compliant font, i.e. no underline, no strikeout and size>0.

In GNOME, go to the "Appearance Preferences" tool, then in the Fonts tab change the Applications Font to affect the default font.

Examples:

```
"Times, Bold 18"
"Arial,Helvetica, 24" (list of fonts for GTK)
"Courier New, Italic Underline -30" (size in pixels)
```

Affects

All elements, since the SIZE attribute depends on the FONT attribute, except for menus.

Notes

When FONT is changed and SIZE is set, then RASTERSIZE is also updated.

Since font face names are not a standard between Windows, Motif and GTK, a few names are specially handled to improve application portability. If you want to use names that work for all systems we recommend using: Courier, Times and Helvetica (same as Motif). Those names always have a native system name equivalent. If you use those names IUP will automatically map to the native system equivalent. See the table bellow:

Motif	Windows	GTK	Description
Helvetica	Arial	Sans	without serif, variable spacing
Courier	Courier New	Monospace	with serif, fixed spacing
Times	Times New Roman	Serif	with serif, variable spacing

Auxiliary Attributes

They will change the FONT attribute, and depends on it. They are used only to set partial FONT parameters of style and size. To do that the FONT attribute is parsed, changed and updated to the new value in the common format definition. This means that if the attribute was set in X-Windows format or in the old Windows and IUP formats, the previous value will be replaced by a new value in the common format definition. Pango additional styles will also be removed.

FONTSTYLE (non inheritable)

Replaces or returns the style of the current FONT attribute.

FONTSIZE (non inheritable)

Replaces or returns the size of the current FONT attribute.

FONTFACE (read-only, non inheritable)

Returns the face name of the current FONT attribute.

CHARSIZE (read-only, non inheritable)

Returns the average character size of the current FONT attribute. This is the factor used by the SIZE attribute to convert its units to pixels.

FOUNDRY [Motif Only] (non inheritable)

Allows to select a foundry for the FONT being selected. Must be set before setting the FONT attribute.

Encoding

The number that represents each character is dependent on the encoding used. For example, in ASCII encoding the letter A has code 65, but codes above 128 can be very different according to the encoding. An encoding is defined by a charset.

IUP uses the default locale in ANSI-C. This means that it does not adopts a specific charset, and so the results can be different if the developer charset is different than the run time charset where the user is running the application. The advantage is that different charsets can be used. For example, if the developer is using a charset, and its user is also using the same encoding, then everything will work fine without the need of text encoding conversions.

GTK uses UTF-8 (ISO10646-1) as its charset for all displayed text, so IUP will automatically convert all strings to (SetAttribute) and from (GetAttribute) UTF-8. If you want to specify strings in the UTF-8 charset at the GTK driver, then set the global attribute UTF8AUTOCONVERT to NO, the default is YES. If the default locale is already UTF-8, but the given string is not UTF-8 then it will be assumed that the string uses the ISO8859-1 charset.

In the future IUP should also support Unicode strings, increasing application portability.

ISO8859-1 and Windows-1252 Displayable Characters

The Latin-1 charset (ISO8859-1) defines an encoding for all of the characters used in Western languages. The first half of Latin-1 is standard ASCII, while the second half (with the eighth bit set) contains accented characters needed for Western languages other than English. In Windows, the ISO8859-1 charset was changed, and some control characters were replaced to include more display characters, this new charset is named Windows-1252. These characters are marked in the table bellow with thick borders.

	!	"	#	$	%	&	'	(	)	*	+	,	-	.	/
32	***33***	***34***	***35***	***36***	***37***	***38***	***39***	***40***	***41***	***42***	***43***	***44***	***45***	***46***	***47***
0	1	2	3	4	5	6	7	8	9	:	;	<	=	>	?
48	***49***	***50***	***51***	***52***	***53***	***54***	***55***	***56***	***57***	***58***	***59***	***60***	***61***	***62***	***63***
@	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O
64	***65***	***66***	***67***	***68***	***69***	***70***	***71***	***72***	***73***	***74***	***75***	***76***	***77***	***78***	***79***
P	Q	R	S	T	U	V	W	X	Y	Z	[	\	]	^	_
80	***81***	***82***	***83***	***84***	***85***	***86***	***87***	***88***	***89***	***90***	***91***	***92***	***93***	***94***	***95***
`	a	b	c	d	e	f	g	h	i	j	k	l	m	n	o
96	***97***	***98***	***99***	***100***	***101***	***102***	***103***	***104***	***105***	***106***	***107***	***108***	***109***	***110***	***111***

p	q	r	s	t	u	v	w	x	y	z	{	|	}	~	
112	113	114	115	116	117	118	119	120	121	122	123	124	125	126	127
€		‚	ƒ	„	…	†	‡	ˆ	‰	Š	‹	Œ		Ž	
128	129	130	131	132	133	134	135	136	137	138	139	140	141	142	143
	‘	’	“	”	•	–	—	˜	™	š	›	œ		ž	Ÿ
144	145	146	147	148	149	150	151	152	153	154	155	156	157	158	159
	¡	¢	£	¤	¥	¦	§	¨	©	ª	«	¬		®	¯
160	161	162	163	164	165	166	167	168	169	170	171	172	173	174	175
°	±	²	³	´	µ	¶	·	¸	¹	º	»	¼	½	¾	¿
176	177	178	179	180	181	182	183	184	185	186	187	188	189	190	191
À	Á	Â	Ã	Ä	Å	Æ	Ç	È	É	Ê	Ë	Ì	Í	Î	Ï
192	193	194	195	196	197	198	199	200	201	202	203	204	205	206	207
Ð	Ñ	Ò	Ó	Ô	Õ	Ö	×	Ø	Ù	Ú	Û	Ü	Ý	Þ	ß
208	209	210	211	212	213	214	215	216	217	218	219	220	221	222	223
à	á	â	ã	ä	å	æ	ç	è	é	ê	ë	ì	í	î	ï
224	225	226	227	228	229	230	231	232	233	234	235	236	237	238	239
ð	ñ	ò	ó	ô	õ	ö	÷	ø	ù	ú	û	ü	ý	þ	ÿ
240	241	242	243	244	245	246	247	248	249	250	251	252	253	254	255

Punctuation and Symbols

Numbers

Letters

Accented

Adapted from http://en.wikipedia.org/wiki/Windows-1252.

VISIBLE

Shows or hides the element.

Value

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

Default: "YES"

Notes

An interface element is only visible if its native parent is also visible.

Affects

All controls that have visual representation.

CLIENTSIZE (read-only) (non inheritable) (since 3.0)

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. For controls that have a native representation is only available when the control is mapped.

Value

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

Affects

All elements that are containers, except menus.

See Also

SIZE, RASTERSIZE

EXPAND (non inheritable)

Allows the element to expand, fulfilling empty spaces inside its container.

It is a non inheritable attribute, but a container can inherits its parents EXPAND attribute. So although EXPAND is non inheritable, it is inheritable for containers. So if you set it at a container it will not affect its children, but a child that is a container will get the same value if not defined at the child itself.

The expansion is done equally for all expandable elements in the same container.

See the Layout Guide for more details on sizes.

Value

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

Default: "NO". For containers the default is "YES".

Affects

All elements, except menus.

MAXSIZE (non inheritable) (since 3.0)

Specifies the element maximum size in pixels during the layout process.

See the Layout Guide for more details on sizes.

Value

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

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").

Affects

All, except menus and the dialog.

The **IupDialog** has the same attribute but within a different behavior although with a very similar objective.

Notes

The limits are applied during the layout computation. It will limit the Natural size and the Current size.

If the element can be expanded, then its empty space will NOT be occupied by other controls although its size will be limited.

See the Layout Guide for mode details on sizes.

See Also

RASTERSIZE, MINSIZE

MINSIZE (non inheritable) (since 3.0)

Specifies the element minimum size in pixels during the layout process.

See the Layout Guide for more details on sizes.

Value

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

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").

Affects

All, except menus and the dialog.

The **IupDialog** has the same attribute but within a different behavior although with a very similar objective.

Notes

The limits are applied during the layout computation. It will limit the Natural size and the Current size.

If the element can be expanded, then its empty space will NOT be occupied by other controls although its size will be limited.

See the Layout Guide for mode details on sizes.

See Also

RASTERSIZE, MAXSIZE

RASTERSIZE (non inheritable)

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

See the Layout Guide for more details on sizes.

Value

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

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.

Affects

All, except menus.

Notes

When this attribute is set, it resets the SIZE attribute. So changes to the FONT attribute will not affect the **User** size of the element.

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.

See Also

SIZE, FONT

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

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

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.

So, a SIZE="4x8" means 1 character width and 1 character height.

Notice that this is the average character size, the space occupied by a specific string is always different than its number of character times its average character size, except when using a monospaced font like Courier. Usually for common strings this size is smaller than the actual size, so it is a good practice to leave more room than expected if you use the SIZE attribute. For smaller font sizes this difference is more noticeable than for larger font sizes.

When this attribute is changed, the RASTERSIZE attribute is automatically updated.

SIZE depends on FONT, so when **FONT** is changed and **SIZE** is set, then **RASTERSIZE** is also updated.

The average character size of the current **FONT** can be obtained from the CHARSIZE attribute.

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

FONT, RASTERSIZE, IupRefresh

FLOATING (non inheritable) (since 3.0)

If an element has FLOATING=YES then its size and position will be ignored by the layout processing in **IupVbox**, **IupHbox** and **IupZbox**.

This is usefull when you do not want that an invisible element to be computed in the box size. But if the element is visible it must be manually sized and positioned using **SIZE/RASTERSIZE** and **POSITION** attributes. Default: "NO". (since 3.0)

Value

"YES" or "NO".

Default: "NO".

Affects

All elements, except menus.

See Also

IupZbox, IupVBox, IupHBox

POSITION (read-only) (non inheritable)

The position of the element relative to the origin of the client area of the native parent.

It is normally read-only, but if FLOATING=YES then POSITION can be set.

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, FLOATING

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.

NAME (non inheritable) (since 3.0)

Name of the control inside the dialog. Not releated to IupSetHandle.

Value

Text.

Notes

The NAME value will be used by IupGetDialogChild to find a child inside a dialog.

Affects

All controls.

See Also

IupGetDialogChild

TIP (non inheritable)

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

Value

Text.

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

These attributes affect the TIP display.

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

TIPBGCOLOR [Windows and Motif Only]: The tip background color. Default: "255 255 225" (Light Yellow)

TIPDELAY [Windows and Motif Only]: Time the tip will remain visible. Default: "5000".

TIPFGCOLOR [Windows and Motif Only]: The tip text color. Default: "0 0 0" (Black)

TIPFONT [Windows and Motif Only]: The font for the tip text. If not defined the font used for the text is the same as the FONT attribute for the element. If the value is SYSTEM then, no font is selected and the default system font for the tip will be used.

TIPICON [GTK only]: name of an image to be displayed in the TIP. See IupImage. (GTK 2.12)

TIPMARKUP [GTK only]: allows the tip string to contains pango markup commands. Can be "YES" or "NO". Default: "NO". Must be set before setting the TIP attribute. (GTK 2.12)

TIPRECT (non inheritable): Specifies a rectangle inside the element where the tip will be activated. Format: "%d %d %d %d"="x1 y1 x2 y2". Default: all the element area. (GTK 2.12)

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. (GTK 2.12)

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: ""

Notes

The '\n' character usually is accepted for line change (except for menus).

The "&" character can be used to define a MNEMONIC, use "&&" to show the "&" character instead on defining a mnemonic.

If a mnemonic is defined then the character relative to it is underlined and a key is associated so that when pressed together with the Alt key activates the control.

In GTk, if you define a mnemonic using "&" and the string has an underscore, then make sure that the mnemonic comes before the underscore.

In GTK, if the MARKUP attribute is defined then the title string can contains pango markup commands. Works only if a mnemonic is NOT defined in the title. Not valid for menus.

Affects

All elements with an associated text.

See Also

FONT

VALUE (non inheritable)

Affects several elements differently - that is, its behavior is element dependent. It is often used to change the control's main value, such as the text of a IupText.

For the IupRadio and IupZbox, elements, which are categorized as composition elements, this attribute represents the element "selected" among the others in the designed composition. To change this attribute in such cases, different mechanisms are necessary according to the programming environment used. When the elements taking part in a composition were created in C, this attribute's contents is a name that must be defined by the IupSetHandle function. When the elements were created in Lua, this attribute's contents is the name of a variable - more precisely, the one receiving the return from the function that created the element you wish to select. In LED it is not possible to dynamically change the value of any attribute, so the elements created in this environment must be identified and manipulated in C by means of their identifying name.

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.

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

General

LANGUAGE

The language used by some pre-defined dialogs.

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

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 Control

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 "X**x**Y", example "200x200". Can be set only in GTK 2.8.

SHIFTKEY (read-only) (since 3.0)

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

CONTROLKEY (read-only) (since 3.0)

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 " ".

KEYPRESS (write-only) (since 3.0)

Sends a key press message to the element with the focus. The value is a key code. See the Keyboard Codes table for a list of the possible values.

KEYRELEASE (write-only) (since 3.0)

Sends a key release message to the element with the focus. The value is a key code. See the Keyboard Codes table for a list of the possible values.

KEY (write-only) (since 3.0)

Sends a key press and a key release messages to the element with the focus. The value is a key code. See the Keyboard Codes table for a list of the possible values.

AUTOREPEAT [Motif Only]

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.

UTF8AUTOCONVERT [GTK Only]

GTK uses UTF-8 as its charset for all displayed text, so IUP will automatically convert all strings to (SetAttribute) and from (GetAttribute) UTF-8. Default: YES. If the default locale is already UTF-8, but the given string is not UTF-8 then it will be assumed that the string uses the ISO8859-1 charset.

System Information

SYSTEMLANGUAGE (read-only)

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

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 or Windows Vista. Some known names:

- "Darwin"
- "FreeBSD"
- "Linux"
- "SunOS"
- "Solaris"
- "IRIX"
- "AIX"
- "HP-UX"
- "Win2K"
- "WinXP"
- "Vista"
- "Win7"

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.

GTKVERSION (read-only) [GTK Only]

Returns the run time version of the GTK toolkit. This is the version being used at the time of the IupOpen function was called by the application.

GTKDEVVERSION (read-only) [GTK Only]

Returns the development version of the GTK toolkit. This is the version at the time the IUP library was compiled.

MOTIFVERSION (read-only) [Motif Only]

Returns the version of the run time Motif.

MOTIFNUMBER (read-only) [Motif Only]

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

COMPUTERNAME (read-only)

Returns the hostname.

USERNAME (read-only)

Returns the user logged in.

XSERVERVENDOR (read-only) [GTK and Motif Only] (since 3.0)

X-Windows Server Vendor string.

XVENDORRELEASE (read-only) [GTK and Motif Only] (since 3.0)

X-Windows Server Vendor release number.

Screen Information

FULLSIZE (read-only)

Returns the full screen size in pixels.

String in the "*width*x*height*" 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 "*width*x*height*" format.

SCREENDEPTH (read-only)

Returns the screen depth in bits per pixel.

TRUECOLORCANVAS (read-only)

Indicates if the display allows creating TrueColor (> 8bpp) **IupCanvas** controls, even if PseudoColor is the default, i.e. even if SCREENDEPTH<=8 . Returns "YES" or "NO". Usefull in Motif.

VIRTUALSCREEN (read-only) (since 3.0) [Win32 and GTK Only]

Returns the virtual screen position and size in pixels. It is the virtual space defined by all monitors in the system.

String in the "*x y width height*" format.

MONITORSINFO (read-only) (since 3.0) [Win32 and GTK Only]

Returns the position and size in pixels of all monitors. Each monitor information is terminated by a "\n" character.

String in the "*x y width height*\n*x y width height*\n..." format.

System Data

HINSTANCE (read-only) [Win32 Only]

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

DLL_HINSTANCE [Win32 Only] (since 3.0)

This attribute changes and returns a handle (HINSTANCE) that identifies the dll where resources are stored.

APPSHELL (read-only) [Motif Only] (since 3.0)

Returns the shell Widget created by XtOpenApplication.

XDISPLAY (read-only) [GTK and Motif Only] (since 3.0)

Returns the X-Windows Display.

XSCREEN (read-only) [GTK and Motif Only] (since 3.0)

Returns the X-Windows Screen.

Default Attributes

DLGBGCOLOR

The default background color for all elements that have the background similar of the dialog.

DLGFGCOLOR (since 3.0)

The default foreground color for all elements that have text over the background of the dialog or similar. Usually is "0 0 0".

MENUBGCOLOR (since 3.0)

The default menu background color. Usually is "255 255 255".

MENUFGCOLOR (since 3.0)

The default menu foreground color. Usually is "0 0 0".

TXTBGCOLOR (since 3.0)

The default background color for editable text, also used by lists and tree. Usually is "255 255 255".

TXTFGCOLOR (since 3.0)

The default foreground color for editable text, also used by lists and tree. Usually is "0 0 0".

DEFAULTFONT

The default font used by all elements, except for menus.

DEFAULTFONTSIZE (since 3.0)

Auxiliar attribute to retrieve and set the default font size used by all elements. It retrieves the size from DEFAULTFONT. When changed will actually change the DEFAULTFONT.

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

Callbacks are used by the application to receive notifications from the system that the user or the system itself has interacted with the user interface of the application. On the other hand attributes are used by the application to communicate with the user interface system.

Even though callbacks have different purposes from attributes, they are also associated to an element by means of an name.

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"
```

Altough 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.

If you cascade many calls to **IupMainLoop** there must be a "return IUP_CLOSE" or **IupExitLoop** call for each cascade level, hiddinh all dialogs will close only one level. Call IupMainLoopLevel to obtain the current level.

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 automatically called, causing the **IupMainLoop** to return. To avoid that set LOCKLOOP=YES before hiding the last dialog.

See Also

IupOpen, IupClose, IupLoopStep, Guide/System Control, IDLE_ACTION, LOCKLOOP.

IupMainLoopLevel (since 3.0)

Returns the current cascade level of **IupMainLoop**. When no calls were done, return value is 0.

Parameters/Return

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

Returns: the cascade level.

Notes

You can use this function to check if **IupMainLoop** was already called and avoid calling it again.

A call to **IupPopup** will increase one level.

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]
```

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 Lua]
```

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**.

In Lua, callbacks are associated by simply setting a function as the value of the callback name, for example:

```
button = iup.button{...
```

```
button.action = function(...    OR
function button:action(...
```

See Also

IupGetCallback, IupSetFunction

IupSetCallbacks

Associates several callbacks to an event.

Parameters/Return

```
Ihandle* IupSetCallbacks(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 same **ih** handle.

Notes

It is useful for setting many callbacks at once while in the creation of an hierarchy of elements, just like **IupSetAttributes**.

See Also

IupSetCallback, IupSetAttributes

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 Lua]
```

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 Lua]
```

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" in 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 or messages to be processed. Often used to perform background operations.

Callback

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

Returns: if IUP_CLOSE is returned the current loop will be closed and the callback will be removed. If IUP_IGNORE is returned the callback is removed and normal processing continues.

Notes

The Idle callback will be called whenever there are no messages left to be processed. But this occurs more frequent than expected, for example if you move the mouse over the application the idle callback will be called many times because the mouse move message is processed so fast that the Idle will be called before another mouse move message is schedule to processing.

So this callback changes the message loop to a more CPU consuming one. It is important that you set it to NULL when not using it. Or the application will be consuming CPU even if the callback is doing nothing.

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

Lua Binding

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

Long Time Operations

If you create a loop or an operation that takes a long time to complete inside a callback of your application then the user interface message loop processing is interrupted until the callback returns, so the user can not click on any control of the application. But there are ways to handle that:

- call **IupLoopStep** or **IupFlush** inside the application callback when it is performing long time operations. This will allow the user to click on a cancel button for instance, because the user interface message loop will be processed.
- split the operation in several parts that are processed by the **Idle** function when no messages are left to be processed for the user interface message loop. This will make a heavy use of the CPU, even if the callback is doing nothing.
- split the operation in several parts but use a **Timer** to process each part.

If you just want to do something simple as a background redraw of an **IupCanvas**, then a better idea is to handle the "idle" state yourself. For example, register a timer for a small time like 500ms, and reset the timer in all the mouse and keyboard callbacks of the **IupCanvas**. If the timer is trigged then you are in idle state. If the **IupCanvas** loses its focus then stop the timer.

Examples

Browse for Example Files

See Also

IupSetFunction, IupTimer.

MAP_CB

Called right after an element is mapped and its layout updated.

Callback

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

ih: identifier of the element that activated the event.

Affects

All that have a native representation.

UNMAP_CB

Called right before an element is unmapped.

Callback

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

ih: identifier of the element that activated the event.

Affects

All that have a native representation.

GETFOCUS_CB

Action generated when an element is given keyboard focus. This callback is called after the KILLFOCUS_CB of the element that loosed the focus. The IupGetFocus function during the callback returns the element that loosed the focus.

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, IupGetFocus, IupSetFocus

KILLFOCUS_CB

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

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, IupGetFocus, IupSetFocus

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(c: number) -> (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.

Returns: If IUP_IGNORE is returned the key is ignored and not processed by the control and not propagated. If returns IUP_CONTINUE, the key will be processed and the event will be propagated to the parent of the element receiving it, this is the default behavior. If returns IUP_DEFAULT the key is processed but it is not propagated. IUP_CLOSE will be processed.

Notes

Keyboard callbacks depend on the keyboard usage of the control with the focus. So if you return IUP_IGNORE the control will usually not process the key. But be aware that sometimes the control process the key in another event so even returning IUP_IGNORE the key can get processed. Although it will not be propagated.

IMPORTANT: The callbacks "K_*" of the dialog or native containers depend on the IUP_CONTINUE return value to work while the control is in 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.

Returns: 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.

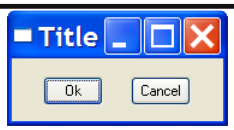
Moreover, this IUP dialog has an indirect advantage: if the user changes its size, the elements (due to being positioned on an abstract layout) are automatically re-

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

```
dialog DIALOG 0, 0, 117, 32
STYLE WS_MINIMIZEBOX | WS_MAXIMIZEBOX |
      WS_CAPTION | WS_SYSMENU |
      WS_THICKFRAME
CAPTION "Title"
BEGIN
    PUSHBUTTON "Ok",IDOK,17,9,33,15
    PUSHBUTTON "Cancel",IDCANCEL,66,9,33,15
END
```



in IupLua

```
dialog = 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="Title"
}
```

Now see the same dialog in LED and in C:

in LED

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

in C

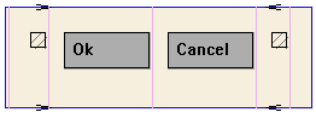
```
dialog = IupSetAttributes(IupDialog
(
  IupSetAttributes(IupHbox
  (
    IupFill(),
    IupSetAttributes(IupButton("Ok", NULL), "SIZE=40"),
    IupSetAttributes(IupButton("Cancel", NULL), "SIZE=40"),
    IupFill(),
    NULL
  ), "MARGIN=15x15, GAP=10"),
), "TITLE=Title")
```

Following, the abstract layout representation of this dialog:

Layout Hierarchy

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

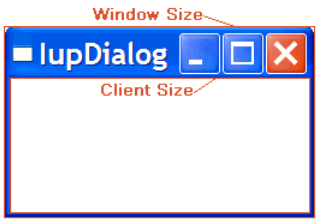
Layout Visualization

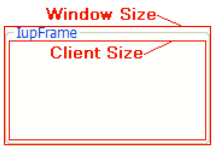


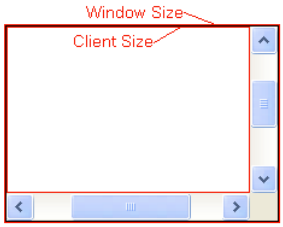
Layout Guide

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 bellow.







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

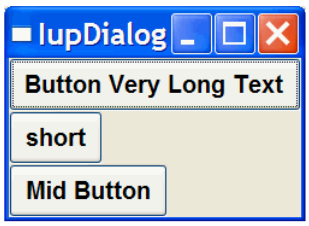
IUP does not require that the application specifies the size of any element. The sizes are automatically calculated so the contents of each element is fully displayed. This size is called **Natural** size. The **Natural** size is calculated just before the element is mapped to the native system and every time **IupMap** is called, even if the element is already mapped.

The **Natural** size of a container is the size that allows all the elements inside the container to be fully displayed. Important: even if the element is invisible its size will be included in the size of its containers, except when FLOATING=Yes.

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)

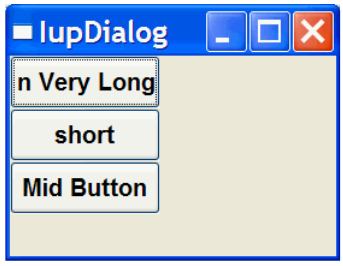
When the application defines the SIZE or RASTERSIZE attributes, it changes the **User** size in IUP. The initial internal value is "0x0". When set to NULL the **User** size is internally set to "0x0".

By default the layout computation uses the **Natural** size of the elements to compose the layout of the dialog, but if the **User** size is defined then it is used instead of the **Natural** size, except for containers (not including the dialog) where the **User** size will be used only if bigger than the **Natural** size. For the dialog, when the **User** size is not defined, the **Natural** size is used only if bigger than the **Current** size, so in this case the dialog will automatically increase its size to fit all its contents, but if the **Natural** size is smaller then the dialog size will remains the same, i.e. the dialog will not automatically shrink its size.

The returned value for SIZE or RASTERSIZE is the **Current** size in IUP. It returns the native **Window** size of the element after 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 are 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.

If the expansion is in the same direction of the box, for instance expand="VERTICAL" in the Vbox of the previous example, then the expandable elements will receive equal spaces to expand according to the remaining empty space in the box. This is why elements in diferent boxes does not align perfectly when EXPAND is set.

SHRINK

To reduze the size of the dialog and its containers 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 if the dialog size is smaller than its

Natural size.

Notice that in the example the dialog inicial size will be 0x0 because it is not defined. The picture shown was captured after manually 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 brach 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 childreen to its branch, but you can add other children using IupAppend or IupInsert. 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 theses 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 **IupAppend** 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. So explicity or implicity there will be a call to:

```
IupCreate  -> IupAppend -> IupMap
IupDestroy <- IupDetach <- IupUnmap
```

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.

In Lua, if the element was created in Lua, you can access any child of the element using the notation "elem[n][n]....", where n is the index of the child. For example:

```
dlg = iup.dialog
{
  iup.hbox
  {
    iup.button{title="Ok"},
    iup.button{title="Cancel"},
  }
}
cancel_button = dlg[1][2]
```

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 an element inside the dialog to update the dialog layout. To force a redraw of an element without layout update call IupUpdate.

The Layout update is done in two phases. First the layout is computed, this can be done without the dialog being mapped. Second is the native elements update from the computed values.

The Layout computation is done in 3 steps: **Natural** size computation, update the **Current** size and update the position.

- The **Natural** size computation is done from the inner elements up to the dialog (first for the children then the element). **User** size (set by RASTERSIZE or SIZE) is used as the **Natural** size if defined, if not usually the contents of the element are used to calculate the **Natural** size.
- Then the **Current** size is computed starting at the dialog down to the inner elements on the layout hierarchy (first the element then the children). Children **Current** size is computed according to layout distribution and containers decoration. At the children if EXPAND is set, then the size specified by the parent is used, else the natural size is used.
- Finally the position is computed starting at the dialog down to the inner elements on the layout hierarchy, after all sizes are computed.

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

classname: class name of the element to be created
params: list of parameters limited by a NULL.

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

See Also

IupAppend, IupDetach, IupMap, IupUnmap, IupDestroy, IupGetClassName

IupDestroy

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

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

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

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

Menu bars associated with dialogs are automatically destroyed when the dialog is destroyed.

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

All dialogs and all elements that have names are automatically destroyed in **IupClose**.

See Also

IupAppend, IupDetach, IupMap, IupUnmap, IupCreate

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.

Returns: IUP_NOERROR if successful. If the element was already mapped returns IUP_NOERROR. If the native creation failed returns IUP_ERROR.

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. Calling **IupMap** for an already mapped dialog is the same as only calling **IupRefresh** for the dialog.

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 a fake (-1) value.

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

This function is automatically called before the dialog is shown 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 must 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)

Unmap the element from the native system. It will also unmap all its children.

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

Parameters/Return

```
void IupUnmap(Ihandle* ih); [in C]
```

```
iup.Unmap(ih: iuplua-tag) [in Lua]
```

ih: Identifier of an interface element.

Notes

When the element is mapped some attributes are stored only in the native system. If the element is **unmaped** those attributes are lost. Use the function IupSaveClassAttributes when you want to **unmap** the element and keep its attributes.

See Also

IupAppend, IupDetach, IupMap, IupCreate, IupDestroy

IupGetClassName (renamed from IupGetType in 2.7)

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"
"cbox"
"progressbar"
"olecontrol"
```

See Also

IupGetClassType, IupGetClassAttributes

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"
```

See Also

IupGetClassName, IupGetClassAttributes

IupGetClassAttributes (Since 3.0)

Returns the names of all registered attributes of a class.

Parameters/Return

```
int IupGetClassAttributes(const char* classname, char** names, int max_n); [in C]
iup.GetClassAttributes(classname: string, max_n: number) -> (names: table, n: number) [in Lua]
```

classname: name of the class
names: table receiving the names. Only the list of names need to be allocated. Each name will point to an internal string.
max_n: maximum number of names the table can receive.

Returns: the number of names loaded to the table or -1 (nil) if class not found. If names==NULL or max_n==0 then returns the actual number of names.

See Also

IupGetClassName, IupGetClassType, IupGetAllAttributes

IupSaveClassAttributes

Saves all registered attributes on the internal hash table.

Parameters/Return

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

ih: identifier of the interface element.

Notes

When the element is mapped some attributes are stored only in the native system. If the element is **unmaped** those attributes are lost. So this function is usefull when you want to **unmap** the element and keep its attributes.

See Also

IupGetClassAttributes, IupGetClassName, IupGetClassType, IupGetAllAttributes

IupSetClassDefaultAttribute (Since 3.0)

Changes the default value of an attribute for a class. It can be any attribute, i.e. registered attributes or user custom attributes.

Parameters/Return

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

classname: name of the class
name: name of the attribute
value: new default value.

Notes

If the value is DEFAULTFONT, DLGBGCOLOR, DLGFGCOLOR, TXTBGCOLOR, TXTFGCOLOR or MENUBGCOLOR then the actual default value will be the global attribute of the same name consulted at the time the attribute is consulted.

Some attributes can NOT have a default value, so even if you set and **IupGetAttribute** return it, internally it will not be used. Those attributes are all non inheritable attributes, see the Attributes Guide in the Inheritance section.

If the new default value is (char*)-1, then the default value is set to be the system default if any is defined.

See Also

IupGetClassName, IupGetClassType, IupGetAllAttributes

IupFill

Creates a Fill composition element, which dynamically occupies empty spaces always trying to expand itself. Its parent **must** be an **IupHbox** or an **IupVbox**. If an EXPAND is set on at least one of the other children of the box, then the Fill is ignored.

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 (non inheritable): 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.

WID (read-only): returns -1 if mapped.

FONT, POSITION, MINSIZE, MAXSIZE: also accepted.

Examples

Browse for Example Files

See Also

IupHbox, IupVbox.

IupHbox

Creates an hbox container for composing elements. It is a box that arranges the elements it contains **horizontally** and left to right.

It does not have a native representation.

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 (non inheritable*): The default value is "YES". See the documentation of the attribute for EXPAND inheritance.

EXPANDCHILDREN (non inheritable): forces all children to expand vertically. Default: "NO". This has the same effect as setting EXPAND=VERTICAL on each child, but their horizontal expansion will be preserved. (since 3.0)

FLOATING (non inheritable) **(at children only)**: If a child has FLOATING=YES then its size and position will be ignored by the layout processing. Default: "NO". (since 3.0)

GAP, CGAP: Defines an horizontal space in pixels between the children, **CGAP** is in the same units of the **SIZE** attribute for the width. Default: "0". (CGAP since 3.0)

NGAP, NCGAP (non inheritable): Same as **GAP** but are non inheritable. (since 3.0)

HOMOGENEOUS (non inheritable): forces all children to get equal horizontal space. The natural size width will be based on the largest child. Default: "NO". Notice that this does not changes the children size, only the available space for each one of them to expand. (since 3.0)

MARGIN, CMARGIN: Defines a margin in pixels, **CMARGIN** is in the same units of the **SIZE** attribute. Its value has the format "*width*x*height*", where *width* and *height* are integer values corresponding to the horizontal and vertical margins, respectively. Default: "0x0" (no margin). (CMARGIN since 3.0)

NMARGIN, NCMARGIN (non inheritable): Same as **MARGIN** but are non inheritable. (since 3.0)

NORMALIZESIZE (non inheritable): normalizes all children natural size to be the biggest natural size among them. All natural width will be set to the biggest width, and all natural height will be set to the biggest height according to is value. Can be NO, HORIZONTAL, VERTICAL or BOTH. Default: "NO". (since 3.0)

SIZE / RASTERSIZE (non inheritable): Defines the width of the box. When consulted behaves as the standard SIZE/RASTERSIZE attributes.

WID (read-only): returns -1 if mapped.

FONT, CLIENTSIZE, POSITION, MINSIZE, MAXSIZE: also accepted.

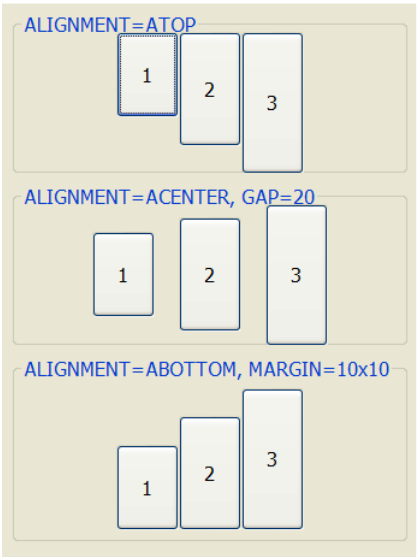
Notes

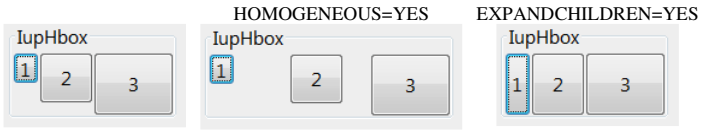
The box can be created with no elements and be dynamic filled using IupAppend.

The box will NOT expand its children, it will allow its children to expand according to the space left in the box parent. So for the expansion to occur, the children must be expandable with EXPAND!=NO, and there must be room in the box parent.

Examples

Browse for Example Files





See Also

IupZbox, IupVBox

IupVbox

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

It does not have a native representation.

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 (non inheritable*): The default value is "YES". See the documentation of the attribute for EXPAND inheritance.

EXPANDCHILDREN (non inheritable): forces all children to expand horizontally. Default: "NO". This has the same effect as setting EXPAND=HORIZONTAL on each child, but their vertical expansion will be preserved. (since 3.0)

FLOATING (non inheritable) **(at children only)**: If a child has FLOATING=YES then its size and position will be ignored by the layout processing. Default: "NO". (since 3.0)

GAP, CGAP: Defines a vertical space in pixels between the children, **CGAP** is in the same units of the **SIZE** attribute for the height. Default: "0". (CGAP since 3.0)

NGAP, NCGAP (non inheritable): Same as **GAP** but are non inheritable. (since 3.0)

HOMOGENEOUS (non inheritable): forces all children to get equal vertical space. The natural size height will be based on the highest child. Default: "NO". Notice that this does not changes the children size, only the available space for each one of them to expand. (since 3.0)

MARGIN, CMARGIN: Defines a margin in pixels, **CMARGIN** is in the same units of the **SIZE** attribute. Its value has the format "*width*x*height*", where *width* and *height* are integer values corresponding to the horizontal and vertical margins, respectively. Default: "0x0" (no margin). (CMARGIN since 3.0)

NMARGIN, NCMARGIN (non inheritable): Same as **MARGIN** but are non inheritable. (since 3.0)

NORMALIZESIZE (non inheritable): normalizes all children natural size to be the biggest natural size among them. All natural width will be set to the biggest width,

and all natural height will be set to the biggest height according to is value. Can be NO, HORIZONTAL, VERTICAL or BOTH. Default: "NO". (since 3.0)

SIZE / RASTERSIZE (non inheritable): Defines the height of the box. When consulted behaves as the standard SIZE/RASTERSIZE attributes.

WID (read-only): returns -1 if mapped.

FONT, CLIENTSIZE, POSITION, MINSIZE, MAXSIZE: also accepted.

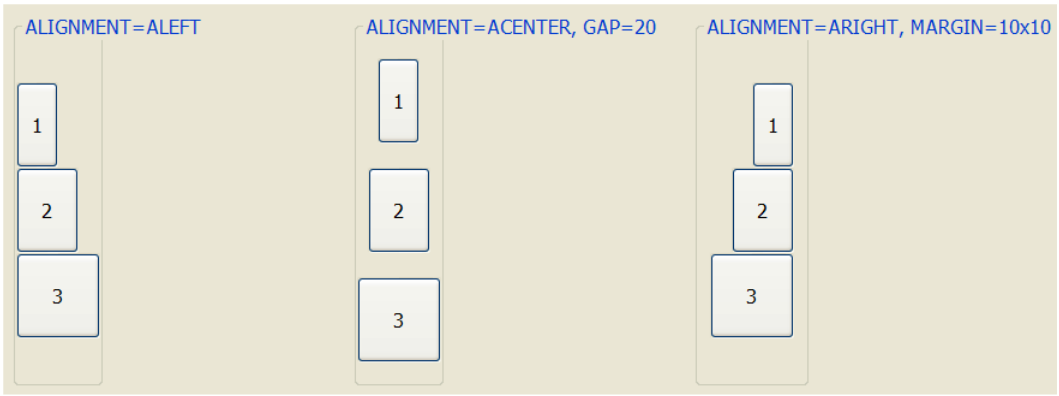
Notes

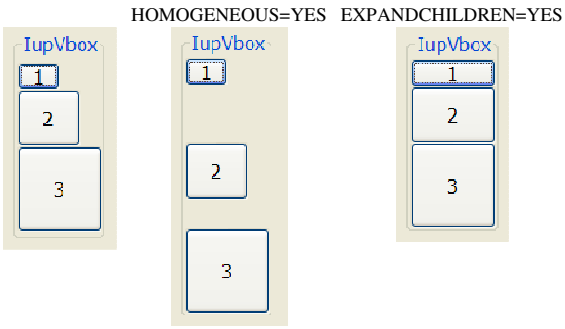
The box can be created with no elements and be dynamic filled using IupAppend.

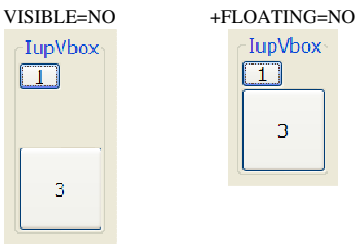
The box will NOT expand its children, it will allow its children to expand according to the space left in the box parent. So for the expansion to occur, the children must be expandable with EXPAND!=NO, and there must be room in the box parent.

Examples

Browse for Example Files







See Also

IupZbox, IupHbox

IupZbox

Creates a zbox container for composing elements. It is a box that piles up the children it contains, only the one child is visible.

It does not have a native representation. Zbox works by changing the VISIBLE attribute of its children, so if any of the grand children has its VISIBLE attribute directly defined then Zbox will NOT change its state.

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 (non inheritable): Defines the alignment of the visible child. Possible values:

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

Default: "NW".

EXPAND (non inheritable): The default value is "YES".

FLOATING (non inheritable) **(at children only)**: If a child has FLOATING=YES then its size and position will be ignored by the layout processing. Default: "NO". (since 3.0)

VALUE (non inheritable): The visible child accessed by its name. The value passed must be the name of one of the children contained in the zbox. Use IupSetHandle or IupSetAttributeHandle to associate a child to a name. In Lua you can also use the element reference directly. When the value is changed the selected child is made visible and all other children are made invisible, regardless their previous visible state.

VALUE_HANDLE (non inheritable): The visible child accessed by its handle. The value passed must be the handle of a child contained in the zbox. When the zbox is created, the first element inserted is set as the visible child. (since 3.0)

VALUEPOS (non inheritable): The visible child accessed by its position. The value passed must be the index of a child contained in the zbox, starting at 0. When the zbox is created, the first element inserted is set as the visible child. (since 3.0)

SIZE / RASTERSIZE (non inheritable): The default size is the smallest size that fits its largest child. All child elements are considered even invisible ones, except when FLOATING=YES in a child.

WID (read-only): returns -1 if mapped.

FONT, CLIENTSIZE, POSITION, MINSIZE, MAXSIZE: also accepted.

Notes

The box can be created with no elements and be dynamic filled using IupAppend.

Examples

Browse for Example Files

See Also

IupHbox, IupVBox

IupRadio

Creates the radio element for grouping mutual exclusive toggles. Only one of its descendent 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

EXPAND (non inheritable): The default value is "YES".

VALUE (non inheritable): name identifier of the active toggle. The name is set by means of IupSetHandle. In Lua you can also use the element reference directly. When consulted if the toggles are not mapped into the native system the return value may be NULL or invalid.

VALUE_HANDLE (non inheritable): Changes the active toggle. The value passed must be the handle of a child contained in the radio. When consulted if the toggles are not mapped into the native system the return value may be NULL or invalid. (since 3.0)

WID (read-only): returns -1 if mapped.

FONT, CLIENTSIZE, POSITION, MINSIZE, MAXSIZE, VISIBLE: also accepted.

Examples

Browse for Example Files

IupNormalizer (since 3.0)

Normalizes all controls from a list so their natural size to be the biggest natural size among them. All natural width will be set to the biggest width, and all natural height will be set to the biggest height according to is value.

Creation

```
Ihandle* IupNormalizer(Ihandle *ih_first, ...); [in C]
Ihandle* IupNormalizerv(Ihandle **ih_list); [in C]
iup.normalizer{ih_first, ...: ihandle} -> (elem: ihandle) [in Lua]
normalizer(ih_first, ...) [in LED]
```

ih_first, ... : List of the identifiers that will be normalized. 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

NORMALIZE (non inheritable): normalization direction. Can be HORIZONTAL, VERTICAL or BOTH. These are the same values of the NORMALIZESIZE attribute. Default: HORIZONTAL.

NORMALIZERGROUP (non inheritable) **(for contained controls use)**: name of a normalizer to automatically add the control. If a normalizer with that name does not exists then one is created.

ADDCONTROL (non inheritable): Adds a control to the normalizer. The value passed must be the name of an element. Use IupSetHandle or IupSetAttributeHandle to associate an element to a name. In Lua you can also use the element reference directly.

ADDCONTROL_HANDLE (non inheritable): Adds a control to the normalizer. The value passed must be a handle of an element.

Notes

It is NOT necessary to add the normalizer to a dialog hierarchy. Every time the NORMALIZE attribute is set a normalization occurs. If the normalizer is added to a dialog hierarchy, then whenever the **Natural** size is calculated a normalization occurs, so add it to the hierarchy before the elements you want to normalize or its normalization will be not used.

The elements do NOT need to be children of the same parent, do NOT need to be mapped, and do NOT need to be in a complete hierarchy of a dialog.

The elements are NOT children of the normalizer. To remove or add other elements, the normalizer must be destroyed and created a new one.

Has the same effect of the NORMALIZESIZE attribute of the **IupVbox** and **IupHbox** controls, but it can be used for elements with different parents, it changes the **User** size of the elements.

See Also

IupHbox, IupVbox

IupCbox

Creates a Cbox element. It is a **concrete** layout container, i.e. its children are positioned in specified coordinates.

The **IupCbox** is equivalent of a **IupVbox** or **IupHbox** where all the children have the FLOATING attribute set to YES, but children must use CX and CY attributes instead of the POSITION attribute.

Creation

```
Ihandle* IupCbox(Ihandle* child, ...); [in C]
Ihandle* IupCboxv(Ihandle** children); [in C]
iup.cbox{child, ...: ihandle} -> (elem: ihandle) [in Lua]
cbox(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

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

EXPAND (non inheritable): The default value is "YES".

SIZE / RASTERSIZE (non inheritable): Must be defined for each child. If not defined for the box, then it will be the bounding box that includes all children in their position.

WID (read-only): returns -1 if mapped.

FONT, CLIENTSIZE, POSITION, MINSIZE, MAXSIZE: also accepted.

Notes

The box can be created with no elements and be dynamic filled using IupAppend.

Examples

Browse for Example Files

See Also

IupVbox, IupHbox

IupSbox

Creates an user expander container. Allows the provided control to be enclosed in a box that allows expanding and contracting the element **size** in one direction.

It does not have a native representation but it uses a **IupCanvas** to implement the bar handler.

Creation

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

COLOR: Changes the color of the bar handler. The value should be given in "R G B" color style. Default: "192 192 192".

DIRECTION (creation only): Indicates the direction of the resize and the position of the bar handler. Possible values are "NORTH", "SOUTH" (vertical direction), "EAST" or "WEST" (horizontal direction). Default: "EAST".

EXPAND (non inheritable): It will expand automatically only in the direction opposite to the handler.

WID (read-only): returns -1 if mapped.

FONT, SIZE, RASTERSIZE, CLIENTSIZE, POSITION, MINSIZE, MAXSIZE: also accepted.

Notes

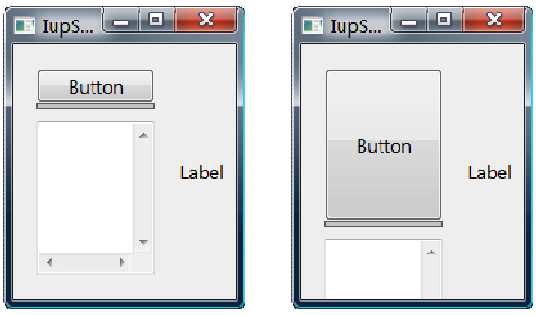
The controls that you want to be resized must have the EXPAND=YES attribute set.

The **IupCanvas** bar handler is always the first child of the sbox. It can be obtained using **IupGetChild** or **IupGetNextChild**.

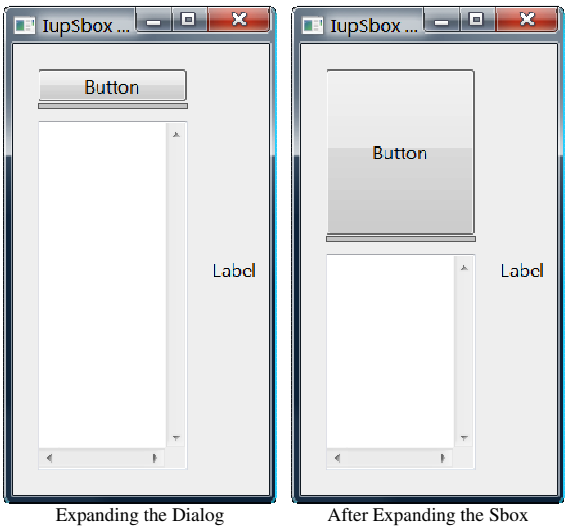
IupSbox can be resized above the maximum size so some controls go to outside the dialog area at right or bottom. In fact this is part of the dynamic layout default reposition of controls inside the dialog. See the **IupRefresh** function. The IUP layout does not have a maximum limit only a minimum.

Examples

Browse for Example Files



Natural Size | After Expanding the Sbox



Expanding the Dialog | After Expanding the Sbox

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.

Parameters/Return

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

ih: Identifier of a container like hbox, vbox, zbox and menu.
new_child: Identifier of the element to be inserted.

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.

The new child can NOT be mapped. It will NOT map the new child into the native system.

If the actual parent is a layout box (**IupVbox**, **IupHbox** or **IupZbox**) and you try to append a child that it is already at the parent child list, then the child is moved to the last child position.

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 (or any other element in the dialog) to update the dialog layout.

See Also

IupDetach, IupInsert, IupHbox, IupVbox, IupZbox, IupMenu, IupMap, IupUnmap, IupRefresh

IupDetach

Detaches an interface element from its parent.

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

It will automatically call **IupUnmap** to **unmap** the element if necessary, and then **detach** the element.

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 container (or any other element in the dialog) to update the dialog layout.

When the element is mapped some attributes are stored only in the native system. If the element is **unmaped** those attributes are lost. Use the function IupSaveClassAttributes when you want to **unmap** the element and keep its attributes.

See Also

IupAppend, IupInsert, IupRefresh, IupUnmap, IupCreate, IupDestroy

IupInsert (Since 3.0)

Inserts an interface element before another child of the container. Valid for any element that contains other elements like dialog, frame, hbox, vbox, zbox, menu, etc.

Parameters/Return

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

ih: Identifier of a container like hbox, vbox, zbox and menu.
ref_child: Identifier of the element to be used as reference. Can be NULL to insert as the first element.
new_child: Identifier of the element to be inserted before the reference.

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.

The new child can NOT be mapped. It will NOT map the new child into the native system.

If the actual parent is a layout box (**IupVbox**, **IupHbox** or **IupZbox**) and you try to insert a child that it is already at the parent child list, then the child is moved to the insert position.

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

IupAppend, IupDetach, IupHbox, IupVbox, IupZbox, IupMenu, IupMap, IupUnmap, IupRefresh

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(child, parent: ihandle) [in Lua]
```

child: Identifier of the element to be moved.
parent: Identifier of the new parent.

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 (or any other element in the dialog) to update the dialog layout.

See Also

IupAppend, IupInsert, IupDetach, IupMap, IupUnmap, IupRefresh

IupGetParent

Returns the parent of a control.

Parameters/Return

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

ih: identifier of the interface element.

Returns: the handle of the parent or NULL if does not have a parent.

See Also

IupGetChild, IupGetNextChild, IupGetBrother

IupGetChild

Returns the a child of the given control given its position.

Parameters/Return

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

IupGetChildPos, IupGetNextChild, IupGetBrother, IupGetParent

IupGetChildPos (since 3.0)

Returns the position of a child of the given control.

Parameters/Return

```
int IupGetChildPos(Ihandle* ih, Ihandle* child); [in C]
iup.GetChildPos(ih, child: ihandle) ->  pos: number [in Lua]
```

ih: identifier of the interface element.
pos: position of the desire child or -1 if child not found.

Notes

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

See Also

IupGetChild, IupGetChildCount, IupGetNextChild, IupGetBrother, IupGetParent

IupGetChildCount(since 3.0)

Returns the number of children of the given control.

Parameters/Return

```
int IupGetChildCount(Ihandle* ih); [in C]
iup.GetChildCount(ih: ihandle) ->  pos: number [in Lua]
```

ih: identifier of the interface element.

See Also

IupGetChildPos, IupGetChild, IupGetNextChild, IupGetBrother, IupGetParent

IupGetNextChild

Returns the a child of the given control given its brother.

Parameters/Return

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

ih: identifier of the interface element. Can be NULL if child not NULL.
child: Identifier of the child brother to be used as reference. To get the first child use NULL.

Returns: the handle of the child or NULL.

Notes

This function will return the children of the control in the exact same order in which they were assigned. If child in not NULL then it returns exactly the same result as IupGetBrother.

Example

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

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

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

  IupOpen(&argc, &argv);

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

  vbox = IupVbox(bt, lb, NULL);

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

  child = IupGetNextChild(vbox, NULL);

  while(child)
  {
    printf("vbox has a child of type %s\n", IupGetClassName(child));
    child = IupGetNextChild(NULL, child);
  }

  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) -> brother: ihandle [in Lua]
```

ih: identifier of the interface element.

See Also

IupGetChild, IupGetNextChild, IupGetParent

IupGetDialog

Returns the handle of the dialog that contains that interface element. Works also for children of a menu that is associated with a dialog.

Parameters/Return

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

ih: Identifier of an interface element.

Returns: the handle of the dialog or NULL if not found.

IupGetDialogChild (since 3.0)

Returns the identifier of the child element that has the NAME attribute equals to the given value on the same dialog hierarchy. Works also for children of a menu that is associated with a dialog.

Parameters/Return

```
Ihandle* IupGetDialogChild(Ihandle *ih, const char* name); [in C]
iup.GetDialogChild(ih: ihandle, name: string) -> (ih: ihandle) [in Lua]
```

ih: Identifier of an interface element that belongs to the hierarchy.
name: name of the control to be found

Returns: NULL if not found.

Notes

This function will only found the child if the NAME attribute is set at the control.

Before the dialog is mapped the function searches the hierarchy, even if the hierarchy does not belongs to a dialog yet, but after the child is mapped the result is immediate (the hierarchy is not searched).

See Also

NAME

IupRefresh

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

Parameters/Return

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

ih: identifier of the interface element.

Notes

Can be used for any control, but it will always affect the whole dialog. Can be called even if the dialog is not mapped.

The elements are immediately repositioned, if the dialog is visible then the change will be immediately reflected on the display.

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 by the native system).

IupMap also updates the dialog layout even if the control 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
IupUpdateChildren

Mark the element or its children to be redraw when the control returns to the system.

Parameters/Return

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

ih: identifier of the interface element.

IupRedraw (since 3.0)

Force the element and its children to be redraw immediately.

Parameters/Return

```
void IupRedraw(Ihandle* ih, int children); [in C]
iup.Redraw(ih: ihandle, children: boolen) [in Lua]
```

ih: identifier of the interface element.
children: flag to update its children.

IupConvertXYToPos (since 3.0)

Converts a (x,y) coordinate in an item position.

Parameters/Return

```
int IupConvertXYToPos(Ihandle *ih, int x, int y); [in C]
iup.ConvertXYToPos(ih: ihandle, 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.
y: Y coordinate of the upper part of the interface element.

Returns: the position starting at 0. If fails returns -1.

Notes

It can be used for **IupText** (returns a position in the string), **IupList** (returns an item) or **IupTree** (returns a node identifier).

See Also

IupText, IupList, IupTree

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.

All dialogs are automatically destroyed in **IupClose**.

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. See also the screenshots of the sample.c results with normal background, changing the dialog BACKGROUND, the dialog BGCOLOR and the children BGCOLOR. (since 3.0)

BORDER (non inheritable) (creation only): Shows a resize border around the dialog. Default: "YES". BORDER=NO is useful 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.

DRAGDROP [Windows and GTK Only] (non inheritable): 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.

EXPAND (non inheritable): The default value is "YES".

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.

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.

TITLE (non inheritable): Dialog's title.

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

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

Exclusive

DEFAULTENTER: Name of the button activated when the user press Enter when focus is in another control of the dialog. Use IupSetHandle or IupSetAttributeHandle to associate a button to a name.

DEFAULTESC: Name of the button activated when the user press Esc when focus is in another control of the dialog. Use IupSetHandle or IupSetAttributeHandle to associate a button to a name.

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 over any system bars in the main monitor. 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.

HWND [Windows Only] (non inheritable, read-only): Returns the Windows Window handle. Available in the Windows driver or in the GTK driver in Windows.

MAXBOX (creation only): Requires a maximize button from the window manager. If RESIZE=NO then MAXBOX will be set to NO. Default: YES. In Motif the decorations are controlled by the Window Manager and may not be possible to be changed from IUP. 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: Name of a menu. Associates a menu to the dialog as a menu bar. The previous menu, if any, is unmapped. Use IupSetHandle or IupSetAttributeHandle to associate a menu to a name. See also IupMenu.

MENUBOX (creation only): Requires a system menu box from the window manager. If hidden will also remove the Close button. Default: YES. In Motif the decorations are controled by the Window Manager and may not be possible to be changed from IUP. In Windows if hidden will hide also MAXBOX and MINBOX.

MINBOX (creation only): Requires a minimize button from the window manager. Default: YES. In Motif the decorations are controlled by the Window Manager and may not be possible to be changed from IUP. 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)

NATIVEPARENT (creation only): Native handle of a dialog to be used as parent. Used only if PARENTDIALOG is not defined.

PARENTDIALOG (creation only): Name of a dialog to be used as parent.

PLACEMENT: Changes how the dialog will be shown. Values: "FULL", "MAXIMIZED", "MINIMIZED" and "NORMAL". Default: NORMAL. After **IupShow/IupPopup** 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. 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. In Motif the decorations are controlled by the Window Manager and may not be possible to be changed from IUP.

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.

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). Updated after SHOW_CB is called and only if the focus was not changed during the callback.

XWINDOW [UNIX Only] (non inheritable, read-only): Returns the X-Windows Window (Drawable). Available in the Motif driver or in the GTK driver in UNIX.

Exclusive [Windows and GTK Only]

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.

OPACITY [Windows and GTK Only]: sets the dialog transparency alpha value. Valid values range from 0 (completely transparent) to 255 (opaque). In Windows set to NULL to remove the LAYERED style bit. (GTK 2.12)

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

TRAY [Windows and GTK Only]: When set to "YES", displays an icon on the system tray. (GTK 2.10)

TRAYIMAGE [Windows and GTK Only]: Name of a IUP image to be used as the tray icon. (GTK 2.10)

TRAYTIP [Windows and GTK Only] :Tray icon's tooltip text. (GTK 2.10)

Exclusive [GTK Only]

DIALOGHINT (creation-only): if enabled sets the window type hint to a dialog hint.

Exclusive [Windows Only]

BRINGFRONT [Windows Only] (write-only): makes the dialog the foreground window. Use "YES" to activate it. Useful for multithreaded applications.

COMPOSITED [Windows Only] (creation only): controls if the window will have an automatic double buffer for all children. Default is "NO". In Windows Vista it is NOT working as expected.

CONTROL [Windows Only] (creation only): Embeds the dialog inside another window.

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.

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

Exclusive MDI [Windows Only]

--- For the MDI Frame ---

MDIFRAME (creation only) [Windows Only] (non inheritable): Configure this dialog as a MDI frame. Can be YES or NO. Default: NO.

MDIACTIVE [Windows Only] (read-only): Returns the name of the current active MDI child. Use IupGetAttributeHandle to directly retrieve the child handle.

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).

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.

MDINEXT [Windows Only] (read-only): Returns the name of the next available MDI child. Use IupGetAttributeHandle to directly retrieve the child handle. 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.

--- For the MDI Client (a IupCanvas) **---**

MDICLIENT (creation only) [Windows Only] (non inheritable): Configure the canvas as a MDI client. Can be YES or NO. No callbacks will be called. This canvas will be used internally only by the MDI Frame and its MDI Children. The MDI frame must have one and only one MDI client. Default: NO.

MDIMENU (creation only) [Windows Only]: Name of a IupMenu to be used as the Window list of a MDI frame. The system will automatically add the list of MDI

child windows there.

--- For the MDI Children ---

MDICHILD (creation only) [Windows Only]: Configure this dialog to be a MDI child. Can be YES or NO. The PARENTDIALOG attribute must also be defined. Each MDI child is automatically named if it does not have one. Default: NO.

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 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.

MOVE_CB [Windows and GTK Only]: Called after the dialog was moved on screen. The coordinates are the same as the X and Y attributes. (since 3.0)

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

ih: identifier of the element that activated the event.
x, **y**: coordinates of the new position.

RESIZE_CB: Action generated when the dialog size is changed. If returns IUP_IGNORE the dialog layout is NOT recalculated. (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. (GTK 2.10)

```
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, UNMAP_CB, GETFOCUS_CB, KILLFOCUS_CB, ENTERWINDOW_CB, LEAVEWINDOW_CB, K_ANY, HELP_CB: All common callbacks are supported.

Notes

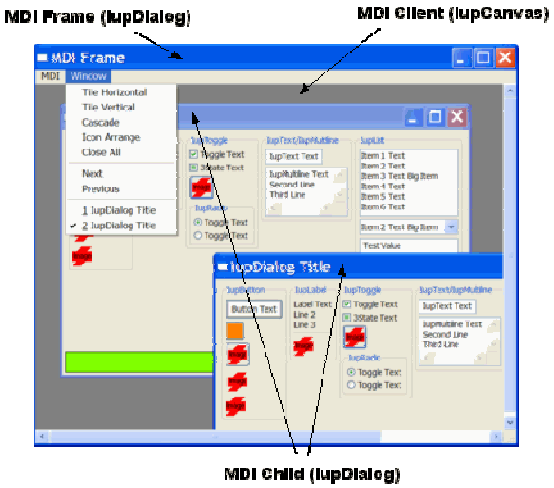
Do not associate an **IupDialog** with the native "dialog" nomenclature in Windows, GTK or Motif. **IupDialog** use native standard windows in all drivers.

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.

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

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;
}
```

Browse for Example Files

See Also

IupFileDlg, IupMessageDlg, IupDestroy, IupShowXY, IupShow, IupPpopup

CURSOR (non inheritable)

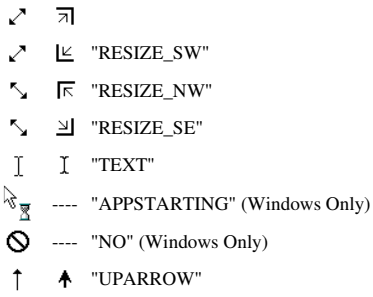
Defines the element's cursor.

Value

Name of a cursor.

It will check first for the following predefined names:

		Name
		"NONE" or "NULL"
		"ARROW"
		"BUSY"
		"CROSS"
		"HAND"
		"HELP"
		"MOVE"
		"PEN" (*)
		"RESIZE_N"
		"RESIZE_S"
		"RESIZE_NS"
		"RESIZE_W"
		"RESIZE_E"
		"RESIZE_WE"
		"RESIZE_NE"



Default: "ARROW"

(*) To use these cursors on Windows, the **iup.rc** file, provided with IUP, must be linked with the application (except when using the IUP DLL).

The GTK cursors have the same appearance of the X-Windows cursors. Althought GTK cursors can have more than 2 colors depending on the X-Server.

If it is not a pre-defined name, then will check for other system cursors. In Windows the value will be used to load a cursor form the application resources. In Motif the value will be used as a X-Windows cursor number, see definitions in the X11 header "cursorfont.h". In GTK the value will be used as a cursor name, see the GDK documentation on Cursors.

If no system cursors were found then the value will be used to try to find an IUP image with the same name. Use **IupSetHandle** to define a name for an **IupImage**. But the image will need an extra attribute and some specific characteristics, see notes bellow.

Notes

For an image to represent a cursor, it should has the attribute "**HOTSPOT"** to define the cursor hotspot (place where the mouse click is actually effective). The default value is "0:0".

Usually only color indices 0, 1 and 2 can be used in a cursor, where 0 will be transparent (must be "BGCOLOR"). The RGB colors corresponding to indices 1 and 2 are defined just as in regular images. In Windows and GTK the cursor can have more than 2 colors. Cursor sizes are usually less than or equal to 32x32.

The cursor will only change when the interface system regains control or when IupFlush is called.

The Windows SDK recommends that cursors and icons should be implemented as resources rather than created at run time.

When the cursor image is no longer necessary, it must be destroyed through function IupDestroy. Attention: the cursor cannot be in use when it is destroyed.

Affects

IupDialog, IupCanvas

See Also

IupImage

ICON

Dialog's icon. This icon will be used when the dialog is minimized.

Value

Name of a IUP image.

Default: NULL

Notes

Icon sizes are usually less than or equal to 32x32.

The Windows SDK recomends that cursors and icons should be implemented as resources rather than created at run time.

On Motif, it only works with some window managers, like *mwm* and *gnome*. Icon colors can have the BGCOLOR values, but it works better if it is at index 0.

Use IupSetHandle or IupSetAttributeHandle to associate an image to a name.

Affects

IupDialog

See Also

IupImage

PARENTDIALOG

The parent dialog of a dialog.

Value

Name of a dialog to be used as parent.

Default: NULL.

Notes

This dialog will be always in front of the parent dialog. If the parent is minimized, this dialog is automatically minimized. 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.

Use IupSetHandle or IupSetAttributeHandle to associate a dialog to a name.

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.

Affects

IupDialog

SHRINK

If this attribute is defined, the elements inside the dialog will try to adjust their sizes even when the dialog's size is smaller than its natural size.

See the Layout Guide for more details on sizes.

Value

"YES" or "NO".

Default: "NO".

Notes

When the user changes the size of the dialog, the elements are automatically re-distributed inside the dialog. Some elements even have their size changed if the EXPAND attribute is active. When this size is smaller than a minimum limit in which all elements still fit the dialog, the elements' distribution is no longer modified. Actually, the virtual size of the dialog remains larger than its actual size on the screen, and some elements to the right and bottom are hidden by the borders of the dialog.

The SHRINK attribute offers an alternative to this behavior. It makes the elements continue to rearrange, even if they must overlap.

The results of this new rearrangement may vary according to the elements' distribution on the dialog.

See the Layout Guide for more details on sizes.

Affects

IupDialog

CONTROL

Windows only. Whether the dialog is embedded inside the parent window or has a window of its own.

Value

YES or NO. If the value is YES, the dialog will appear embedded inside its parent window (you must set a parent, either with PARENTDIALOG or NATIVEPARENT, or this setting will be ignored). If the value is NO, the dialog will have its own window.

Notes

This is useful for implementing ActiveX controls, with the help of the LuaCOM library. ActiveX controls run embedded inside another window. LuaCOM will send a window creation event the the control, passing a handle to the parent window and the size of the control. You can use this to set the dialog's NATIVEPARENT and RASTERSIZE attributes. See the LuaCOM documentation for more information.

Affects

IupDialog

See Also

NATIVEPARENT, **PARENTDIALOG**, **RASTERSIZE**

CLOSE_CB

Called just before a dialog is hidden due to some action over it - for example, double clicking the system's menu box, usually located to the left in the title bar.

Callback

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

ih: identifies the element that activated the event.

Returns: if IUP_IGNORE, it prevents the dialog from being hidden. If you destroy the dialog in this callback, you must return IUP_IGNORE. IUP_CLOSE will be processed.

Affects

IupDialog

DROPFILES_CB

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

If defined after the element is mapped then the attribute DRAGDROP must be set to YES.

[Windows and GTK Only] (GTK 2.6)

Callback

```
int function(Ihandle *ih, const char* filename, int num, int x, int y); [in C]
elem:dropfiles_cb(filename: string; num, x, y: number) -> (ret: number) [in Lua]
```

ih: identifier of the element that activated the event.
filename: Name of the dragged file.
num: Number of the dragged file. If several files are dragged, num counts the number of dragged files starting from "total-1" to "0".
x: X coordinate of the point where the user released the mouse button.
y: Y coordinate of the point where the user released the mouse button.

Returns: If IUP_IGNORE is returned the callback will NOT be called for the next dropped files, and processing of dropped files will be interrupted.

Affects

IupDialog, IupCanvas, IupGLCanvas, IupText, IupList

RESIZE_CB

Action generated when the element size is changed.

Callback

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

ih: identifier of the element that activated the event.
width: the width of the internal element size in pixels not considering the decorations (client size)
height: the height of the internal element size in pixels not considering the decorations (client size)

Notes

This action is also generated when the dialog is mapped or shown. In Windows and Motif, it is generated after a map and before show.

Affects

IupCanvas, IupGLCanvas, IupDialog

SHOW_CB

Called right after the dialog is opened, minimized or restored from a minimization.

Callback

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

ih: identifier of the element that activated the event.
state: indicates which of the following situations generated the event:

> IUP_HIDE, IUP_SHOW, IUP_RESTORE (was minimized or maximized), IUP_MINIMIZE or IUP_MAXIMIZE. (IUP_HIDE and IUP_MAXIMIZE since 3.0)
>
> IUP_MAXIMIZE is not received in Motif when activated from the maximize button.

Returns: IUP_CLOSE will be processed.

Affects

IupDialog

IupPopup

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

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 exemple 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 ends 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: horizontal position of the dialog or menu relative to the origin of the screen. 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. Not valid for menus. (Since 3.0)
- IUP_CURRENT: use the current position of the dialog. This is the default value in Lua if the parameter is not defined. Not valid for menus. (Since 3.0)

y: vertical position of the dialog or menu relative to the origin of the screen. 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. Not valid for menus. (Since 3.0)
- IUP_CURRENT: use the current position of the dialog. This is the default value in Lua if the parameter is not defined. Not valid for menus. (Since 3.0)

Returns: IUP_NOERROR if sucessful. 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.

When IUP_CENTERPARENT is used but PARENTDIALOG is not defined then it is replaced by IUP_CENTER.

When IUP_CURRENT is used at the first time the dialog is shown then it will be replaced by IUP_CENTERPARENT.

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. Althougth it interrupts the processing, it does not destroy the dialog when it ends. To destroy the dialog, **IupDestroy** must be called.

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.

Returns: IUP_NOERROR if sucessful. 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 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: horizontal position of the dialog relative to the origin of the screen. 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. This is the default value in Lua if the parameter is not defined. (Since 3.0)

y: vertical position of the dialog relative to the origin of the screen. 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. This is the default value in Lua if the parameter is not defined.(Since 3.0)

Returns: IUP_NOERROR if sucessful. 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.

When IUP_CENTERPARENT is used but PARENTDIALOG is not defined then it is replaced by IUP_CENTER.

When IUP_CURRENT is used at the first time the dialog is shown then it will be replaced by IUP_CENTERPARENT.

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 **IupShowXY** 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.

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.

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 element, 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 new dialog does not have the Help button neither filters. Also this new dialog needs CoInitializeEx with COINIT_APARTMENTTHREADED (done in **IupOpen**), if the COM library was initialized

with COINIT_MULTITHREADED prior to **IupOpen** then the new dialog will have limited functionality. 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 (read-only): 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;test.*". 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 [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/test|a.txt|b.txt|c.txt|" 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 existent file when in "SAVE" dialog. Default is "NO", i.e. prompt before overwrite. (GTK 2.8)

PARENTDIALOG: Makes the dialog be treated as a child of the specified dialog.

SHOWHIDDEN: Show hidden files. Default: NO. (since 3.0) (GTK 2.6)

SHOWPREVIEW: A preview area is shown inside the File Dialog. Can have values "YES" or "NO". Default: "NO". In Windows, you must link with the "iup.rc" resource file so the preview area can be enabled (not necessary if using "iup.dll"). Valid only if the FILE_CB callback is defined, use it to retrieve the file name and the necessary attributes to paint the preview area. (in Motif since 3.0)

Read only attributes that are valid inside the FILE_CB callback when status="PAINT":
PREVIEWDC: Return the Device Context (HDC in Windows and GC in UNIX)
PREVIEWWIDTH and **PREVIEWHEIGHT**: Return 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.

If the attribute PREVIEWGLCANVAS is defined then it is used as the name of an existent IupGLCanvas control to be mapped internally to the preview canvas. Notice that this is not a fully implemented IupGLCanvas that inherits from IupCanvas. This does the minimum necessary so you can use IupGLCanvas auxiliary functions for the preview canvas and call OpenGL functions. No IupCanvas attributes or callbacks are available. (since 3.0)

STATUS (read-only): Indicates the status of the selection made:

"1": New file.
"0": Normal, existing file or directory.
"-1": Operation cancelled.

TITLE: Dialog's title.

VALUE (read-only): 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. (Windows only in 2.x)

```
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.
- "SELECT" - a file has been selected.
- "OTHER" - an invalid file or a directory is selected. file_name is the one selected. (Since 3.0)
- "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 redrawn. Used only when SHOWPREVIEW=YES. If an invalid file or a 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.

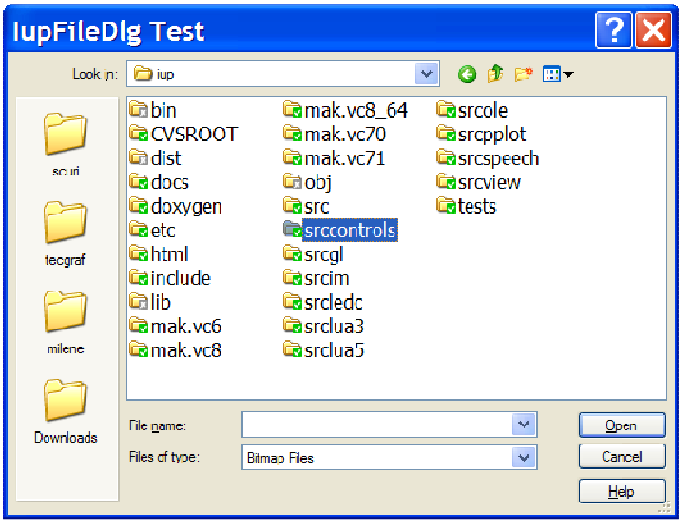
The IupGetFile function simply creates and popup a IupFileDlg.

In Windows, the FILE and the DIRECTORY attributes also accept strings containing "/" as path separators, but the VALUE attribute will always return strings using the "\" character.

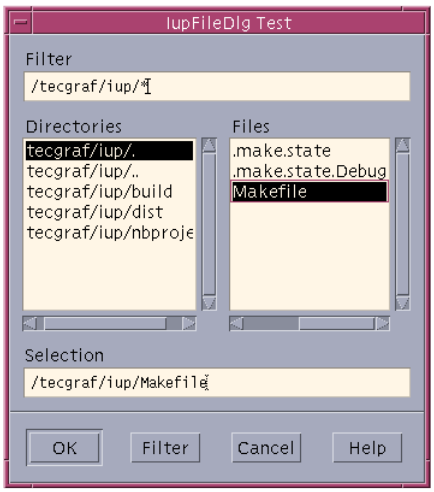
Examples

```
Ihandle *dlg = IupFileDlg();

IupSetAttribute(dlg, "DIALOGTYPE", "OPEN");
IupSetAttribute(dlg, "TITLE", "IupFileDlg 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);
```

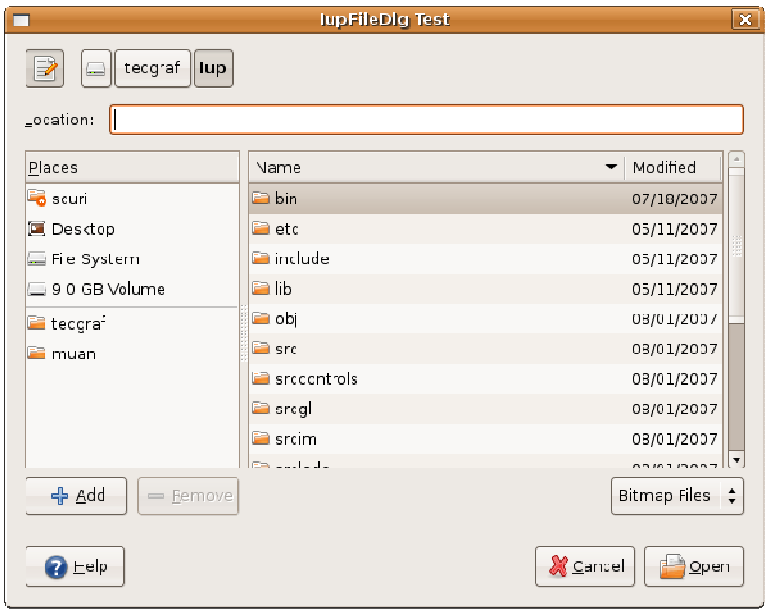
Windows XP



Motif/Mwm



GTK/GNOME



Browse for Example Files

See Also

IupMessage, IupScanf, IupListDialog, IupAlarm, IupGetFile, IupPopup

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. Default: "1".

BUTTONRESPONSE: Number of the pressed button. Can be "1" or "2". Default: "1".

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.

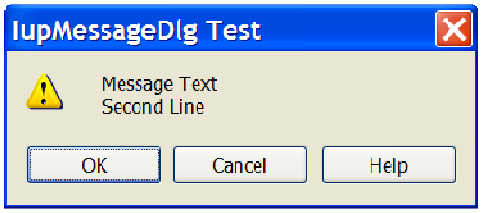
The **IupMessage** function simply creates and popup a **IupMessageDlg**.

In Windows each different dialog type is always associated with a different beep sound.

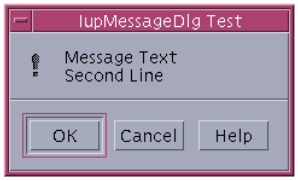
Examples

```
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);
```

Windows XP



Motif/Mwm



GTK/GNOME

See Also

IupMessage, IupScanf, IupListDialog, IupAlarm, IupGetFile, IupPopup

IupColorDlg (since 3.0)

Creates the Color Dialog element. It is a predefined dialog for selecting a color.

There are 3 versions of the dialog. One for Windows only, one for GTK only and one for all systems, but it is based on the **IupColorBrowser** control that depends on the CD library.

The Windows and GTK dialogs can be shown only with the **IupPopup** function. The **IupColorBrowser** based dialog is a **IupDialog** that can be shown as any regular **IupDialog**.

IMPORTANT: The **IupColorBrowser** based dialog is included in the Controls Library. When the Controls Library is initialized the Windows and GTK dialogs are not available anymore, i.e. before the Controls Library initialization only the Windows and GTK dialogs are available, after only the **IupColorBrowser** based dialog is available.

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 [ColorBrowser and GTK only]: if defined it will enable an alpha selection additional controls with its initial value. If the user pressed the Ok button contains the returned value. Default: no defined, or 255 if SHOWALPHA=YES.

COLORTABLE: list of colors separated by ";". In GTK and in the ColorBrowser based 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). If a color is not specified then the default color is

used. You can skip colors using ";;".

PARENTDIALOG (creation only): Name of a dialog to be used as parent. This dialog will be always in front of the parent dialog.

SHOWALPHA [ColorBrowser and GTK only]: if enabled will display the alpha selection controls, regardless if ALPHA is defined for the initial value or not.

SHOWCOLORTABLE: if enabled will display the color table, regardless if **COLORTABLE** is defined or not. The default colors in the color table are different in GTK and in the ColorBrowser based dialog. In Windows the default colors are all black.

SHOWHEX [ColorBrowser only]: if enabled will display the Hexadecimal notation of the color.

SHOWHELP [ColorBrowser only]: if enabled will display the Help button. In GTK and Windows, the Help button is shown only if the HELP_CB callback is defined.

STATUS (read-only): defined to "1" if the user pressed the Ok button, NULL if pressed the Cancel button.

TITLE: Dialog title.

VALUE: The color value in RGB coordinates and optionally alpha. It is used as the initial value and contains the selected value if the user pressed the Ok button. Format: "R G B" or "R G B A". Each component range from 0 to 255.

VALUEHSI [ColorBrowser only]: The color value in HSI coordinates. It is used as the initial value and contains the selected value if the user pressed the Ok button. Format: "H S I". Each component range from 0-359, 0-100 and 0-100 respectively.

VALUEHEX [ColorBrowser only]: The color value in RGB Hexadecimal notation. It is used as the initial value and contains the selected value if the user pressed the Ok button. Format: "#RRGGBB". Each component range from 0-255, but in hexadecimal notation.

Callbacks

HELP_CB: Action generated when the Help button is pressed.

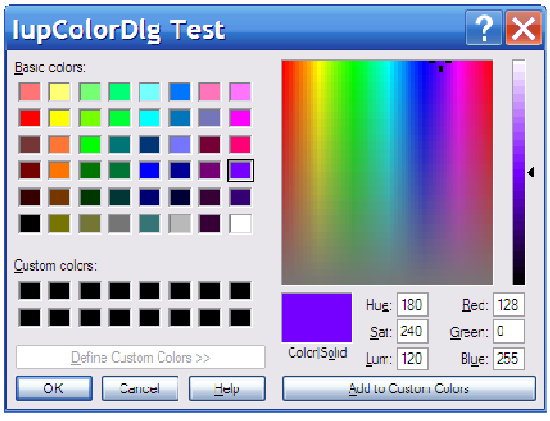
Notes

The GTK and Windows dialogs are native pre-defined dialogs that are 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.

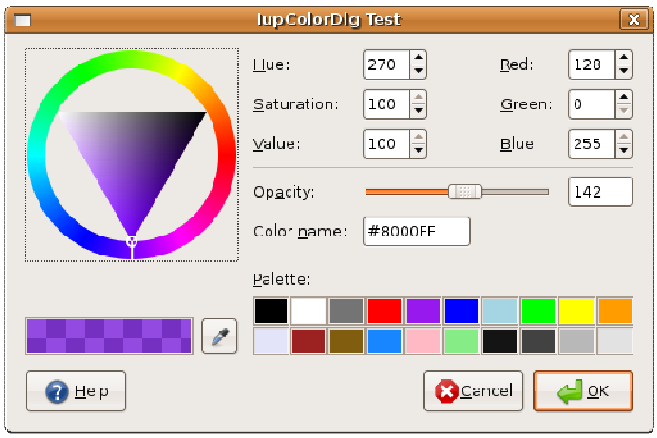
Examples

```
Ihandle* dlg = IupColorDlg();

IupSetAttribute(dlg, "VALUE", "128 0 255");
IupSetAttribute(dlg, "ALPHA", "142");
IupSetAttribute(dlg, "SHOWHEX", "YES");
IupSetAttribute(dlg, "SHOWCOLORTABLE", "YES");
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);
```

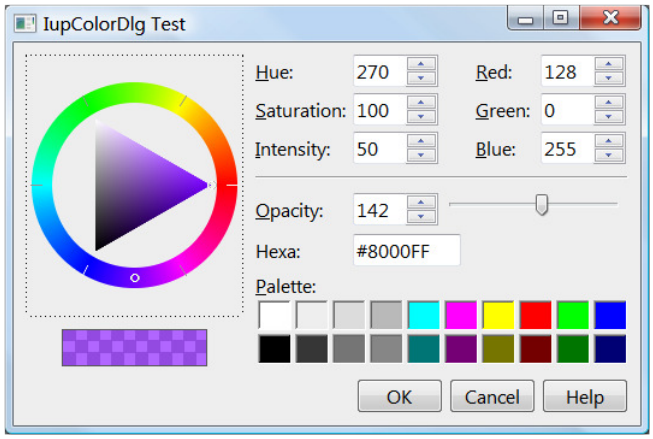
Windows XP



GTK/GNOME



ColorBrowser Based



See Also

IupMessageDlg, IupFileDlg, IupPopup

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]: The initial color value and the returned selected value if the user pressed the Ok button. In Windows the Choose Font dialog allows hte 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 (read-only): defined to "1" if the user pressed the Ok button, NULL if pressed the Cancel button.

TITLE: Dialog title.

VALUE: 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**.

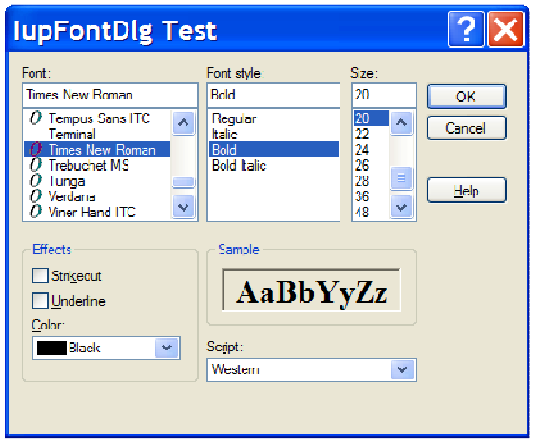
To show the dialog, use function **IupPopup**. In Lua, use the **popup** function.

The dialog is mapped only inside **IupPopup**, **IupMap** does nothing.

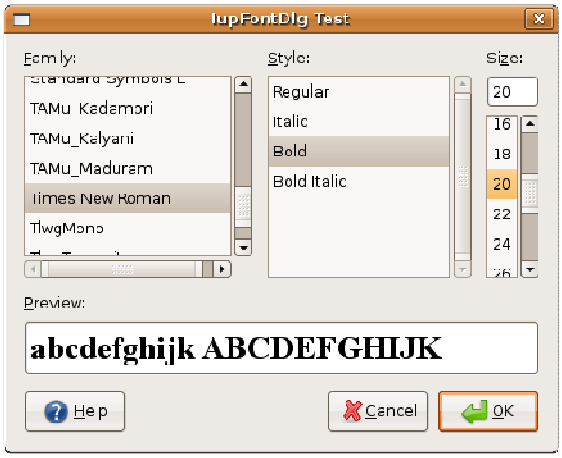
Examples

```
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, IupFileDlg, 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)

Returns: the number of the **button** selected by the user (1, 2 or 3) , or 0 if failed. It fails only if b1 is not defined.

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.

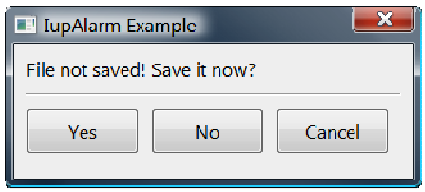
A button is not shown if its parameter is NULL. This is valid only for **b2** and **b3**.

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

Browse for Example Files



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.

Returns: a **status** code, whose values can be:

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

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 function will reuse the directory from one call to another, so in the next call will open in the directory of the last selected file.

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

IupFileDlg, IupMessage, IupScanf, IupListDialog, IupAlarm, IupSetLanguage.

IupGetColor

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

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.

Returns: in C a code 1 if the OK button is pressed, or 0 otherwise. In Lua the code is not returned, instead the r,g,b values are returned or nil otherwise.

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.

Examples

Browse for Example Files

See Also

IupMessage, IupScanf, IupListDialog, IupAlarm, IupGetFile.

IupGetParam

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

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,...) -> (status: boolean, ...) [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.

Returns: a **status** code 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. In Lua, the values are returned by the function in the same order they were passed.

Callbacks

```
int function(Ihandle* dialog, int param_index, void* user_data); [in C]
luafunction(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.

Returns: 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*", where "n" is the parameter index in the order they are specified starting at 0, but separators are not counted. Notice that this is not the actual control, 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 n^{th} parameter, indexed by the declaration order not couting separators.
"OK" - returns an IUP Ihandle*, the main button.
"CANCEL" - returns an IUP Ihandle*, the close button.

For a parameter:

"LABEL" - returns an IUP Ihandle*, the label associated with the parameter.
"CONTROL" - returns an IUP Ihandle*, the real control associated with the parameter.
"AUXCONTROL" - returns an IUP Ihandle*, the auxiliary control associated with the parameter (only for Valuators).
"INDEX" - returns an integer value associated with the parameter index. **IupGetInt** can also be used.
"VALUE" - returns the parameter value as a string, but **IupGetFloat** and **IupGetInt** can also be used. It usually contains the new value of the control while the VALUE attribute of the control still has the old value.

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

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

dialog: Identifier of the dialog.
param_index: parameter to be retrieved.

Notes

The format string must have the following format, notice the "\n" at the end

"***text%x*[*extra*]{tip}**\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 number filtered text box, use "int" in C)
r = real (shows a real number filtered text box, use "float" in C)
a = angle in degrees (shows a real number filtered text box and a dial [if **IupControlsOpen** is called], 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)
c = string (same as **s**, but also show a color button to open a color 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 omited. 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 to be displayed after the toggle. The strings can not have commas ',', nor brackets '[' or ']'.
mask is an optional mask for the string and multiline types. The dialog uses the MASK attribute internally. In this case we do no use the brackets '[' and ']' to avoid confusion with the specified mask.
|item0|item1|item2,...| are the items of the list. At least one item must exist. Again the brackets are not used to increase the possibilities for the strings, instead you must use '|'. Items index are zero based start.
[dialogtype|filter|directory|nochangedir|nooverwriteprompt] are the respective attribute values passed to the IupFileDlg control when activated. All commas must exist, but you can let empty values to use the default values. No mask can be set.

tip is a string that is displayed in a TIP for the main control of the parameter. (since 3.0)

There is no extra parameters for the color string. The mask is automatically set to capture 3 or 4 unsigned integers from 0 to 255 (R G B) or (R G B A) (alpha is optional).

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

A integer parameter always has a spin attached to the text to incremente and decrement the value. A real parameter only has a spin in a full intervall is defined (min and max), in this case the default step is (max-min)/20. When the callback is called because a spin was activated then the attribute SPINNING of the dialog will be defined to a non NULL and non zero value.

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.

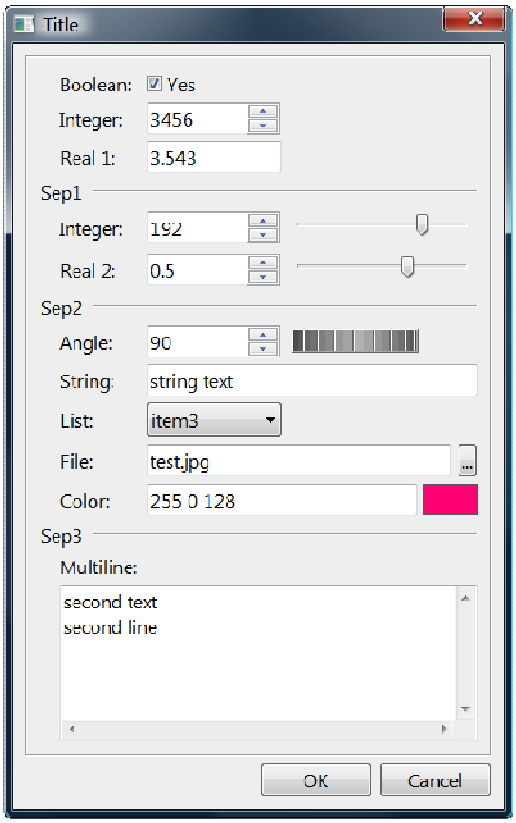
The dialog uses a global attribute called "PARENTDIALOG" as the parent dialog if it is defined. It also uses a global attribute called "ICON" as the dialog icon if it is defined.

Examples

Browse for Example Files

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";
char pcolor[100] = "255 0 128";
int plist = 2;
char pstring2[200] = "second text\nsecond line";
char pfile_name[500] = "test.jpg";

if (!IupGetParam("Title", param_action, 0,
                 "Boolean: %b[No,Yes]{Boolean Tip}\n"
                 "Integer: %i{Integer Tip}\n"
                 "Real 1: %r{Real Tip}\n"
                 "Sep1 %t\n"
                 "Integer: %i[0,255]{Integer Tip 2}\n"
                 "Real 2: %r[-1.5,1.5]{Real Tip 2}\n"
                 "Sep2 %t\n"
                 "Angle: %a[0,360]{Angle Tip}\n"
                 "String: %s{String Tip}\n"
                 "List: %l|item1|item2|item3|{List Tip}\n"
                 "File: %f[OPEN|*.bmp;*.jpg|CURRENT|NO|NO]{File Tip}\n"
                 "Color: %c{Color Tip}\n"
                 "Sep3 %t\n"
                 "Multiline: %m{Multiline Tip}\n",
                 &pboolean, &pinteger, &preal, &pinteger2, &preal2, &pangle, pstring, &plist, pfile_name, pcolor, pstring2, NULL))
  return IUP_DEFAULT;
```



See Also

IupScanf, IupGetColor, IupMask, IupVal, IupDial, IupList, IupFileDlg.

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]
```

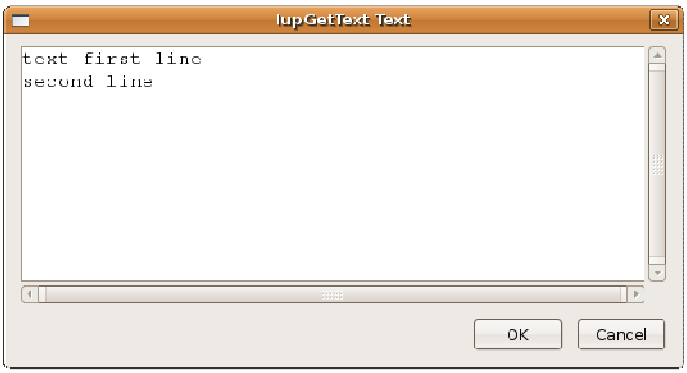
text: It contains the initial value of the text and the returned text. It must have room for the edited string.

Returns: a non zero value if successfull. In Lua returns the text or nil if an error occured.

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.

Examples



See Also

IupMessage, IupScanf, IupListDialog, IupAlarm, IupSetLanguage.

IupListDialog

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

Creation and Show

```
int IupListDialog(int type, const char *title, int size, const char** list, int op, int max_col, int max_lin, int* marks); [in C]
iup.ListDialog(type: number, title: string, size: number, list: table of strings, op: number, max_col: number, max_lin: number, marks: t
```

type: 1=simple selection; 2=multiple selection
title: Text for the dialog's title
size: Number of options
list: List of options. Must have **size** elements
op: Initial selected item when type=1. starts 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
marks: List of the items selection state, used only when type=2. Can be NULL when type=1. When type=2 must have **size** elements

Returns: When type=1, the function returns the number of the selected option (starts at 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 1 and the **marks** parameter will have value 1 for the options selected by the user and value 0 for non-selected options. In Lua, the input table mark is changed.

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.

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]
```

title: dialog title
message: text message contents

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

Browse for Example Files

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.

Deprecated: Use IupGetParam instead.

Creation and Show

```
int IupScanf(const char *format, ...); [in C]
iup.Scanf(format: string, ...) -> (...) [in Lua]
```

format: Reading format
...: List of variables

Returns: In C the number of successfully read fields, or -1 when the user has canceled the operation. In Lua, the code is not returned, the values are returned by the function in the same order they were passed, or nil when the user has canceled the operation.

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 text field in a label.
t is the maximum number of characters allowed
v is the number of visible characters in the text 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)

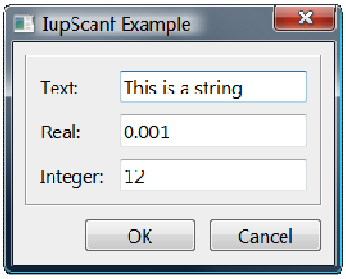
All the fields use a text box for input. If you need better control of what characters the user enters, you should use IupGetParam. This other dialog also has many other resources not available in **IupScanf**.

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

Controls

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

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

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

Controls are automatically destroyed when the dialog is destroyed.

IupButton

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

ALIGNMENT (non inheritable): horizontal and vertical alignment. Possible values: "ALEFT", "ACENTER" and "ARIGHT", combined to "ATOP", "ACENTER" and "ABOTTOM". Default: "ACENTER:ACENTER". Partial values are also accepted, like "ARIGHT" or ":ATOP", the other value will be used from the current

alignment. In Motif, vertical alignment is restricted to "ACENTER". (since 3.0)

BGCOLOR: Background color. BGCOLOR is ignored always when FLAT=YES because it will inherit from the native parent. If text and image are not defined, the BGCOLOR is used to show a color not necessary as a background color of the button, in this case set the button size because the natural size will be very small. In Windows, when using the Windows driver or the GTK driver, the BGCOLOR attribute is ignored if text or image is defined. Default: the global attribute DLGBGCOLOR.

CANFOCUS (creation only) (non inheritable): enables the focus traversal of the control. Default: YES. (since 3.0)

FLAT (creation only): Hides the button borders until the mouse enter the button area. Can be YES or NO.

FOCUSONCLICK (creation only): if NO the button will not receive the focus when clicked. Default: YES. Useful for toolbar buttons.

FGCOLOR: Text color. Default: the global attribute DLGFGCOLOR.

IMAGE (non inheritable): Image name. If set before map defines the behavior of the button to contain an image. The natural size will be size of the image in pixels, plus the button borders. Use IupSetHandle or IupSetAttributeHandle to associate an image to a name. See also IupImage. If TITLE is also defined both will be shown (except in Motif). (GTK 2.6)

IMINACTIVE (non inheritable): Image name of the element when inactive. If it is not defined then the IMAGE is used and the colors will be replaced by a modified version of the background color creating the disabled effect. GTK will also change the inactive image to look like other inactive objects. (GTK 2.6)

IMPRESS (non inheritable): Image name of the pressed button. If IMPRESS and IMAGE are defined, the button borders are not shown, but the border size is always included in size computation. When the button has the focus, in Motif and GTK a small border is displayed. When the button is clicked the pressed image does not offset, except in GTK. (GTK 2.6)

IMPRESSBORDER (non inheritable): if enabled the button borders will be shown even if IMPRESS is defined. Can be "YES" or "NO". Default: "NO".

IMAGEPOSITION (non inheritable): Position of the image relative to the text when both are displayed. Can be: LEFT, RIGHT, TOP, BOTTOM. Default: LEFT. (since 3.0) (GTK 2.10)

MARKUP [GTK only]: allows the title string to contains pango markup commands. Works only if a mnemonic is NOT defined in the title. Can be "YES" or "NO". Default: "NO".

PADDING: internal margin. Works just like the MARGIN attribute of the **IupHbox** and **IupVbox** containers, but uses a different name to avoid inheritance problems. Default value: "0x0". (since 3.0)

SPACING (creation only): defines the spacing between the image associated and the button's text. Default: "2".

TITLE (non inheritable): Button's text. If IMAGE is not defined before map, then the default behavior is to contain a text. The button behavior can not be changed after map. The natural size will be larger enough to include all the text in the selected font, even using multiple lines, plus the button borders. The '\n' character is accepted for line change. The "&" character can be used to define a mnemonic, the next character will be used as key. Use "&&" to show the "&" character instead on defining a mnemonic. The button can be activated from any control in the dialog using the "Alt+key" combination. (mnemonic support since 3.0)

ACTIVE, FONT, EXPAND, X, Y, POSITION, MINSIZE, MAXSIZE, WID, TIP, SIZE, RASTERSIZE, ZORDER, VISIBLE: also accepted.

Callbacks

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

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

ih: identifier of the element that activated the event.

Returns: IUP_CLOSE will be processed.

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

MAP_CB, UNMAP_CB, GETFOCUS_CB, KILLFOCUS_CB, ENTERWINDOW_CB, LEAVEWINDOW_CB, K_ANY, HELP_CB: All common callbacks are supported.

Notes

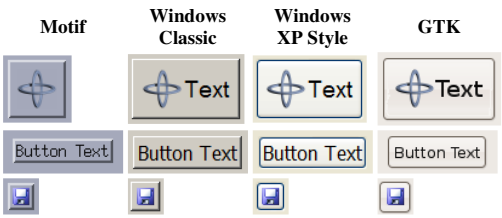
Buttons with images and/or texts can not change its behavior after mapped. This is a creation dependency. But after creation the image can be changed for another image, and the text for another text.

Buttons are activated using Enter or Space keys.

Examples

Browse for Example Files

The buttons with image and text simultaneous have PADDING=5x5, the other buttons have no padding.



See Also

IupImage, IupToggle, IupButton

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

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

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. In GTK or Motif if you set the ACTION callback after map then you should also set BGCOLOR to any value just after setting the callback or the first redraw will be lost. Default: "255 255 255".

BORDER (creation only): Shows a border around the canvas. Default: "YES".

CANFOCUS (creation only) (non inheritable): enables the focus traversal of the control. Default: YES. (since 3.0)

CLIPRECT [Windows and GTK Only] (only during ACTION): Specifies a rectangle that has its region invalidated for painting, it could be used for clipping. Format: "%d %d %d %d"="x1 y1 x2 y2".

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

EXPAND (non inheritable): The default value is "YES". The natural size is the size of 1 character.

DRAGDROP [Windows and GTK Only] (non inheritable): 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 (non inheritable): 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.

HDC_WMPAINT [Windows Only] (non inheritable): Contains the HDC created with the BeginPaint inside the WM_PAINT message. Valid only during the ACTION callback.

HWND [Windows Only] (non inheritable, read-only): Returns the Windows Window handle. Available in the Windows driver or in the GTK driver in Windows.

SCROLLBAR (creation only): Associates a horizontal and/or vertical scrollbar to the canvas. Default: "NO". The secondary attributes are all non inheritable.

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".
XMIN: Minimum value of the horizontal scrollbar. Default: "0.0".
XMAX: Maximum 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)

XDISPLAY [UNIX Only](non inheritable, read-only): Returns the X-Windows Display. Available in the Motif driver or in the GTK driver in UNIX.

XWINDOW [UNIX Only](non inheritable, read-only): Returns the X-Windows Window (Drawable). Available in the Motif driver or in the GTK driver in UNIX.

ACTIVE, FONT, X, Y, POSITION, MINSIZE, MAXSIZE, 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.

DROPFILES_CB [Windows and GTK Only]: Action generated when one or more files are dropped in the element.

FOCUS_CB: Called when the canvas gets or looses 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 loosing 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.

When the canvas has the focus, pressing the arrow keys may change the focus to another control in some systems. If your callback process the arrow keys, we recommend you to return IUP_IGNORE so it will not lose its focus.

RESIZE_CB: Action generated when the canvas size is changed.

SCROLL_CB: Called when the scrollbar is manipulated. (GTK 2.8) Also the POSX and POSY values will not be correctly updated for older GTK versions.

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, UNMAP_CB, GETFOCUS_CB, KILLFOCUS_CB, ENTERWINDOW_CB, LEAVEWINDOW_CB, K_ANY, HELP_CB: All common callbacks are supported.

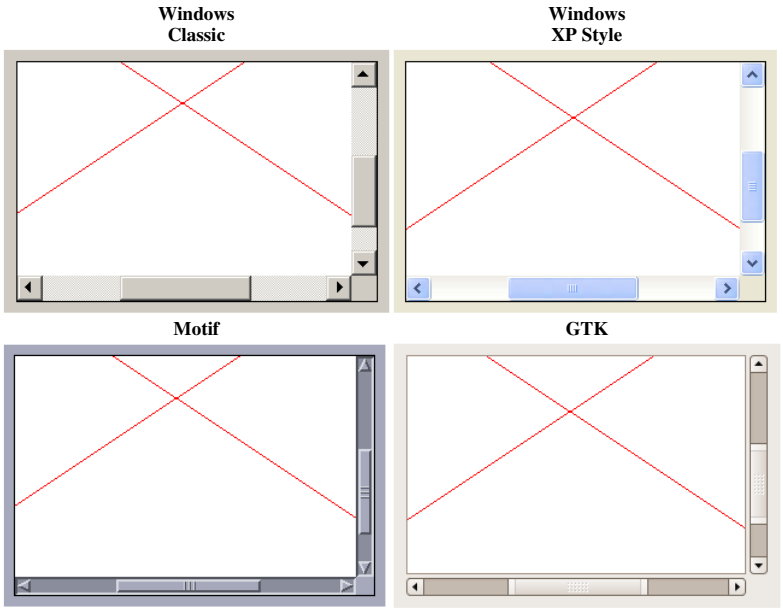
Notes

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 for Example Files



SCROLLBAR (creation only)

Associates a horizontal and/or vertical scrollbar to the element.

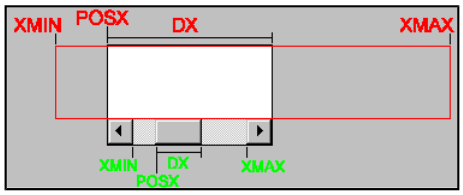
Value

"VERTICAL", "HORIZONTAL", "YES" (both) or "NO" (none).

Default: "NO"

Notes

The scrollbar allows you to create a virtual space associated to the element. In the image below, such space is marked in **red**, as well as the attributes that affect the composition of this space. In **green** you can see how these attributes are reflected on the scrollbar.



Hence you can clearly deduce that POSX is limited to XMIN and XMAX-DX, or **XMIN<=POSX<=XMAX-DX**. When the virtual space has the same size as the canvas, DX equals XMAX-XMIN, the scrollbar can be automatically hidden. See the attribute XAUTOHIDE.

The same is valid for YMIN, YMAX, DY and POSY. But remember that the Y axis is oriented from top to bottom in IUP. So if you want to consider YMIN and YMAX as bottom-up oriented, then the actual YPOS must be obtained using **YMAX-DY-POSY**.

If you have to change the properties of the scrollbar (XMIN, XMAX and DX) but you want to keep the thumb still (if possible) in the same relative position, then you have to also recalculate its position (POSX) using the old position as reference to the new one. For example, you can convert it to a 0-1 interval and then scale to the new limits:

```
old_posx_relative = (old_posx - old_xmin)/(old_xmax - old_xmin)
posx = (xmax - xmin)*old_posx_relative + xmin
```

IupList and **IupMultiline** scrollbars are automatically managed and do NOT have the POS, MIN, MAX and D attributes.

Affects

IupList, IupMultiline, IupCanvas, POSX, XMIN, XMAX, DX, XAUTOHIDE, POSY, YMIN, YMAX, DY, YAUTOHIDE.

DX

Size of the horizontal scrollbar's thumbnail in any unit.

Value

Any floating-point value greater than zero and smaller than the difference between XMAX and XMIN.

Default:: "0.1".

Notes

LINEX, XMAX and XMIN are only updated in the scrollbar when DX is updated.

When the canvas is visible, a change in DX can generate a redraw in the horizontal scrollbar on the screen.

A change in these values can affect the attribute POSX.

Affects

IupCanvas

See Also

SCROLLBAR

DY

Size of the vertical scrollbar's thumbnail in any unit.

Value

Any floating-point value greater than zero and smaller than the difference between YMAX and YMIN.

Default:: "0.1".

Notes

LINEY, YMAX and YMIN are only updated in the scrollbar when DY is updated.

When the canvas is visible, a change in DY can generate a redraw in the horizontal scrollbar on the screen.

A change in these values can affect the attribute POSY.

Affects

IupCanvas

See Also

SCROLLBAR

POSX

Thumbnail position in the horizontal scrollbar in any unit.

Value

Any floating-point value. Must be a value between XMIN and XMAX-DX.

Default: "0.0"

Notes

When the canvas is visible, a change in POSX can generate a redraw in the horizontal scrollbar on the screen.

This attribute does not generate a redraw of the canvas. If you need a redraw call **IupUpdate**.

Affects

IupCanvas

See Also

SCROLLBAR

POSY

Thumbnail position in the vertical scrollbar in any unit.

Value

Any floating-point value. Must be a value between YMIN and YMAX-DY.

Default: "0.0"

Notes

When the canvas is visible, a change in POSY can generate a redraw in the vertical scrollbar on the screen.

This attribute does not generate a redraw of the canvas. If you need a redraw call **IupUpdate**.

Affects

IupCanvas

See Also

SCROLLBAR

XMIN

Minimum value of the horizontal scrollbar, in any unit.

Value

Any floating-point value.

Default: "0.0"

Notes

A change in this value will only be effective after the attribute DX is changed.

Affects

IupCanvas

See Also

SCROLLBAR

XMAX

Maximum value of the horizontal scrollbar, in any unit.

Value

Any floating-point value.

Default: "1.0"

Notes

A change in this value will only be effective after the attribute DX is changed.

Affects

IupCanvas

See Also

SCROLLBAR

YMIN

Minimum value of the vertical scrollbar, in any unit.

Value

Any floating-point value.

Default: "0.0"

Notes

A change in this value will only be effective after the attribute DY is changed.

Affects

IupCanvas

See Also

SCROLLBAR

YMAX

Maximum value of the vertical scrollbar, in any unit.

Value

Any floating-point value.

Default: "1.0"

Notes

A change in this value will only be effective after the attribute DY is changed.

Affects

IupCanvas

See Also

SCROLLBAR

BUTTON_CB

Action generated when a mouse button is pressed or released.

Callback

```
int function(Ihandle* ih, int button, int pressed, int x, int y, char* status); [in C]
elem:button_cb(but, pressed, x, y: number, status: string) -> (ret: number) [in Lua]
```

ih: identifies the element that activated the event.
button: identifies the activated mouse button:

IUP_BUTTON1 - left mouse button (button 1);
IUP_BUTTON2 - middle mouse button (button 2);
IUP_BUTTON3 - right mouse button (button 3).

pressed: boolean that indicates the state of the button:

0 - mouse button was released;
1 - mouse button was pressed.

x, **y**: position in the canvas where the event has occurred, in pixels.
status: status of the mouse buttons and some keyboard keys at the moment the event is generated. The following macros must be used for verification:

iup_isshift(status)
iup_iscontrol(status)
iup_isbutton1(status)
iup_isbutton2(status)
iup_isbutton3(status)
iup_isbutton4(status)
iup_isbutton5(status)
iup_isdouble(status)
iup_isalt(status)
iup_issys(status)

They return 1 if the respective key or button is pressed, and 0 otherwise. These macros are also available in Lua, returning a boolean.

Returns: IUP_CLOSE will be processed. On some controls if IUP_IGNORE is returned the action is ignored (this is system dependent).

Notes

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

In general a double click, is preceded by a single click/unclick calls. In GTK, the double click is preceded by an additional single click call.

Affects

IupCanvas, IupButton, IupText, IupList, IupGLCanvas

MOTION_CB

Action generated when the mouse moves.

Callback

```
int function(Ihandle *ih, int x, int y, char *status); [in C]
elem:motion_cb(x, y: number, status: string) -> (ret: number) [in Lua]
```

ih: identifier of the element that activated the event.
x, **y**: position in the canvas where the event has occurred, in pixels.
status: status of mouse buttons and certain keyboard keys at the moment the event was generated. The same macros used for BUTTON_CB can be used for this status.

Affects

IupCanvas, IupGLCanvas

KEYPRESS_CB

Action generated when a key is pressed or released. If the key is pressed and held several calls will occur. It is called after the callback **K_ANY** is processed.

Callback

```
int function(Ihandle *ih, int c, int press); [in C]
elem:keypress_cb(c, press: number) -> (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.
press: 1 is the user pressed the key or 0 otherwise.

Returns: If IUP_IGNORE is returned the key is ignored by the system. IUP_CLOSE will be processed.

Affects

IupCanvas

SCROLL_CB

Called when some manipulation is made to the scrollbar. The canvas is automatically redrawn only if this callback is NOT defined.

(GTK 2.8)

Callback

```
int function(Ihandle *ih, int op, float posx, float posy); [in C]
elem:scroll_cb(op, posx, posy: number) -> (ret: number) [in Lua]
```

ih: identifier of the element that activated the event.
op: indicates the operation performed on the scrollbar.

If the manipulation was made on the vertical scrollbar, it can have the following values:

```
IUP_SBUP - line up
IUP_SBDN - line down
IUP_SBPGUP - page up
IUP_SBPGDN - page down
IUP_SBPOSV - vertical positioning
IUP_SBDRAGV - vertical drag
```

If it was on the horizontal scrollbar, the following values are valid:

```
IUP_SBLEFT - column left
IUP_SBRIGHT - column right
IUP_SBPGLEFT - page left
IUP_SBPGRIGHT - page right
IUP_SBPOSH - horizontal positioning
IUP_SBDRAGH - horizontal drag
```

posx, **posy**: the same as the **ACTION** canvas callback (corresponding to the values of attributes POSX and POSY).

Notes

IUP_SBDRAGH and IUP_SBDRAGV are not supported in GTK.

Affects

IupCanvas, IupGLCanvas, SCROLLBAR

WHEEL_CB

Action generated when the mouse wheel is rotated. If this callback is not defined the wheel will automatically scroll the canvas in the vertical direction by some lines, the SCROLL_CB callback if defined will be called with the IUP_SBDRAGV operation.

Callback

```
int function(Ihandle *ih, float delta, int x, int y, char *status); [in C]
elem:wheel_cb(delta, x, y: number, status: string) -> (ret: number) [in Lua]
```

ih: identifier of the element that activated the event.
delta: the amount the wheel was rotated in notches.
x, **y**: position in the canvas where the event has occurred, in pixels.
status: status of mouse buttons and certain keyboard keys at the moment the event was generated. The same macros used for BUTTON_CB can be used for this status.

Notes

In Motif and GTK delta is always 1 or -1. In Windows is some situations delta can reach the value of two. In the future with more precise wheels this increment can be changed.

The wheel will only work if the focus is at the canvas.

Affects

IupCanvas, IupGLCanvas

WOM_CB

Action generated when an audio device receives an event.

[Windows Only]

Callback

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

ih: identifies the element that activated the event.
state: can be opening=1, done=0, or closing=-1.

Notes

This callback is used to syncronize video playback with audio. It is sent when the audio device:

Message	Description
opening	is opened by using the **waveOutOpen** function.
done	is finished with a data block sent by using the **waveOutWrite** function.
closing	is closed by using the **waveOutClose** function.

You must use the HWND attribute when calling **waveOutOpen** in the *dwCallback* parameter and set *fdwOpen* to CALLBACK_WINDOW.

Affects

IupDialog, IupCanvas, IupGLCanvas

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

BGCOLOR: ignored, transparent in all systems. Will use the background color of the native parent. Except if TITLE is not defined and BGCOLOR is defined before map (can be changed later), then the frame will have a color background.

EXPAND (non inheritable): The default value is "YES".

FGCOLOR: Text title color. Not available in Windows when using Windows XP Visual Style. Default: the global attribute DLGFGCOLOR.

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

TITLE (non inheritable): Text the user will see at the top of the frame. If not defined during creation it can not be added lately, to be changed it must be at least "" during creation.

ACTIVE, FONT, X, Y, POSITION, CLIENTSIZE, MINSIZE, MAXSIZE, WID, SIZE, RASTERSIZE, ZORDER, VISIBLE: also accepted.

Callbacks

MAP_CB, UNMAP_CB: common callbacks are supported.

Notes

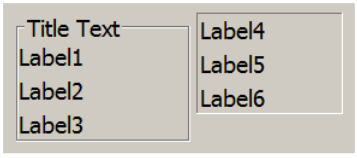
In Windows, a Frame with TITLE==NULL is not the same control as then TITLE!=NULL. When TITLE==NULL it does not have Visual Styles and uses a sharp rectangle border. When TITLE!=NULL it has Visual Styles and the border is a rounded rectangle. To always use Visual Styles set the title to "" before mapping, but be aware that a vertical space for the title will be always reserved at the top border.

Examples

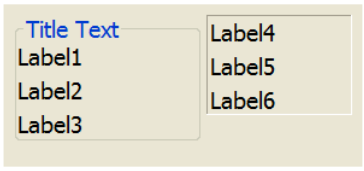
Browse for Example Files

```
frame1 = IupFrame
         (
           IupVbox
           (
             IupLabel("Label1"),
             IupLabel("Label2"),
             IupLabel("Label3"),
             NULL
           )
         );

frame2 = IupFrame
         (
           IupVbox
           (
             IupLabel("Label4"),
             IupLabel("Label5"),
             IupLabel("Label6"),
             NULL
           )
         );

IupSetAttribute(frame1, "TITLE", "Title Text");
IupSetAttribute(frame2, "SUNKEN", "YES");
```

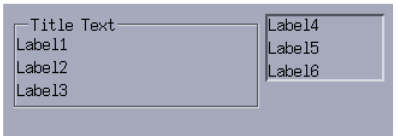
Windows 2000



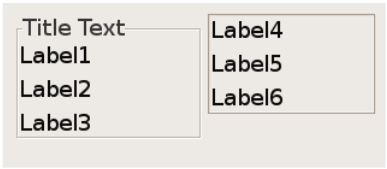
Windows XP



Motif



GTK



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: The only difference between an active label and an inactive one is its visual feedback. Possible values: "YES, "NO". Default: "YES".

ALIGNMENT (non inheritable): 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 ":ATOP", the other value will be used from the current alignment. In Motif, vertical alignment is restricted to "ACENTER". (vertical alignment since 3.0)

BGCOLOR: ignored, transparent in all systems. Will use the background color of the native parent.

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) (GTK 2.6)

FGCOLOR: Text color. Default: the global attribute DLGFGCOLOR.

IMAGE (non inheritable): Image name. 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. Use IupSetHandle or IupSetAttributeHandle to associate an image to a name. See also IupImage.

IMINACTIVE (non inheritable) [GTK and Motif only]: Image name of the element when inactive. If it is not defined then the IMAGE is used and the colors will be replaced by a modified version of the background color creating the disabled effect. GTK will also change the inactive image to look like other inactive objects.

MARKUP [GTK only]: allows the title string to contains pango markup commands. Works only if a mnemonic is NOT defined in the title. Can be "YES" or "NO". Default: "NO".

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 natural size will be larger enough to include all the text in the selected font, even using multiple lines. The '\n' character is accepted for line change. The "&" character can be used to define a mnemonic, the next character will be used as key. Use "&&" to show the "&" character instead of defining a mnemonic. The next control from the label will be activated from any control in the dialog using the "Alt+key" combination. (mnemonic support since 3.0)

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, MINSIZE, MAXSIZE, WID, TIP, SIZE, RASTERSIZE, ZORDER, VISIBLE: also accepted.

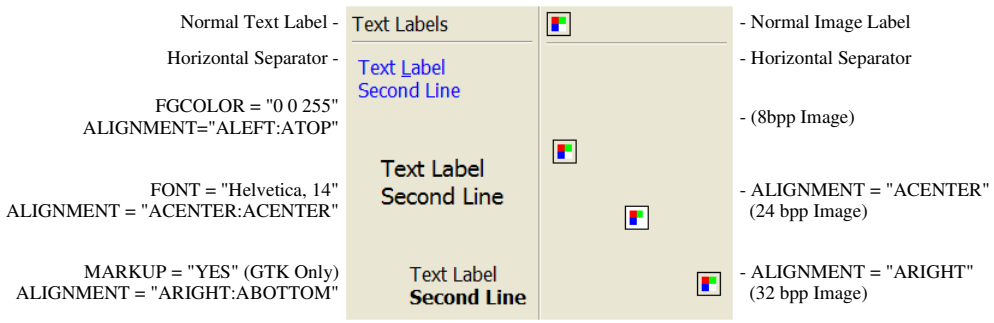
Callbacks

MAP_CB, UNMAP_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



Browse for Example Files

See Also

IupImage, IupButton.

IupList

Creates an interface element that displays a list of items. The list can be visible or can be dropped down. It also can have an edit box for text input. So it is a 4 in 1 element. In native systems the dropped down case is called Combo Box.

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": n^{th} item in the list.

(non inheritable) The values can be any text. Items before "1" are ignored. Before map the first item with a NULL is considered the end of the list and items can be set in any order. After map, there are a few rules:

- if "1" is set to NULL, all items are removed.
- if "n" is set to NULL, all items after n are removed.
- if "n" is between the first and the last item, the current n^{th} item is replaced. The effect is the same as removing the old item and inserting a new one at the old position.
- if "n+1" is set then it is appended after the last item.
- Items after "n+1" are ignored. (since 3.0)

APPENDITEM (write-only): inserts an item after the last item. Ignored if set before map. (since 3.0)

AUTOHIDE: scrollbars are shown only if they are necessary. Default: "YES".

BGCOLOR: Background color of the text. Default: the global attribute TXTBGCOLOR.

CANFOCUS (creation only) (non inheritable): enables the focus traversal of the control. Default: YES. (since 3.0)

COUNT (read-only) (non inheritable): returns the number of items. Before mapping it counts the number of non NULL items before the first NULL item. (since 3.0)

DRAGDROP [Windows and GTK Only] (non inheritable): 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. (since 3.0)

DROPDOWN (creation only): 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. To select another option, the user must press this button, which displays all items in the list. Can be "YES" or "NO". Default "NO".

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".

EDITBOX (creation only): Adds an edit box to the list. Can be "YES" or "NO". Default "NO".

FGCOLOR: Text color. Default: the global attribute TXTFGCOLOR.

INSERTITEMn (write-only): inserts an item before the given n position. n starts at 1. Ignored if out of bounds. Ignored if set before map. (since 3.0)

MULTIPLE (creation only): Allows selecting several items simultaneously (multiple list). Default: "NO". Only valid when EDITBOX=NO and DROPDOWN=NO.

REMOVEITEM (write-only): removes the given value. value starts at 1. If value is NULL removes all the items. Ignored if set before map. (since 3.0)

SCROLLBAR (creation only): Associates automatic scrollbars to the list when DROPDOWN=NO. Can be: "YES" or "NO" (none). Default: "YES". For all systems, when SCROLLBAR=YES the natural size will always include its size even if the native system hides the scrollbars. If **AUTOHIDE**=YES scrollbars are shown only if they are necessary, by default AUTOHIDE=YES. In Motif, SCROLLBAR=NO is not supported and if EDITBOX=YES the horizontal scrollbar is never shown.

When DROPDOWN=YES the scrollbars are system dependent, and do NOT depend on the SCROLLBAR or AUTOHIDE attributes. Usually the scrollbars are shown if necessary. In GTK, scrollbars are never shown and all items are always visible. In Motif, the horizontal scrollbar is never shown. In Windows, if DROPEXPAND=YES then the horizontal scrollbar is never shown.

SHOWDROPDOWN (write-only): opens or closes the dropdown list. Can be "YES" or "NO". Valid only when DROPDOWN=YES. Ignored if set before map.

SIZE: Size of the list. The **Natural Size** is defined by the number of elements in the list and the with of the largest item, the default has room for 5 characters in 1 item. In IUP 3, the **Natural Size** ignores the list contents if VISIBLECOLUMNS or VISIBLELINES attributes are defined. The text in the edit box is ignored when considering the list contents.

SORT (creation only): force the list to be alphabetically sorted. When using INSERTITEMn or APPENDITEM the position will be ignored. (since 3.0)

TOPITEM (write-only): position the given item at the top of the list or near to make it visible. Valid only when DROPDOWN=NO. (since 3.0)

SPACING: internal padding for each item. Notice that vertically the distance between each item will be actually 2x the spacing. It also affects the horizontal margin of the item. In Windows, the text is aligned at the top left of the item always. Valid only when DROPDOWN=NO. (since 3.0)

VALUE (non inheritable): Depends on the DROPDOWN+EDITBOX combination:

- EDITBOX=YES: Text entered by the user.
- DROPDOWN=YES or MULTIPLE=NO: Integer number representing the selected item in the list (begins at 1). It can be zero if there is no selected item. The value can be NULL for no item selected (since 3.0) (In Motif when DROPDOWN=YES there is always an item selected, except only when the list is empty).
- MULTIPLE=YES: 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.

VISIBLE_ITEMS: Number of items that are visible when DROPDOWN=YES is used for the dropdown list. Default: 5. Ignored in GTK.

VISIBLECOLUMNS: Defines the number of visible columns for the **Natural Size**, this means that will act also as minimum number of visible columns. It uses a wider character size then the one used for the SIZE attribute so strings will fit better without the need of extra columns. (since 3.0)

VISIBLELINES: When DROPDOWN=NO defines the number of visible lines for the **Natural Size**, this means that will act also as minimum number of visible lines. (since 3.0)

APPEND, CARET, CARETPOS, **CLIPBOARD**, **CUEBANNER, FILTER, INSERT, PADDING**, **MASK, NC, READONLY, SELECTEDTEXT, SELECTION, SELECTIONPOS**, **SCROLLTO**, **SCROLLTOPOS** : Same as the IupText attributes, but are valid only when EDITBOX=YES and effective only for the edit box inside the list.

ACTIVE, FONT, EXPAND, X, Y, POSITION, MINSIZE, MAXSIZE, WID, TIP, RASTERSIZE, ZORDER, VISIBLE: also accepted.

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 *ih, char *text, int pos, int state); [in C]
elem:action(text: string, pos, state: number) -> (ret: number) [in Lua]
```

ih: identifier of the element that activated the event.
text: Text of the changed item.
pos: Number of the changed item.
state: Equal to 1 if the option was selected or to 0 if the option was deselected.

The state=0 is simulated internally by IUP in all systems. If you add or remove items to/from the list and you count on the state=0 value, then after adding/removing items set the VALUE attribute to ensure proper state=0 value.

BUTTON_CB: Action generated when any mouse button is pressed or released inside the list. Called only when DROPDOWN=NO. If the list has an editbox the message is called when cursor is at the listbox only (ignored at the editbox). Use IupConvertXYToPos to convert (x,y) coordinates in item position. (since 3.0)

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

```
int function(Ihandle *ih, int lin, int col, int pos); [in C]
elem:caret_cb(lin, col, pos: number) -> (ret: number) [in Lua]
```

ih: identifier of the element that activated the event.
lin, col: line and column number (start at 1).
pos: 0 based character position.

For lists **lin** is always 1, and **pos** is always "**col**-1".

This is the same CARET_CB callback definition as for the IupText.

DBLCLICK_CB: Action generated when the user double click an item. Called only when DROPDOWN=NO. (since 3.0)

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

ih: identifier of the element that activated the event.
pos: Number of the selected item.
text: Text of the selected item.

DROPDOWN_CB: Action generated when the list of a dropdown is shown and hidden. Called only when DROPDOWN=YES. (since 3.0)

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

ih: identifier of the element that activated the event.
state: state of the list 1=shown, 0=hidden.

DROPFILES_CB [Windows and GTK Only]: Action generated when one or more files are dropped in the element. (since 3.0)

EDIT_CB: Action generated when the text in the text box is manually changed by the user, but before its value is actually updated. Valid only when EDITBOX=YES.

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

ih: identifier of the element that activated the event.
c: valid alpha numeric character or 0.
new_value: Represents the new text value.

Returns: IUP_CLOSE will be processed, but the change will be ignored. If IUP_IGNORE, the system will ignore the new value. If **c** is valid and returns a valid alpha numeric character, this new character will be used instead. The VALUE attribute can be changed only if IUP_IGNORE is returned.

This is the same ACTION callback definition as for the IupText.

MOTION_CB: Action generated when the mouse is moved over the list. Called only when DROPDOWN=NO. If the list has an editbox the message is called when cursor is at the listbox only (ignored at the editbox). Use IupConvertXYToPos to convert (x,y) coordinates in item position. (since 3.0)

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 *ih, char *value); [in C]
elem:multiselect_cb(value: string) -> (ret: number) [in Lua]
```

ih: identifier of the element that activated the event.
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. If this callback is defined the **ACTION** callback will not be called.

The non changed items marked with 'x' are simulated internally by IUP in all systems. If you add or remove items to/from the list and you count on the 'x' values, then after adding/removing items set the VALUE attribute to ensure proper 'x' values.

VALUECHANGED_CB: Called after the value was interactively changed by the user. Called when the selection is changed or when the text is edited. (since 3.0)

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

ih: identifier of the element that activated the event.

MAP_CB, UNMAP_CB, GETFOCUS_CB, KILLFOCUS_CB, ENTERWINDOW_CB, LEAVEWINDOW_CB, K_ANY, HELP_CB: All common callbacks are supported.

Notes

Text is always left aligned.

The GETFOCUS_CB and KILLFOCUS_CB callbacks behave differently depending on the list configuration and on the native system:

- If DROPDOWN=NO and EDITBOX=YES, then the list never gets the focus, the callbacks are called only when the edit box is clicked.
- In Motif if DROPDOWN=YES then when the dropdown button is clicked the list looses its focus and when the dropped list is closed the list regain the focus, also when that happen if the list looses its focus to another control the kill focus callback is not called.
- In GTK, if DROPDOWN=YES and EDITBOX=NO, both callbacks are called only when navigating with the keyboard (tip: if you need those callbacks with mouse navigation set EDITBOX=YES and READONLY=YES). Also in GTK, if DROPDOWN=YES and EDITBOX=YES then when the dropdown button is clicked the list looses its focus and it gets it back only if the edit box is clicked.

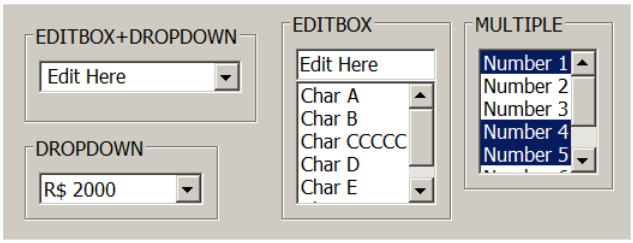
In Windows, if EDITBOX=YES then the tooltips are shown only when the cursor is near the control border. Also the selection and caret attributes are not preserved if the list loses its focus, or in other words these attributes are only useful in Windows if the list has the focus.

In GTK older than 2.12, the editbox of a dropdown will not follow the list attributes: FONT, BGCOLOR, FGCOLOR and SPACING.

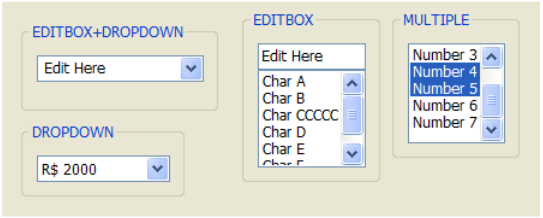
Examples

Browse for Example Files

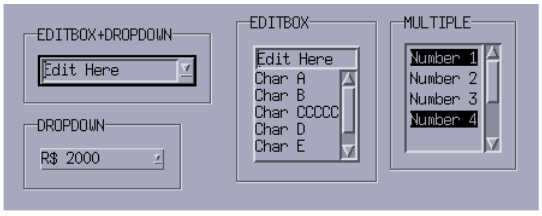
Windows 2000



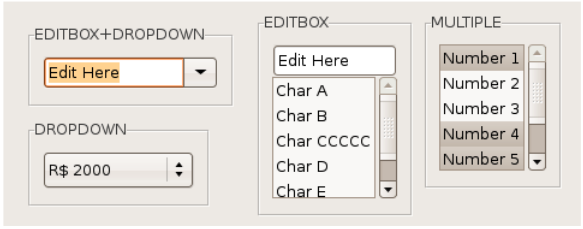
Windows XP



Motif



GTK



See Also

IupListDialog, Iuptext

IupMultiLine (deprecated since 3.0, use IupText with MULTILINE=YES)

Creates an editable field with one or more lines.

Since IUP 3.0, **IupText** has support for multiple lines when the MULTILINE attribute is set to YES. Now when a **IupMultiline** element is created in fact a **IupText** element with MULTILINE=YES is created.

See IupText

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.

Examples

Browse for Example Files

IupProgressBar (since 3.0)

Creates a progress bar control. Shows a percent value that can be updated to simulate a progression.

It is similar of **IupGauge**, but uses native controls internally. Also does not have support for text inside the bar.

Creation

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

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

Attributes

BGCOLOR [Windows Classic and Motif only]: controls the background color. Default: the global attribute DLGBGCOLOR.

DASHED (creation only in Windows) [Windows and GTK only]: Changes the style of the progress bar for a dashed pattern. Default is "NO".

FGCOLOR [Windows Classic and Motif only]: Controls the bar color. Default: the global attribute DLGFGCOLOR.

MARQUEE (creation only in Windows) [Windows XP and GTK only]: displays an undefined state. Default: NO. In GTK you still need to set VALUE using any value to update the display. In Windows the update is automatic, you can set the attribute after map but only to start or stop the animation.

MAX (non inheritable): Contains the maximum value. Default is "1". The control display is not updated, must set VALUE attribute to update.

MIN (non inheritable): Contains the minimum value. Default is "0". The control display is not updated, must set VALUE attribute to update.

ORIENTATION (creation only): can be "VERTICAL" or "HORIZONTAL". Default: "HORIZONTAL". Horizontal goes from left to right, and vertical from bottom to top.

RASTERSIZE: The initial size is defined as "200x30". Set to NULL to allow the use of smaller values in the layout computation.

VALUE (non inheritable): Contains a number between "MIN" and "MAX", controlling the current position.

ACTIVE, EXPAND, FONT, X, Y, POSITION, MINSIZE, MAXSIZE, WID, TIP, SIZE, ZORDER, VISIBLE: also accepted.

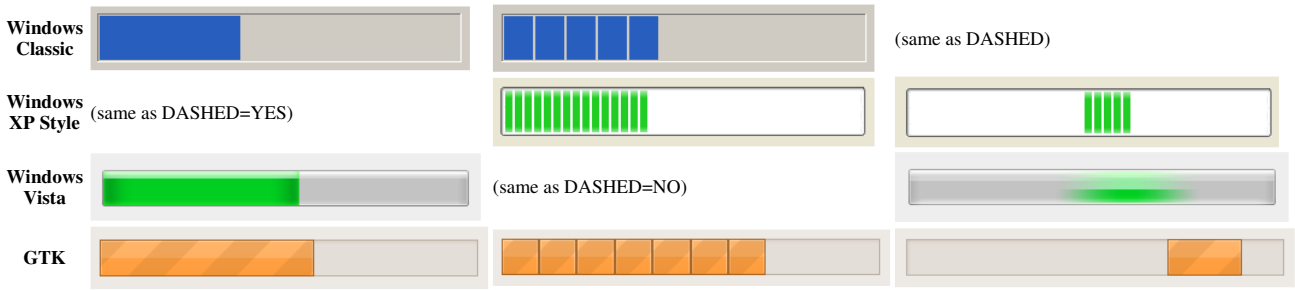
Callbacks

MAP_CB, UNMAP_CB: common callbacks are supported.

Examples

Browse for Example Files

	DASHED=NO	DASHED=YES	MARQUEE=YES
Motif		(same as DASHED=NO)	(same as DASHED=NO)



See Also

IupGauge

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.

Unlike the SPIN attribute of the **IupText** element, the **IupSpin** element can NOT automatically increment the value and it is NOT inserted inside the **IupText** area.

Creation

IupSpin inherits from a **IupVbox**, and contains two **IupButton**.

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

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

IupSpinbox inherits from a **IupHbox** and contains a **IupSpin**.

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

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

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 *ih, int inc); [in C]
elem:spin_cb(inc: number) -> (ret: number) [in Lua]
```

Examples

```
Ihandle* spinbox = IupSpinbox(IupText(NULL));
```

See Also

IupText, IupVbox, IupHbox, IupButton

IupTabs (since 3.0)

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. Also known as Notebook in native systems.

Creation

```
Ihandle* IupTabs(Ihandle* child, ...); [in C]
Ihandle* IupTabsv(Ihandle** children); [in C]
iup.tabs{child, ...: ihandle} -> (elem: ihandle) [in Lua]
tabs(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.

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

Attributes

BGCOLOR: In Windows and in GTK when in Windows, the tab buttons background it will be always defined by the system. In Windows the default background is different from the dialog background. Default: the global attribute DLGBGCOLOR.

EXPAND (non inheritable): The default value is "YES".

FGCOLOR: Tabs title color. Default: the global attribute DLGFGCOLOR.

MULTILINE [Windows Only] (non inheritable): Enable multiple lines of tab buttons. This will hide the tab scroll and fits to make all tab buttons visible. Can be "YES" or "NO". Default "NO". It is always enabled when TABTYPE=LEFT or TABTYPE=RIGHT. (since 3.0)

PADDING (non inheritable): internal margin of the tab title. Works just like the MARGIN attribute of the IupHbox and IupVbox containers, but uses a different name to avoid inheritance problems. Default value: "0x0". (since 3.0)

SIZE (non inheritable): The default size is the smallest size that fits its largest child. All child elements are considered even invisible ones.

TABIMAGEn (non inheritable): image name to be used in the respective tab. Use IupSetHandle or IupSetAttributeHandle to associate an image to a name. See also IupImage. In Motif, the image is shown only if **TABTITLEn** is NULL. In Windows and Motif set the BGCOLOR attribute before setting the image. (since 3.0)

TABORIENTATION (non inheritable): Indicates the orientation of tab text, which can be "HORIZONTAL" or "VERTICAL". Default is "HORIZONTAL". VERTICAL is supported only in GTK and in Windows. In Windows, VERTICAL is only supported when TABTYPE=LEFT or TABTYPE=RIGHT. (GTK 2.6)

TABTITLEn (non inheritable): Contains the text to be shown in the respective tab title. If this value is NULL, it will remain empty. (since 3.0)

TABTITLE (non inheritable) **(at children)**: Same as **TABTITLEn** but set in each child. Works only before the child is added to the tabs. Deprecated, use **TABTITLEn**.

TABTYPE (non inheritable) (creation only in Windows): Indicates the type of tab, which can be "TOP", "BOTTOM", "LEFT" or "RIGHT". Default is "TOP". In Windows, if LEFT or RIGHT, then MULTILINE=YES and TABORIENTATION=VERTICAL always. In Windows, when not TOP, then visual style is removed from tabs.

VALUE (non inheritable): Changes the active tab by its name. The value passed must be the name of one of the elements contained in the tabs. Use IupSetHandle or IupSetAttributeHandle to associate a child to a name. In Lua you can also use the element reference directly.

VALUE_HANDLE (non inheritable): Changes the active tab by its handle. The value passed must be the handle of a child contained in the tabs. When the tabs is created, the first element inserted is set as the visible child. (since 3.0)

VALUEPOS (non inheritable): Changes the active tab by its position, starting at 0. When the tabs is created, the first element inserted is set as the visible child. (since 3.0)

ACTIVE, FONT, X, Y, POSITION, CLIENTSIZE, MINSIZE, MAXSIZE, WID, RASTERSIZE, ZORDER, VISIBLE: also accepted.

Callbacks

TABCHANGE_CB: Callback called when the user shifts the active tab. The parameters passed are:

```
int function(Ihandle* ih, Ihandle* new_tab, Ihandle* old_tab); [in C]
elem:tabchange_cb(new_tab, old_tab: ihandle) -> (ret: number) [in Lua]
```

ih: identifier of the element that activated the event.
new_tab: the new tab selected by the user
old_tab: the previously selected tab

MAP_CB, UNMAP_CB, GETFOCUS_CB, KILLFOCUS_CB, ENTERWINDOW_CB, LEAVEWINDOW_CB, K_ANY, HELP_CB: All common callbacks are supported.

Notes

The Tabs can be created with no children and be dynamic filled using IupAppend.

The ENTERWINDOW_CB and LEAVEWINDOW_CB callbacks are called only when the mouse enter or leaves the tabs buttons area.

The Tabs children, differently from a **IupZbox**, automatically receives a name if does not already have one when it is appended to the tabs in the native system. Also **IupTabs** does NOT depends on the VISIBLE attribute.

In GTK, when the tabs buttons are scrolled, the active tab is also changed.

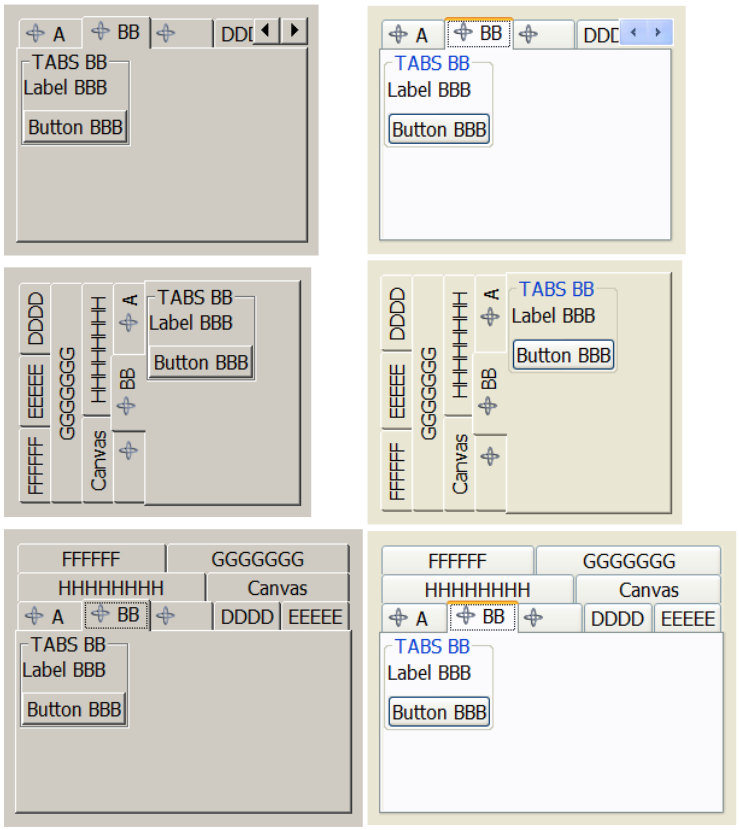
When you change the active tab the focus is usually not changed. If you want to control the focus behavior call **IupSetFocus** in the TABCHANGE_CB callback.

Examples

Browse for Example Files

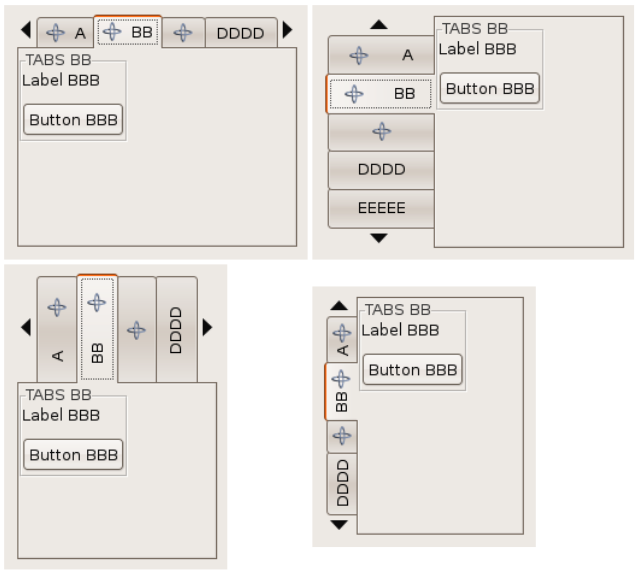
In Windows, the Visual Styles work only when TABTYPE is TOP.

Windows Classic	Windows XP Style



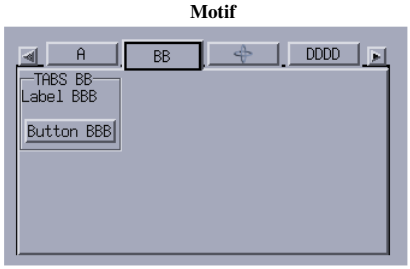
GTK is the only one that supports vertical text in the TOP configuration, but does not supports multiple lines of tab buttons.

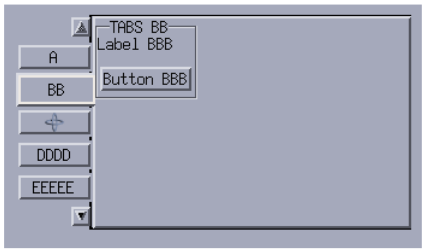
GTK



Motif does not supports vertical text.

Motif





IupText

Creates an editable text field.

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

ALIGNMENT [Windows and GTK Only] (non inheritable): text alignment. Possible values: "ALEFT", "ARIGHT", "ACENTER". Default: "ALEFT". In Motif, text is always left aligned.

APPEND (write-only): Inserts a text at the end of the current text. In the Multiline, if APPENDNEWLINE=YES, a "\n" character will be automatically inserted before the appended text (APPENDNEWLINE default is YES). Ignored if set before map.

BGCOLOR: Background color of the text. Default: the global attribute TXTBGCOLOR.

BORDER (creation only): Shows a border around the text. Default: "YES".

CANFOCUS (creation only) (non inheritable): enables the focus traversal of the control. Default: YES. (since 3.0)

CARET (non inheritable): Position of the insertion point. Its format depends in MULTILINE=YES. The first position, **lin** or **col**, is "1".

For multiple lines: a string with the "**lin**,**col**" format, where **lin** and **col** are integer numbers corresponding to the caret's position.

For single line: a string in the "**col**" format, where **col** is an integer number corresponding to the caret's position.

When **lin** is greater than the number of lines, the caret is placed at the last line. When **col** is greater than the number of characters in the given line, the caret is placed after the last character of the line.

If the caret is not visible the text is scrolled to make it visible.

In Windows, if the element does not have the focus the returned value is the position of the first character of the current selection. The caret is only displayed if the element has the keyboard focus, but its position can be changed even if not visible. When changed it will also change the selection but the text will be scrolled only when it receives the focus.

See the Notes bellow if using UTF-8 strings in GTK.

CARETPOS (non inheritable): Also the position of the insertion point, but using a zero based character unique index "pos". Useful for indexing the VALUE string. See the Notes bellow if using UTF-8 strings in GTK. (since 3.0)

CLIPBOARD (write-only): clear, cut, copy or paste the selection to or from the clipboard. Values: "CLEAR", "CUT", "COPY" or "PASTE". In Windows UNDO is also available, and REDO is available when FORMATTING=YES. (since 3.0)

CUEBANNER [Windows Only] (non inheritable): a text that is displayed when there is no text at the control. It works as a textual cue, or tip to prompt the user for input. Valid only for MULTILINE=NO, and it is not available for Windows 2000. (since 3.0)

DRAGDROP [Windows and GTK Only] (non inheritable): 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. (since 3.0)

FGCOLOR: Text color. Default: the global attribute TXTFGCOLOR.

FILTER [Windows Only] (non inheritable): allows a custom filter to process the characters: Can be LOWERCASE, UPPERCASE or NUMBER (only numbers allowed). (since 3.0)

FORMATTING [Windows and GTK Only] (non inheritable): When enabled allow the use of text formatting attributes. In GTK is always enabled, but only when MULTILINE=YES. Default: NO. (since 3.0)

INSERT (write-only): Inserts a text in the caret's position, also replaces the current selection if any. Ignored if set before map.

MASK (non inheritable): Defines a mask that will filter interactive text input.

MULTILINE (creation only) (non inheritable): allows the edition of multiple lines. In single line mode some characters are invalid, like "\t", "\r" and "\n". Default: NO.

NC: Maximum number of characters allowed for keyboard input, larger text can still be set using attributes. The maximum value is the limit of the VALUE attribute. The "0" value is the same as maximum. Default: maximum.

OVERWRITE [Windows and GTK Only] (non inheritable): turns the overwrite mode ON or OFF. Works only when FORMATTING=YES. (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. Default value: "0x0". In Windows, only the horizontal value is used. (since 3.0) (GTK 2.10 for single line)

PASSWORD (creation only) [Windows and GTK Only] (non inheritable): Hide the typed character using an "*". Default: "NO".

READONLY: Allows the user only to read the contents, without changing it. Restricts keyboard input only, text value can still be changed using attributes. Navigation keys are still available. Possible values: "YES", "NO". Default: NO.

SCROLLBAR (creation only): Valid only when MULTILINE=YES. Associates an automatic horizontal and/or vertical scrollbar to the multiline. Can be: "VERTICAL", "HORIZONTAL", "YES" (both) or "NO" (none). Default: "YES". For all systems, when SCROLLBAR!=NO the natural size will always include its size even if the native system hides the scrollbar. If **AUTOHIDE**=YES scrollbars are visible only if they are necessary, by default AUTOHIDE=NO. In Windows when FORMATTING=NO, AUTOHIDE is not supported. In Motif AUTOHIDE is not supported.

SCROLLTO (non inheritable, write only): Scroll the text to make the given position visible. It uses the same format and reference of the CARET attribute ("lin:col" or "col" starting at 1). (since 3.0)

SCROLLTOPOS (non inheritable, write only): Scroll the text to make the given position visible. It uses the same format and reference of the CARETPOS attribute ("pos" starting at 0).

SELECTEDTEXT (non inheritable): Selection text. Returns NULL if there is no selection. When changed replaces the current selection. Similar to INSERT, but does nothing if there is no selection.

SELECTION (non inheritable): Selection interval. Returns NULL if there is no selection. Its format depends in MULTILINE=YES. The first position, **lin** or **col**, is "1".

For multiple lines: a string in the "**lin1**,**col1**:**lin2**,**col2**" format, where **lin1**, **col1**, **lin2** and **col2** are integer numbers corresponding to the selection's interval. **col2** correspond to the character after the last selected character.

For single line: a string in the "**col1**:**col2**" format, where **col1** and **col2** are integer numbers corresponding to the selection's interval. **col2** correspond to the character after the last selected character.

In Windows, when changing the selection the caret position is also changed.

The values ALL and NONE are also accepted independently of MULTILINE (since 3.0).

See the Notes bellow if using UTF-8 strings in GTK.

SELECTIONPOS (non inheritable): Same as SELECTION but using a zero based character index "**pos1**:**pos2**". Useful for indexing the VALUE string. The values ALL and NONE are also accepted. See the Notes bellow if using UTF-8 strings in GTK. (since 3.0)

SIZE (non inheritable): Since the contents can be changed by the user, the **Natural Size** is not affected by the text contents (since 3.0). In IUP 2.x the **Natural Size** was defined by the number of lines in the text and the with of the largest line. For IUP 3, use VISIBLECOLUMNS and VISIBLELINES to control the **Natural Size**.

SPIN (non inheritable, creation only): enables a spin control attached to the element. Default: NO. The spin increments and decrements an integer number. The editing in the element is still available. (since 3.0)

SPINVALUE (non inheritable): the current value of the spin. The value is limited to the minimum and maximum values.
SPINMAX (non inheritable): the maximum value. Default: 100.
SPINMIN (non inheritable): the minimum value. Default: 0.
SPININC (non inheritable): the increment value. Default: 1.
SPINALIGN (creation only): the position of the spin. Can be LEFT or RIGHT. Default: RIGHT. In GTK is always RIGHT.
SPINWRAP (creation only): if the position reach a limit it continues from the opposite limit. Default: NO.
SPINAUTO (creation only): enables the automatic update of the text contents. Default: YES. Use SPINAUTO=NO and the VALUE attribute during SPIN_CB to control the text contents when the spin is incremented.

In Windows, the increment is multiplied by 5 after 2 seconds and multiplied by 20 after 5 seconds of a spin button pressed. In GTK, the increment change is progressively accelerated when a spin button is pressed.

TABSIZE [Windows and GTK Only]: Valid only when MULTILINE=YES. Controls the number of characters for a tab stop. Default: 8.

VALUE (non inheritable): Text entered by the user. The '\n' character indicates a new line, valid only when MULTILINE=YES. After the element is mapped and if there is no text will return the empty string "".

VISIBLECOLUMNS: Defines the number of visible columns for the **Natural Size**, this means that will act also as minimum number of visible columns. It uses a wider character size then the one used for the SIZE attribute so strings will fit better without the need of extra columns. As for SIZE you can set to NULL after map to use it as an initial value. Default: 5 (since 3.0)

VISIBLELINES: When MULTILINE=YES defines the number of visible lines for the **Natural Size**, this means that will act also as minimum number of visible lines. As for SIZE you can set to NULL after map to use it as an initial value. Default: 1 (since 3.0)

WORDWRAP (creation only): Valid only when MULTILINE=YES. if enabled will force a word wrap of lines that are greater than the with of the control. The horizontal scrollbar is removed. Default: NO.

ACTIVE, FONT, EXPAND, X, Y, POSITION, MINSIZE, MAXSIZE, WID, TIP, RASTERSIZE, ZORDER, VISIBLE: also accepted.

Callbacks

ACTION: Action generated when the text is edited, but before its value is actually changed. Can be generated when using the keyboard, undo system or from the clipboard.

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

ih: identifier of the element that activated the event.

c: valid alpha numeric character or 0.
new_value: Represents the new text value.

Returns: IUP_CLOSE will be processed, but the change will be ignored. If IUP_IGNORE, the system will ignore the new value. If **c** is valid and returns a valid alpha numeric character, this new character will be used instead. The VALUE attribute can be changed only if IUP_IGNORE is returned.

BUTTON_CB: Action generated when any mouse button is pressed or released. Use IupConvertXYToPos to convert (x,y) coordinates in character positioning. (since 3.0)

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

```
int function(Ihandle *ih, int lin, int col, int pos); [in C]
elem:caret_cb(lin, col, pos: number) -> (ret: number) [in Lua]
```

ih: identifier of the element that activated the event.
lin, col: line and column number (start at 1).
pos: 0 based character position.

For single line controls **lin** is always 1, and **pos** is always "**col**-1".

DROPFILES_CB [Windows and GTK Only]: Action generated when one or more files are dropped in the element. (since 3.0)

MOTION_CB: Action generated when the mouse is moved. Use IupConvertXYToPos to convert (x,y) coordinates in character positioning. (since 3.0)

SPIN_CB: Action generated when a spin button is pressed. Valid only when SPIN=YES. When this callback is called the ACTION callback is not called. The VALUE attribute can be changed during this callback only if SPINAUTO=NO. (since 3.0)

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

ih: identifier of the element that activated the event.
pos: the value of the spin (after it was incremented).

Returns: IUP_IGNORE is processed in Windows and Motif. It is ignored in GTK.

VALUECHANGED_CB: Called after the value was interactively changed by the user. (since 3.0)

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

ih: identifier of the element that activated the event.

MAP_CB, UNMAP_CB, GETFOCUS_CB, KILLFOCUS_CB, ENTERWINDOW_CB, LEAVEWINDOW_CB, K_ANY, HELP_CB: All common callbacks are supported.

Auxiliary Functions

```
void IupTextConvertLinColToPos(Ihandle* ih, int lin, int col, int *pos); [in C]
iup.TextConvertLinColToPos(ih: ihandle, lin, col: number) -> pos: number [in Lua]
```

Converts a (lin, col) character positioning into an absolute position. For single line controls **pos** is always "**col**-1". (since 3.0)

```
void IupTextConvertPosToLinCol(Ihandle* ih, int pos, int *lin, int *col); [in C]
iup.TextConvertPosToLinCol(ih: ihandle, pos: number) -> lin, col: number [in Lua]
```

Converts an absolute position into a (lin, col) character positioning. For single line controls **lin** is always 1, and **col** is always "**pos**+1". (since 3.0)

Notes

When MULTILINE=YES the Enter key will add a new line, and the Tab key will insert a Tab. So the "DEFAULTENTER" button will not be processed when the element has the keyboard focus, also to change focus to the next element press <Ctrl>+<Tab>.

In Windows, if you press a Ctrl+key combination that is not supported by the control, then a beep is sound.

When using UTF-8 strings in GTK be aware that all attributes are indexed by characters, NOT by byte index, because some characters in UTF-8 can use more than one byte. This also applies to Windows if FORMATTING=YES depending on the Windows codepage (for example East Asian codepage where some characters take two bytes).

Navigation, Selection and Clipboard Keys

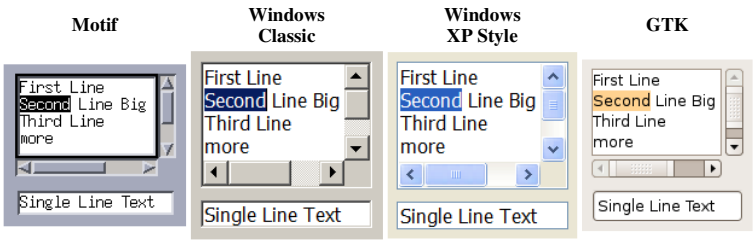
Here is a list of the common keys for all drivers. Other keys are available depending on the driver.

Keys	Action
Navigation	
Arrows	move by individual characters/lines
Ctrl+Arrows	move by words/paragraphs
Home/End	move to begin/end line
Ctrl+Home/End	move to begin/end text
PgUp/PgDn	move vertically by pages
Ctrl+PgUp/PgDn	move horizontally by pages
Selection	
Shift+Arrows	select charaters
Ctrl+A	select all
Deleting	
Del	delete the character at right
Backspace	delete the character at left

Clipboard	
Ctrl+C	copy
Ctrl+X	cut
Ctrl+V	paste

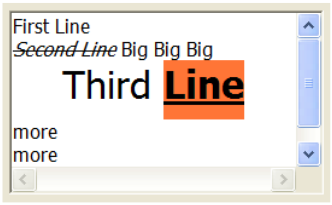
Examples

Browse for Example Files



When FORMATTING=YES in Windows or GTK (formatting attributes are set to a formatag object that it is a **IupUser**):

```
"ALIGNMENT" = "CENTER"
"SPACEAFTER" = "10"
"FONTSIZE" = "24"
"SELECTION" = "3,1:3,50"
"ADDFORMATTAG"
"BGCOLOR" = "255 128 64"
"UNDERLINE" = "SINGLE"
"WEIGHT" = "BOLD"
"SELECTION" = "3,7:3,11"
"ADDFORMATTAG"

"ITALIC" = "YES"
"STRIKEOUT" = "YES"
"SELECTION" = "2,1:2,12"
"ADDFORMATTAG"
```



When SPIN=YES:

See Also

IupList, IupMultiLine

FORMATTING [Windows and GTK Only] (non inheritable) (since 3.0)

When enabled allow the use of text formatting attributes. In GTK is always enabled, but only when MULTILINE=YES. Default: NO.

Value

Can be: YES or NO. Default: NO.

Affects

IupText

Auxiliary Attributes

ADDFORMATTAG [write only] (non inheritable)

Name of a format tag element to be added to the **IupText**. The name is associated in C using IupSetHandle. The name association must be done before setting the attribute. It will set the ADDFORMATTAG_HANDLE with the associated handle.

ADDFORMATTAG_HANDLE [write only] (non inheritable)

Handle of a format tag element to be added to the **IupText**. The tag element will be automatically destroyed when the **IupText** is mapped. If the **IupText** is already mapped, the format tag is immediately destroyed when the attribute is set. The format tag can NOT be reused.

REMOVEFORMATTING [write only] (non inheritable)

Removes the formatting of the current selection. The given value is ignored, and can be NULL.

Format Tag

The format tag element is a simple IupUser element with some known attributes that will be interpreted when the tag is updated in the native system.

The formatting depends on the existing text, so if VALUE attribute is set, all formatting is lost. You must set it again for the new text.

If the FONT attribute of the **IupText** is set then it will affect the format of all characters in the text.

The default values can not be dynamically changed.

General Format Tag Attributes

SELECTION: same as the IupText SELECTION attribute. If not defined the **IupText** attribute will be used. If the **IupText** attribute is also not defined then the current position will receive the format, so new text will be formatted with the tag. Different tags that use the same selection interval are combined.

UNITS [Windows Only]: By default all distance units are integers in pixels, but in Windows you can also specify integer units in TWIPs (one twip is 1/1440 of an inch). Can be TWIP or PIXELS. Default: PIXELS.

Paragraph Format Tag Attributes

ALIGNMENT: Can be JUSTIFY, RIGHT, CENTER and LEFT. Default: LEFT.

INDENT: paragraph indentation, the distance between the margin and the paragraph. In Windows the right indentation, and the indentation of the second and subsequent lines (relative to the indentation of the first line) can be independently set using the **INDENTRIGHT** and **INDENTOFFSET** attributes, but only when **INDENT** is set.

LINESPACING: the distance between lines of the same paragraph. In Windows, the values SINGLE, ONEHALF and DOUBLE can be used.

NUMBERING [Windows Only]: Can be BULLET (bullet symbol), ARABIC (arabic numbers - 1,2,3...), LCLETTER (lowercase letters - a,b,c...), UCLETTER (uppercase letters - A,B,C...), LCROMAN (lowercase Roman numerals - i,ii,iii...), UCROMAN (uppercase Roman numerals - I,II,III...) and NONE. Default: NONE.

NUMBERINGSTYLE [Windows Only]: Can be RIGHTPARENTESES "a)", PARENTESES "(a)", PERIOD "a.", NONUMBER (it will skip the numbering or bullet for the item) and NONE "". Default: NONE.

NUMBERINGTAB [Windows Only]: Minimum distance from a paragraph numbering or bullet to the paragraph text.

SPACEAFTER: distance left empty above the paragraph.

SPACEBEFORE: distance left empty bellow the paragraph.

TABSARRAY: a sequence of tab positions and alignment up to 32 tabs. It uses the format:"pos align pos align pos align...". Position is the distance relative to the left margin and alignment can be LEFT, CENTER, RIGHT and DECIMAL. In GTK only LEFT is currently supported. When DECIMAL alignment is used, the text is aligned according to a decimal point or period in the text, it is normally used to align numbers.

Character Format Tag Attributes

BGCOLOR: string containing a color in the format "rrr ggg bbb" for the background of the text.

DISABLED [Windows Only]: Can be YES or NO. Default NO. Set the visual appearance to disabled.

FGCOLOR: string containing a color in the format "rrr ggg bbb" for the text.

FONTSCALE: a size scale relative to the selected or current size. Values greatter than 1 will increase the font. Values smaller than 1 will shirnk the font. Default: 1.0. The following values are also accpeted: "XX-SMALL" (0.58), "X-SMALL" (0.64), "SMALL" (0.83), "MEDIUM" (1.0), "LARGE" (1.2), "X-LARGE" (1.44), "XX-LARGE" (1.73).

FONTFACE: the face name of the font.

FONTSIZE: the size of the font in pixels or points. Pixel size uses negative values.

ITALIC: Can be YES or NO. Default NO.

LANGUAGE [GTK Only]: A text with a description of the text language. The same value can be used in the "SYSTEMLANGUAGE" global attribute.

RISE: the distance, positive or negative from the base line. Can also use the values SUPERSCRIPT and SUBSCRIPT, but this values will also reduce the size of the font.

SMALLCAPS [GTK Only]: Can be YES or NO. Default NO. (Does not work always, depends on the font)

PROTECTED: Can be YES or NO. Default NO. When set to YES the selected text can NOT be edited.

STRETCH [GTK Only]: Can be EXTRA_CONDENSED, CONDENSED, SEMI_CONDENSED, NORMAL, SEMI_EXPANDED, EXPANDED and EXTRA_EXPANDED. Default NORMAL. (Does not work always, depends on the font)

STRIKEOUT: Can be YES or NO. Default NO.

UNDERLINE: Can be SINGLE, DOUBLE, DOTTED or NONE. Default NONE. DOTTED is supported only in Windows.

WEIGHT: Can be EXTRALIGHT, LIGHT, NORMAL, SEMIBOLD, BOLD, EXTRABOLD and HEAVY. Default: NORMAL.

MASK (non inheritable) (since 3.0)

Defines a mask that will filter interactive text input.

Value

string

Set to NULL to remove the mask.

Notes

Since the validation process is performed key by key when the user is typing, an intermediate value cannot be typed if it does not follow the mask rules.

Pre-Defined Masks

Definition	Value	Description
IUP_MASK_INT	"[+/-]?/d+"	integer number

IUP_MASK_UINT	"/d+"	unsigned integer number
IUP_MASK_FLOAT	"[+/-]?(/d+/.?/d*|/./d+)"	floating point number
IUP_MASK_UFLOAT	"(/d+/.?/d*|/./d+)"	unsigned floating point number
IUP_MASK_EFLOAT	"[+/-]?(/d+/.?/d*|/./d+)([eE][+/-]?/d+)?"	floating point number with exponential notation

Auxiliar Attributes

MASKCASEI (non inheritable)

If YES, will turn the filter case insensitive. Default: NO. Must be set before MASK.

MASKINT (non inheritable) (write only)

Defines an integer mask with limits. Format: "%d:%d" ("min:max"). It will replace MASK.

MASKFLOAT (non inheritable) (write only)

Defines a floating point mask with limits. Format: "%f:%f" ("min:max"). It will replace MASK.

Pattern Specification

The pattern to be searched in the text can be defined by the rules given below.

- "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" characters
[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 any 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

Affects

IupText, IupMultiline, IupList and IupMatrix

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

ALIGNMENT (non inheritable): horizontal and vertical alignment when IMAGE is defined. Possible values: "ALEFT", "ACENTER" and "ARIGHT", combined to "ATOP", "ACENTER" and "ABOTTOM". Default: "ACENTER:ACENTER". Partial values are also accepted, like "ARIGHT" or ":ATOP", the other value will be used from the current alignment. In Motif, vertical alignment is restricted to "ACENTER". In Windows works only when XP Style is active. Text is always left aligned. (since 3.0)

BGCOLOR: Background color of toggle mark when displaying a text. The text background is transparent, it will use the background color of the native parent. When displaying an image in Windows the background is ignored and the system color is used. Default: the global attribute DLGBGCOLOR.

CANFOCUS (creation only) (non inheritable): enables the focus traversal of the control. Default: YES. (since 3.0)

FGCOLOR: Color of the text shown on the toggle. In Windows, when using XP Style FGCOLOR is ignored. Default: the global attribute DLGFGCOLOR.

IMAGE (non inheritable): Image name. When the IMAGE attribute is defined, the TITLE is not shown. This makes the toggle looks just like a button with an image, but its behavior remains the same. Use IupSetHandle or IupSetAttributeHandle to associate an image to a name. See also IupImage. (GTK 2.6)

IMPRESS (non inheritable): Image name of the pressed toggle. Unlike buttons, toggles always display the button border when IMAGE and IMPRESS are both defined. (GTK 2.6)

IMINACTIVE (non inheritable): Image name of the inactive toggle. If it is not defined but IMAGE is defined then for inactive toggles the colors will be replaced by a modified version of the background color creating the disabled effect. (GTK 2.6)

MARKUP [GTK only]: allows the title string to contains pango markup commands. Works only if a mnemonic is NOT defined in the title. Can be "YES" or "NO". Default: "NO".

PADDING: internal margin when IMAGE is defined. Works just like the MARGIN attribute of the **IupHbox** and **IupVbox** containers, but uses a different name to avoid inheritance problems. Default value: "0x0". (since 3.0)

RADIO (read-only): returns if the toggle is inside a radio. Can be "YES" or "NO". Valid only after the element is mapped, before returns NULL. (since 3.0)

RIGHTBUTTON (Windows Only) (creation only): place the check button at the right of the text. Can be "YES" or "NO". Default: "NO".

VALUE (non inheritable): Toggle's state. Values can be "ON" or "OFF". If 3STATE=YES then can also be "NOTDEF". Default: "OFF". In GTK if you change the state of a radio, the unchecked toggle will receive an ACTION callback notification.

TITLE (non inheritable): Toggle's text. If IMAGE is not defined before map, then the default behavior is to contain a text. The button behavior can not be changed after map. The natural size will be larger enough to include all the text in the selected font, even using multiple lines, plus the button borders or check box if any. The '\n' character is accepted for line change. The "&" character can be used to define a mnemonic, the next character will be used as key. Use "&&" to show the "&" character instead on defining a mnemonic. The toggle can be activated from any control in the dialog using the "Alt+key" combination. (mnemonic support since 3.0)

3STATE (creation only): Enable a three state toggle. Valid for toggles with text only and that do not belong to a radio. Can be "YES" or NO". Default: "NO".

ACTIVE, FONT, EXPAND, X, Y, POSITION, MINSIZE, MAXSIZE, WID, TIP, SIZE, RASTERSIZE, ZORDER, VISIBLE: also accepted.

Callbacks

ACTION: Action generated when the toggle's state (on/off) was changed. The callback also receives the toggle's state.

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

ih: identifier of the element that activated the event.
state: 1 if the toggle's state was shifted to on; 0 if it was shifted to off.

Returns: IUP_CLOSE will be processed.

VALUECHANGED_CB: Called after the value was interactively changed by the user. Called after the ACTION callback, but under the same context. (since 3.0)

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

ih: identifier of the element that activated the event.

MAP_CB, UNMAP_CB, GETFOCUS_CB, KILLFOCUS_CB, ENTERWINDOW_CB, LEAVEWINDOW_CB, K_ANY, HELP_CB: All common callbacks are supported.

Notes

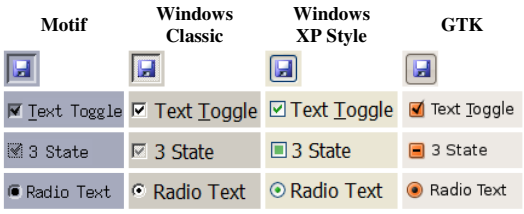
Toggle with image or text can not change its behavior after mapped. This is a creation attribute. But after creation the image can be changed for another image, and the text for another text.

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. If you need to dynamically remove toggles that belongs to a radio in Windows, then put the radio inside a **IupFrame** that has a title.

Examples

Browse for Example Files



See Also

IupImage, IupButton, IupLabel, IupRadio.

IupTree (since 3.0)

Creates a tree containing nodes of branches or leaves. Both branches and leaves can have an associated text and image.

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" actions, branches can only generate a "renamed" action.

The focus 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]
```

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

Attributes

General

ADDEXPANDED
BGCOLOR
CANFOCUS
COUNT
EXPAND
FGCOLOR
HIDELINES
HIDEBUTTONS
INDENTATION
RASTERSIZE
SHOWDRAGDROP
SPACING
TOPITEM

Nodes

CHILDCOUNT
FGCOLOR

DEPTH
KIND
PARENT
STATE
TITLE
TITLEFONT
USERDATA

Images

IMAGELEAF
IMAGEBRANCHCOLLAPSED
IMAGEBRANCHEXPANDED
IMAGEid
IMAGEEXPANDEDid

Focus Node

VALUE

Marks

MARK
MARKED
MARKEDMODE
MARKSTART

Hierarchy

ADDLEAF
ADDBRANCH
COPYNODE
DELNODE
EXPANDALL
FINDUSERDATA
INSERTLEAF
INSERTBRANCH
MOVENODE

Editing

RENAMENODE
RENAMECARET
RENAMESELECTION
SHOWRENAME

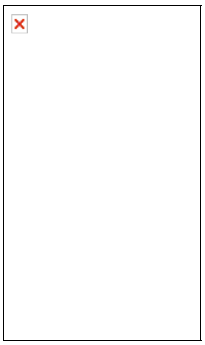
Callbacks

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.
SHOWRENAME_CB: Action generated before a node is renamed.
RENAME_CB: Action generated after a node is renamed.
DRAGDROP_CB: Action generated when a drag & drop is executed.
NODEREMOVED_CB: Action generated when a node is about to be removed.
RIGHTCLICK_CB: Action generated when the right mouse button is pressed over a node.

Notes

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. The root node has id=0 and depth=0. The tree nodes have a sequencial identification number (id), starting by the root, with id=0, and increases for each node independent from the node depth. The following picture ilustrates the numbering of the nodes in a tree.



Tree nodes and Ids

Since you have to add each node the creation of this tree can be done in several ways because the action attributes ADD* and INSERT* use an existent node to position the new node. The following pseudo code initializes the tree from top to bottom sequencially:

```
TITLE0 = "Figures"
  ADDLEAF0 = "Other"    // Use the previous node as reference
  ADDBRANCH1 = "triangle"
```

```
    ADDLEAF2 = "equilateral"
    ADDLEAF3 = "isoceles"
    ADDLEAF4 = "scalenus"
  INSERTBRANCH2 = "parallelogram"  // Use the previous node at the same depth as reference
    ADDLEAF6 = "square"
    ADDLEAF7 = "diamond"
  INSERTBRANCH6 = "2D"
  INSERTBRANCH9 = "3D"
```

The following pseudo code initializes the tree from bottom to top sequencially (except for branches), and also uses the focus node:

```
VALUE = 0  // Set the focus node at the root (default for a new element)
TITLE = "Figures"
ADDBRANCH = "3D"
ADDBRANCH = "2D"
ADDBRANCH = "parallelogram"
ADDLEAF1 = "diamond"
ADDLEAF1 = "square"
ADDBRANCH = "triangle"
ADDLEAF1 = "scalenus"
ADDLEAF1 = "isoceles"
ADDLEAF1 = "equilateral"
ADDLEAF = "Other"
```

Notice that in both cases the initialization of the tree is highly dependent on the order of the operations. Currently we can NOT garantie the order before mapping to the native system, so the initialization must be performed after the tree is mapped.

Scrollbars are automatically displayed if the tree is greatter than its display area.

Branches may be added in IupLua using a Lua Table, see iup.TreeSetValue.

Manipulation

Node insertion or removal is done by means of attributes. It is allowed to remove nodes and branches inside callbacks associated to opening or closing branches.

This means that the user may insert nodes and branches only when necessary when the parent branch is opened, allowing the use of a larger IupTree without too much overhead. Then when the parent branch is closed the subtree can be removed. But the subtree must have at least 1 node so the branch can be opened and closed, empty branches can NOT be opened.

User Data

The node Id does not always correspond to the same node as the tree is modified. For example, a id=2 will always refer to the third node in the tree. For that reason, there is also the **USERDATAid**, which allows identifying a specific node. The **USERDATAid** always refers to the same node. The **USERDATAid** may contain a user-created structure with other informations for the node.

Images

IupTree has three types of images: one associated to the leaf, one to the collapsed branch and the other to the expanded branch. Each image can be changed, both globally and individually.

The predefined images used in IupTree can be obtained by means of function IupGetHandle.The names of the predefined images are: IMGLEAF, IMGCOLLAPSED, IMGEXPANDED, IMGBLANK (blank sheet of paper) and IMGPAPER (written sheet of paper). By default:

```
"IMAGELEAF" uses "IMGLEAF"
"IMAGEBRANCHCOLLAPSED" uses "IMGCOLLAPSED"
"IMAGEBRANCHEXPANDED" uses "IMGEXPANDED"

"IMGBLANK" and "IMGPAPER" are designed for use as "IMAGELEAF"
```

Simple Marking

Is the IupTree's default operation mode (MARKMODE=SINGLE). In this mode only one node can be selected.

Multiple Marking

IupTree allows marking several nodes simultaneously using the Shift and Control keys. To use multiple marking set MARKMODE=MULTIPLE. In GTK and Motif multiple nodes can also be selected using a rubber band if SHOW_DRAGDROP=NO.

When a user keeps the Control key pressed, the individual marking mode is used. This way, the focus 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 focus 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 keys being pressed, or when the space bar is pressed together with Control.

Navigation

Using the keyboard:

- **Arrow Up/Down**: Moves the focus node to the neighbor node, according to the arrow direction.
- **Home/End**: Moves the focus node to the root/last node.
- **Page Up/Page Down**: Moves the focus node to the node one visible page above/below the focus node.
- **Enter**: If the focus 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+Arrow Up/Down**: Moves only the focus node.
- **Ctrl+Space**: Marks or unmarks the node at focus.
- **F2**: Calls the rename callback or invoke the inplace rename.
- **Esc**: cancels inplace rename.

In Motif when pressing Tab the focus goes to the next visible node, if there are no next visible node then the next control in the dialog receives the focus. In Windows and Gtk the focus simply goes directly to the next control.

Using the left mouse button:

- **Clicking a node**: Moves the focus node to the clicked node.

- **Clicking a (-/+) box**: Makes the branch to the right of the (-/+) box collapse/expand.
- **Double-clicking a node**: Moves the focus node to the clicked node. If the 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.
- **Clicking twice a node**: Calls the rename callback or invoke the inplace rename.
- **Clicking and dragging a node**: if SHOWDRAGDROP=Yes starts a drag. When mouse is released, the DRAGDROP_CB callback is called. If the callback does not exist or if it returns IUP_CONTINUE then the node is moved to the new position. If Ctrl is pressed then the node is copied instead of moved. In Motif drag is performed with the middle mouse button.

Removing a Node with "Del"

By default the Del key is not processed, but you can implement it using a simple K_ANY callback:

```
int k_any(Ihandle* ih, int c)
{
  if (c == K_DEL)
   IupSetAttribute(ih,"DELNODE","MARKED");
  return IUP_CONTINUE;
}
```

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 *ih, int id, void *userid); [in C]
iup.TreeSetUserId(ih: ihandle, id: number, userid: userdata/table) [in Lua]
```

ih: Identifier of the IupTree interacting with the user.
id: Node identifier.
userid: User pointer or Lua table to be associated with the node. Use NULL (nil) value to remove the association.

Associates an userid with a given id. If the id of the node is changed, the userid remains the same.

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.)

It is similar of setting the **USERDATAid** attribute, but with the additional feature of storing the Lua object in the registry.

```
void* IupTreeGetUserId(Ihandle *ih, int id); [in C]
iup.TreeGetUserId(ih: ihandle, id: number) -> (ret: userdata/table) [in Lua]
```

ih: Identifier of the IupTree interacting with the user.
id: Node identifier.

Returns the pointer or Lua table associated to the node or NULL if none was associated. **SetUserId** must have been called for the node with the given id.

It is similar of retrieving the **USERDATAid** attribute, but the Lua object is retrieved from the REGISTRY.

```
int IupTreeGetId(Ihandle *ih, void *userid); [in C]
iup.TreeGetId(ih: ihandle, userid: userdata/table) -> (ret: number) [in Lua]
```

ih: Identifier of the IupTree interacting with the user.
userid: Pointer or Lua table associated to the node.

Returns the id of the node that has the userid on success or -1 (nil) if not found. **SetUserId** must have been called with the same userid.

It is equivalent of retrieving the **FINDUSERDATAp** attribute.

```
iup.TreeSetValue(ih: ihandle, tree: table, [id: number]) [in Lua]
```

ih: Identifier of the IupTree interacting with the user.
tree: table of nodes.
id: optional existing node. The default is the root (0).

Initializes the tree using the given Lua table as values for the tree nodes using ADDBRANCH and ADDLEAF. For example:

```
tree = iup.tree{}
tree_init =
{
  branchname = "Figures",
  "Other",
  {
    branchname = "triangle",
    state = "COLLAPSED",
    "equilateral",
    "isoceles",
    "scalenus",
  },
  {
    branchname = "parallelogram",
    "square",
    { leafname = "diamond", color = "92 92 255", titlefont = "Courier, 14" },
  },
  { branchname = "2D" },
  { branchname = "3D" },
}

dlg = iup.dialog{tree}
dlg:map()

iup.TreeSetValue(tree, tree_init)

dlg:show()
```

Inside a table **branchname** defines a branch and its title, **leafname** defines a leaf and its title. When a node inside a branch is not a table then it is a leaf and only defines the leaf title. When **leafname** or **branchname** are used you can also define other node attributes: **color**, **state**, **titlefont**, **marked**, **image** and **imageexpanded**; without specifying the node id. You can also use **userid** to associate an userdata or table just like in **iup.TreeSetUserId**. (since 3.0)

Utility Functions

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

```
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 id value. For ex:

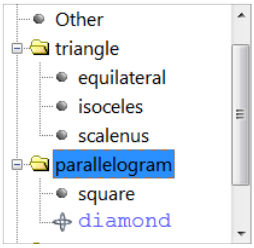
```
IupTreeSetAttribute(n, "KIND" , 30, v) = IupSetAttribute(n, "KIND30", v)
IupTreeSetAttribute(n, "ADDLEAF" , 10, v) = IupSetAttribute(n, "ADDLEAF10", v)
```

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

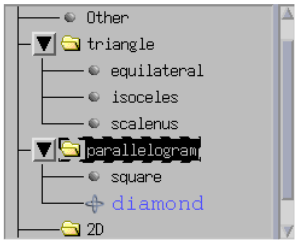
Examples

Browse for Example Files

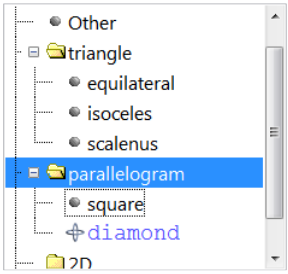
Windows



Motif



GTK



IupTree Attributes

General

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: "YES".

AUTODRAGDROP (creation ony) [GTK Only]: When SHOWDRAGDROP=Yes, it enables the automatic drag&drop inside the tree. This also means that the **DRAGDROP_CB** callback will not be called. All procesing is done internally by GTK. Default: NO. (since 3.0)

BGCOLOR: Background color of the tree. Default: the global attribute TXTBGCOLOR.

CANFOCUS (creation only) (non inheritable): enables the focus traversal of the control. Default: YES. (since 3.0)

COUNT (read only) (non inheritable): returns the total number of nodes in the tree. (since 3.0)

DRAGDROP [Windows and GTK Only] (non inheritable): 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. This is NOT related to the drag&drop of nodes inside the tree. (since 3.0)

EXPAND (non inheritable): The default value is "YES".

FGCOLOR: default text foreground color. Once each node is created it will not chage its color when FGCOLOR is changed. Default: the global attribute TXTFGCOLOR. (since 3.0)

HIDEBUTTONS (creation only): hide the expand and collapse buttons. In GTK, branches will be only expanded programmatically. In Motif it did not work and crash the test. (since 3.0) (GTK 2.12)

HIDELINES (creation only): hide the lines that connect the nodes in the hierarchy. (since 3.0) (GTK 2.10)

INDENTATION: sets the indentation level in pixels. The visual effect of changing the indentation is highly system dependent. In GTK it acts as an additional indent value, and the lines do not follow the extra indent. In Windows is limited to a minimum of 5 pixels. (since 3.0) (GTK 2.12)

RASTERSIZE (non inheritable): the initial size is "400x200". Set to NULL to allow the automatic layout use smaller values.

SHOWDRAGDROP (creation only): Enables the drag and drop of nodes, and enables the **DRAGDROP_CB** callback. Default: "NO". Works only if MARKMODE=SINGLE.

SPACING: vertical internal padding for each node. Notice that the distance between each node will be actually 2x the spacing. (since 3.0)

TOPITEM (write-only): position the given node identifier at the top of the tree or near to make it visible. If any parent node is collapsed then they are automatically expanded. (since 3.0)

ACTIVE, EXPAND, FONT, X, Y, POSITION, MINSIZE, MAXSIZE, WID, TIP, SIZE, RASTERSIZE, ZORDER, VISIBLE: also accepted.

The NAME common attribute is still not supported because of a backward compatibility code. Old applications must change the use of the old NAME attribute to TITLE, so the new NAME common attribute can be enabled in future versions.

Nodes (non inheritable)

For these attributes "id" is the specified branch identifier. If id is empty, then the focus node is used as the specified node.

CHILDCOUNTid (read only): returns the immediate children count of the specified branch. It does not count children of child that are branches. (since 3.0)

COLORid: text foreground color of the specified node. The value should be a string in the format "R G B" where R, G, B are numbers from 0 to 255.

DEPTHid (read only): returns the depth of the specified node. The root node has depth=0, its imediate children has depth=1, their children has depth=2 and so on.

KINDid (read only): returns the kind of the specified node. Possible values:

- "LEAF": The node is a leaf
- "BRANCH": The node is a branch

PARENTid (read only): returns the identifier of the specified node.

STATEid: the state of the specified branch. In Windows, it will be effective only if the branch has children. In GTK, it will be effective only if the parent is expanded. Possible values:

- "EXPANDED": Expanded branch state (shows its children)
- "COLLAPSED": Collapsed branch state (hides its children)

TITLEid: the text label of the specified node.

TITLEFONTid: the text font of the specified node. The format is the same as the FONT attribute. (since 3.0)

USERDATAid: the userdata associated with the specified node. (since 3.0)

Images (non inheritable)

IMAGEid: image name to be used in the specified node, where id is the specified node identifier. If id is empty, then the focus node is used as the specified node. Use IupSetHandle or IupSetAttributeHandle to associate an image to a name. See also IupImage. In Windows and Motif set the BGCOLOR attribute before setting the image. If node is a branch it is used when collapsed.

IMAGEEXPANDEDid: same as the IMAGE attribute but used for expanded branches.

IMAGELEAF: the image name that will be shown for all leaves. Default: "IMGLEAF". Internal values "IMGBLANK" and "IMGPAPER" are also available. If BGCOLOR is set the image is automatically updated.

IMAGEBRANCHCOLLAPSED: the image name that will be shown for all collapsed branches. Default: "IMGCOLLAPSED". If BGCOLOR is set the image is automatically updated.

IMAGEBRANCHEXPANDED: the image name that will be shown for all expanded branches. Default: "IMGEXPANDED". If BGCOLOR is set the image is automatically updated.

Focus Node

VALUE (non inheritable): The focus node identifier. When MARKMODE=SINGLE the node is also selected. The tree is always scrolled so the node becomes visible. In Motif the tree will receive the focus. Also accepts the values:

"ROOT": the root node
"LAST": the last visible node
"NEXT": the next visible node, one node after the focus node. If at the last does nothing
"PREVIOUS": the previous visible node, one node before the focus node. If at the root does nothing
"PGDN": the next visible node, ten nodes node after the focus node. If at the last does nothing
"PGUP": the previous visible node, ten nodes before the focus node. If at the root does nothing

Marks

MARK (write only) (non inheritable): Selects a range of nodes in the format "star,end" (%d-%d). Allowed only when MARKMODE=MULTIPLE. Also accepts the values:

"INVERTid": Inverts the specified node mark state, where id is the specified node identifier. If id is empty, then the focus node is used as reference node.

"BLOCK": Marks all nodes between the focus node and the initial block-marking node defined by MARKSTART
"CLEARALL": Unmark all nodes
"MARKALL": Marks all nodes
"INVERTALL": Inverts the marking of all nodes

MARKEDid (non inheritable): The marking state of the specified node, where id is the specified node identifier. If id is empty, then the focus node is used as reference node. Can be: YES or NO. Default: NO

MARKMODE: defines how the nodes can be selected. Can be: SINGLE or MULTIPLE. Default: SINGLE.

MARKSTART (non inheritable): Defines the initial node for the block marking, used when MARK=BLOCK. The value must be the node identifier. Default: 0 (root node).

Hierarchy (non inheritable)

ADDLEAFid (write only): Adds a new leaf after the reference node, where id is the reference node identifier. If id is empty, then the focus node is used as reference node. The value is used as the text label of the new node. The id of the new node will be the id of the reference node + 1. The reference node is marked and all others unmarked. The reference node position remains the same. If the reference node does not exist, nothing happens. If the reference node is a branch then the depth of the new node is one depth increment from the depth of the reference node, if the reference node is a leaf then the new node has the same depth. If you need to add a node after a specified node but at a different depth use **INSERTLEAF**. Ignored if set before map.

ADDBRANCHid (write only): Same as **ADDLEAF** for branches. Branches can be created expanded or collapsed depending on **ADDEXPANDED**. Ignored if set before map.

COPYNODEid (write only): Copies a node and its children, where id is the specified node identifier. If id is empty, then the focus node is used as the specified node. The value is the destination node identifier. If the destination node is a branch and it is expanded, then the specified node is inserted as the first child of the destination node. If the branch is not expanded or the destination node is a leaf, then it is inserted as the next brother of the leaf. The specified node is not changed. All node attributes are copied, except user data. Ignored if set before map. (since 3.0)

DELNODEid (write only): Removes a node and/or its children, where id is the specified node identifier. If id is empty, then the focus node is used as the specified node. The root cannot be removed. Ignored if set before map. Possible values:

- "SELECTED": deletes the specified node and its children
- "CHILDREN": deletes only the children of the specified node
- "MARKED": deletes all the selected nodes (and all their children), id is ignored

EXPANDALL (write only): expand or contracts all nodes. Can be YES (expand all), or NO (contract all). (since 3.0)

FINDUSERDATAp (read only): finds the id of the given userdata. "p" must be the address of a pointer formatted in the string as "%p". (since 3.0)

INSERTLEAFid, **INSERTBRANCHid** (write only): Same as **ADDLEAF** and **ADDBRACH** but the depth of the new node is always the same of the reference node. (since 3.0)

MOVENODEid (write only): Moves a node and its children, where id is the specified node identifier. If id is empty, then the focus node is used as the specified node. The value is the destination node identifier. If the destination node is a branch and it is expanded, then the specified node is inserted as the first child of the destination node. If the branch is not expanded or the destination node is a leaf, then it is inserted as the next brother of the leaf. The specified node is removed. User data and all node attributes are preserved. Ignored if set before map. (since 3.0)

Editing

RENAME (write only): Forces a rename action to take place. Valid only when SHOWRENAME=YES.

RENAMECARET: the caret's position of the text box when in-place renaming. Same as the CARET attribute for IupText.

RENAMESELECTION: the selection interval of the text box when in-place renaming. Same as the SELECTION attribute for IupText.

SHOWRENAME (creation only): Allows the in place rename of a node. Default: "NO". Since IUP 3.0, F2 and clicking twice only starts to rename a node if SHOWRENAME=Yes.

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 *ih, int id, int status) [in C]
elem:selection_cb(id, status: number) -> (ret: number) [in Lua]
```

ih: identifier of the element that activated the event.
id: Node identifier.
status: 1 - node was selected, 0 - node was unselected.

MULTISELECTION_CB: Action generated when multiple nodes are selected during the same operation (when mouse is clicked and Shift is pressed, or when a navigation key is released and shift is pressed). If not defined the SELECTION_CB will be called for all selected nodes everytime a new node is selected. The block of nodes is always completely included, independent if some node were already marked.

```
int function(Ihandle *ih, int* ids, int n) [in C]
elem:multiselection_cb(ids: table, n: number) -> (ret: number) [in Lua]
```

ih: identifier of the element that activated the event.
ids: Array of node identifiers.
n: Number of nodes in the array.

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, or hits Enter on a collapsed branch.

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

ih: identifier of the element that activated the event.

id: node identifier.

Returns: 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, or hits Enter on an expanded branch.

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

ih: identifier of the element that activated the event.
id: node identifier.

Returns: 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 a leaf, or hits Enter on a leaf.

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

ih: identifier of the element that activated the event.
id: node identifier.

SHOWRENAME_CB: Action generated when a node is about to be renamed. It occurs when the user clicks twice the node or press **F2**. Called only if SHOWRENAME=YES.

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

ih: identifier of the element that activated the event.
id: node identifier.

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

```
int function(Ihandle *ih, int id, char *title); [in C]
elem:rename_cb(id: number, title: string) -> (ret: number) [in Lua]
```

ih: identifier of the element that activated the event.
id: node identifier.
title: new node title.

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

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

```
int function(Ihandle *ih, 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]
```

ih: identifier of the element that activated the event.
drag_id: Identifier of the clicked node where the drag start.
drop_id: Identifier of the clicked node where the drop were executed.
isshift: Boolean flag indicatinf the shift key state.
iscontrol: Boolean flag indicatinf the control key state.

Returns: if returns IUP_CONTINUE, or if the callback is not defined and **SHOWDRAGDROP=YES**, then the node is moved to the new position. If Ctrl is pressed then the node is copied instead of moved. If the drop node is a branch and it is expanded, then the drag node is inserted as the first child of the node. If the branch is not expanded or the drop node is a leaf, then it is inserted as the next brother of the leaf.

NODEREMOVED_CB: Action generated when a node is going to be removed. It is only a notification, the action can not be aborted. (since 3.0)

```
int function(Ihandle *ih, int id, void* userdata); [in C]
elem:noderemoved_cb(id: number, userid: userdata/table) -> (ret: number) [in Lua]
```

ih: identifier of the element that activated the event.
id: node identifier.
userdata/userid: USERDATA attribute in C, or userid object in Lua.

RIGHTCLICK_CB: Action generated when the right mouse button is pressed over the IupTree.

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

ih: identifier of the element that activated the event.
id: node identifier.

BUTTON_CB: Action generated when any mouse button is pressed or released inside the element. Use IupConvertXYToPos to convert (x,y) coordinates in the node identifier. (since 3.0)

MOTION_CB: Action generated when the mouse is moved over the element. Use IupConvertXYToPos to convert (x,y) coordinates in item the node identifier. (since 3.0)

DROPFILES_CB [Windows and GTK Only]: Action generated when one or more files are dropped in the element. (since 3.0)

MAP_CB, UNMAP_CB, GETFOCUS_CB, KILLFOCUS_CB, ENTERWINDOW_CB, LEAVEWINDOW_CB, K_ANY, HELP_CB: All common callbacks are supported.

In Motif the tree always resets the focus to the root node when receive the focus. The KILLFOCUS_CB callback is called only when the focus is at the root node. Also in Motif some LEAVEWINDOW_CB events are delayed to when the user enter again, firing a leave and enter events at enter time.

IupVal (since 3.0)

Creates a Valuator control. Selects a value in a limited interval. Also known as Scale or Trackbar in native systems.

Creation

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

type: Type of valuator. Can be NULL.

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

Attributes

BGCOLOR: transparent in all systems except in Motif. It will use the background color of the native parent.

CANFOCUS (creation only) (non inheritable): enables the focus traversal of the control. Default: YES. (since 3.0)

INVERTED: Invert the minimum and maximum positions on screen. When INVERTED=YES maximum is at top and left (minimum is bottom and right), when INVERTED=NO maximum is at bottom and right (minimum is top and left). The initial value depends on TYPE passed as parameter on creation, if TYPE=VERTICAL default is YES, if TYPE=HORIZONTAL default is NO. (since 3.0)

MAX: Contains the maximum valuator value. Default is "1". When changed the display will not be updated until VALUE is set.

MIN: Contains the minimum valuator value. Default is "0". When changed the display will not be updated until VALUE is set.

PAGESTEP: Controls the increment for pagedown and pageup keys. It is not the size of the increment. The increment size is "pagestep*(max-min)", so it must be 0<pagestep<1. Default is "0.1".

RASTERSIZE (non inheritable): The initial size is 100 pixels along the major axis, and the handler normal size on the minor axis. If there are ticks then they are added to the natural size on the minor axis. The handler can be smaller than the normal size. Set to NULL to allow the automatic layout use smaller values.

SHOWTICKS [Windows and Motif Only]: The number of tick marks along the valuator trail. Minimum value is "2". Default is "0", in this case the ticks are not shown. It can not be changed to 0 from a non zero value, or vice-versa, after the control is mapped. GTK does not support ticks.

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)", so it must be 0<step<1. Default is "0.01".

TICKSPOS [Windows Only] (creation only): Allows to position the ticks in both sides (BOTH) or in the reverse side (REVERSE). Default: NORMAL. The normal position for horizontal orientation is at the top of the control, and for vertical orientation is at the left of the control. In Motif, the ticks position is always normal. (since 3.0)

TYPE (non inheritable): Informs whether the valuator is "VERTICAL" or "HORIZONTAL". Vertical valuators are bottom to up, and horizontal valuators are left to right variations of min to max (but can be inverted using INVERTED). Default: "HORIZONTAL".

VALUE (non inheritable): Contains a number between MIN and MAX, indicating the valuator position. Default: "0.0".

ACTIVE, EXPAND, FONT, X, Y, POSITION, MINSIZE, MAXSIZE, WID, TIP, SIZE, ZORDER, VISIBLE: also accepted.

Callbacks

VALUECHANGED_CB: Called after the value was interactively changed by the user.

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

ih: identifier of the element that activated the event.

MAP_CB, UNMAP_CB, GETFOCUS_CB, KILLFOCUS_CB, ENTERWINDOW_CB, LEAVEWINDOW_CB, K_ANY, HELP_CB: All common callbacks are supported.

Notes

This control replaces the old IupVal implemented in the additional controls. The old callbacks are still supported but called only if the VALUECHANGED_CB callback is not defined. The MOUSEMOVE_CB callback is only called when the user moves the handler using the mouse. The BUTTON_PRESS_CB callback is called only when the user press a key that changes the position of the handler. The BUTTON_RELEASE_CB callback is called only when the user release the mouse button after moving the handler (except in GTK that is never called).

In Motif, after the user clicks the handler a KILLFOCUS will be ignored when the control loses its focus.

Keyboard Mapping

This is the default mapping when INVERTED has the default value, or TYPE=HORIZONTAL+INVERTED=NO.

Keys	Action for HORIZONTAL
Right Arrow	move right, increment by one step
Left Arrow	move left, decrement by one step
Ctrl+Right Arrow or PgDn	move right, increment by one page step
Ctrl+Left Arrow or PgUp	move left, decrement by one page step
Home	move all left, set to minimum
End	move all right, set to maximum

This is the default mapping when INVERTED has the default value, or TYPE=VERTICAL+INVERTED=YES.

Keys	Action for VERTICAL
Up Arrow	move up, increment by one step
Down Arrow	move down, decrement by one step
Ctrl+Up Arrow or PgUp	move up, increment by one page step
Ctrl+Down Arrow or PgDn	move down, decrement by one page step
Home	move all up, set to maximum
End	move all down, set to minimum

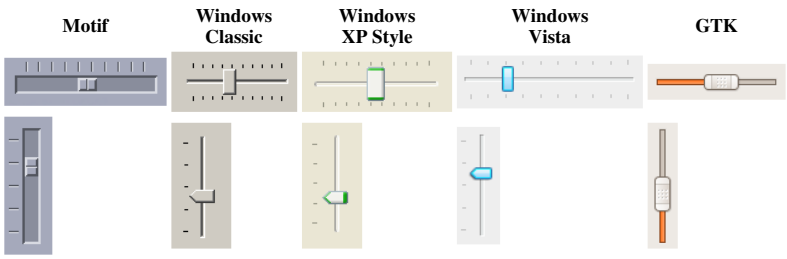
Visually all the keys move to the same direction independent from the INVERTED attribute.

Semantically all the keys change the value depending on the INVERTED attribute.

This behavior is slightly different from the defined by the native systems (Home and End keys are different). But it is the same in all systems.

Examples

Browse for Example Files



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**), the CD_IUP driver (**iupcd**), and with the CD library (**cd**). To use its bindings to Lua, the program must also be linked to the **iupluacontrols** library.

OpenGL Canvas

The drawing canvas compatible with OpenGL is called IupGLCanvas.

- 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 coluns), their appearence and interation. This mechanism offers full flexibility to applications, but requires programmers attention to avoid infinite loops inside this functions. Using callbacks, cells can be also grouped to form major or hierarchical elements, such as headers, footers etc. This callback approach was intentionally chosen to allow all cells to be dinamically 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 atributtes are used, set this attribute to "NO" to avoid grid drawing over spanned cells.

BUFFERIZE: Disables the automatic redrawing of the control, so many attributes can be changed without many redraws. When set to "NO" the control is redrawn. When REPAINT attribute is set, BUFFERIZE is automatically set to "NO". Default: "NO".

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

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

FIRST_COL (read-only) (non inheritable): Returns the number of the first visible column.

FIRST_LINE (read-only) (non inheritable): Returns the number of the first visible line.

FULL_VISIBLE (write-only) (non inheritable)**:** 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.

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

LIMITS*L:C* (read-only) (non inheritable): 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.

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) (non inheritable)**:** When set with any value, provokes the control to be redrawn.

SIZE (non inheritable): there is no initial size. You must define SIZE or RASTERSIZE.

SCROLLBAR (creation only): Default: "YES".

ACTIVE, BGCOLOR, FONT, X, Y, POSITION, MINSIZE, MAXSIZE, WID, TIP, SIZE, RASTERSIZE, ZORDER, VISIBLE: also accepted.

Callbacks

DRAW_CB: called when a specific cell needs to be redrawn.

```
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 redrawn, in grid coordinates.
xmin, xmax, ymin, ymax: the raster bounding box of the redrawn 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.

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

```
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 height (in pixels). Default is 30 pixels.

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).

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

NCOLS_CB: called when then controls 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.

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.

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.

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).

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

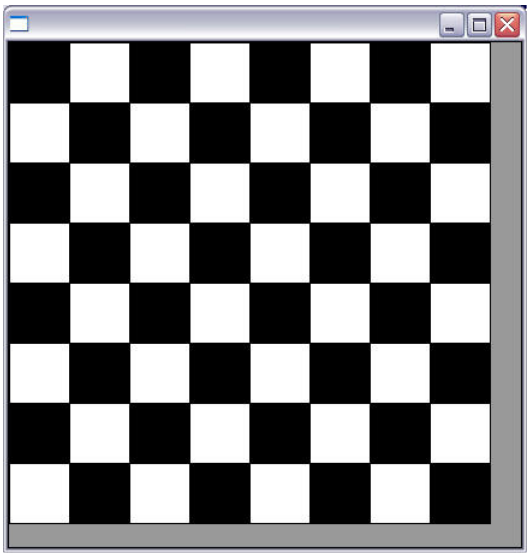
Return: an integer that specifies the desired width (in pixels). Default is 60 pixels.

MAP_CB, UNMAP_CB, GETFOCUS_CB, KILLFOCUS_CB, ENTERWINDOW_CB, LEAVEWINDOW_CB, K_ANY, HELP_CB: All common callbacks are supported.

Examples

Browse for Example Files

Checkerboard Pattern



Numbering Cells

See Also

IupCanvas

IupColorbar

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

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 AndrÃ© 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 automatic redrawing of the control, so many attributes can be changed without many redraws. Default: "NO". When set to "NO" the control is redrawn.

CELL*n*: 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".

SIZE: there is no initial size. You must define SIZE or RASTERSIZE.

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, BGCOLOR, FONT, X, Y, POSITION, MINSIZE, MAXSIZE, WID, TIP, EXPAND, 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 existent 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, UNMAP_CB, GETFOCUS_CB, KILLFOCUS_CB, ENTERWINDOW_CB, LEAVEWINDOW_CB, K_ANY, HELP_CB: All common callbacks are supported.

Notes

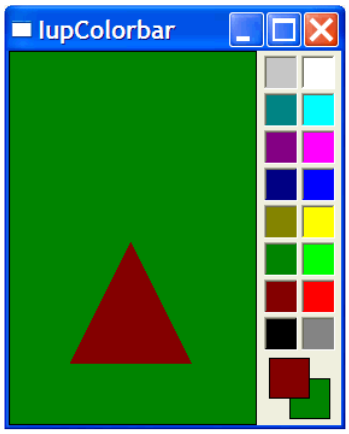
When the control has the focus the keyboard can be used to change the colors and activate the callbacks. Use the arrow keys to move from cell to cell, **Home** goes to the first cell, **End** goes to the last cell. **Space** will activate the **SELECT_CB** callback for the primary color, **Ctrl+Space** will activate the **SELECT_CB** callback for the

secondary color. **Shift+Space** will activate the **EXTENTED_CB** callback. **Shift+Enter** will activate the **CELL_CB** callback.

Examples

Browse for Example Files

Creates a Colorbar for selection of two colors.



See Also

IupCanvas, IupImage

IupColorBrowser

Creates an element for selecting a color. The selection is done using a cylindrical projection of the RGB cube. The transformation defines a coordinate color system called HSI, that is still the RGB color space but using cylindrical coordinates.

H is for Hue, and it is the angle around the RGB cube diagonal starting at red (RGB=255 0 0).

S is for Saturation, and it is the normal distance from the color to the diagonal, normalized by its maximum value at the specified Hue. This also defines a point at the diagonal used to define **I**.

I is for Intensity, and it is the distance from the point defined at the diagonal to black (RGB=0 0 0). **I** can also be seen as the projection of the color vector onto the diagonal. But **I** is not linear, see Notes bellow.

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

EXPAND: The default is "NO".

RASTERSIZE (non inheritable): the initial size is "181x181". Set to NULL to allow the automatic layout use smaller values.

RGB (non inheritable): the color selected in the control, in the “r g bâ€ format; r, g and b are integers ranging from 0 to 255. Default: "255 0 0".

HSI (non inheritable): the color selected in the control, in the â€œh s iâ€ format; h, s and i are floating point numbers ranging from 0-360, 0-1 and 0-1 respectively.

ACTIVE, BGCOLOR, FONT, X, Y, POSITION, MINSIZE, MAXSIZE, WID, TIP, SIZE, ZORDER, VISIBLE: also accepted.

Callbacks

CHANGE_CB: Called when the user releases the left mouse button over the control, defining the selected color.

```
int change(Ihandle *ih, unsigned char r, unsigned char g, unsigned char b); [in C]
elem:change_cb(r: number, g: number, b: number) -> (ret: number) [in Lua]
```

ih: identifier of the element that activated the event.
r, g, b: color value.

DRAG_CB: Called several times while the color is being changed by dragging the mouse over the control.

```
int drag(Ihandle *ih, unsigned char r, unsigned char g, unsigned char b); [in C]
elem:drag_cb(r: number, g: number, b: number) -> (ret: number) [in Lua]
```

ih: identifier of the element that activated the event.
r, g, b: color value.

VALUECHANGED_CB: Called after the value was interactively changed by the user. It is called whenever a **CHANGE_CB** or a **DRAG_CB** would also be called, it is just called after them. (since 3.0)

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

ih: identifier of the element that activated the event.

MAP_CB, UNMAP_CB, GETFOCUS_CB, KILLFOCUS_CB, ENTERWINDOW_CB, LEAVEWINDOW_CB, K_ANY, HELP_CB: All common callbacks are supported.

Notes

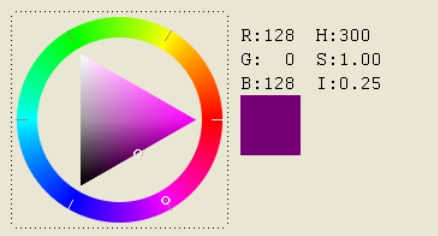
When the control has the focus the keyboard can be used to change the color value. Use the arrow keys to move the cursor inside the SI triangle, and use Home(0), PageUp, PageDn and End(180) keys to move the cursor inside the Hue circle.

The Hue in the HSI coordinate system defines a plane that it is a triangle in the RGB cube. But the maximum saturation in this triangle is different for each Hue because of the geometry of the cube. In ColorBrowser this point is fixed at the center of the **I** axis. So the **I** axis is not completely linear, it is linear in two parts, one from 0 to 0.5, and another from 0.5 to 1.0. Although the selected values are linear specified you can notice that when Hue is changed the gray scale also changes, visually compacting values above or bellow the I=0.5 line according to the selected Hue.

This is the same HSI specified in the IM toolkit, except for the non linearity of **I**. This non linearity were introduced so a simple triangle could be used to represent the SI plane.

Examples

Browse for Example Files



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

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 (non inheritable): the initial size is "16x80", "80x16" or "40x35" according to the dial type. Set to NULL to allow the automatic layout use smaller values.

TYPE (creation only) (non inheritable): dial layout configuration "VERTICAL", "HORIZONTAL" or "CIRCULAR". Default: "HORIZONTAL".

UNIT: unit of the angle. Can be "DEGREES" or "RADIANS". Default is "RADIANS". Used only in the callbacks.

VALUE (non inheritable): The dial angular value in radians. 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, BGCOLOR, FONT, X, Y, POSITION, MINSIZE, MAXSIZE, 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 converted according to UNIT.

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 converted according to UNIT.

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 converted according to UNIT.

VALUECHANGED_CB: Called after the value was interactively changed by the user. It is called whenever a **BUTTON_PRESS_CB**, a **BUTTON_RELEASE_CB** or a **MOUSEMOVE_CB** would also be called, but if defined those callbacks will not be called. (since 3.0)

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

ih: identifier of the element that activated the event.

MAP_CB, UNMAP_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.

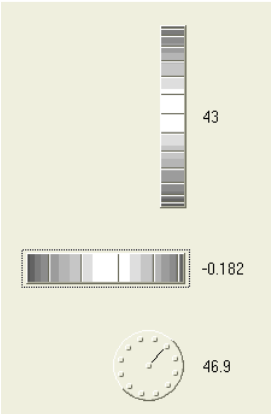
When the wheel is rotated only the mouse move callback is called, and it increments the last angle the dial was rotated.

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 for Example Files



See Also

IupCanvas

IupMatrix

Creates a matrix of alphanumeric fields. Therefore, all values of the matrixs 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 existance 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

General Attributes

- CURSOR
- FOCUS_CELL
- HIDEFOCUS
- READONLY
- RESIZEMATRIX
- USETITLESIZE

Cell Attributes

- L:C
- ALIGNMENTn
- BGCOLOR
- FGCOLOR
- FONT
- FRAMECOLOR
- VALUE

Column Attributes

- RASTERWIDTHn
- SORTSIGNn
- WIDTHn
- WIDTHDEF

Line Attributes

- HEIGHTn
- HEIGHTDEF
- RASTERHEIGHTn

Size Attributes

- NUMCOL
- NUMCOL_VISIBLE
- NUMLIN
- NUMLIN_VISIBLE

Mark Attributes

- MARKAREA
- MARKMODE
- MARK
- MARKED
- MARKMULTIPLE

Action Attributes

- ADDCOL
- ADDLIN
- DELCOL
- DELLIN
- EDIT_MODE
- ORIGIN
- SHOW
- REDRAW

Text Edition Attributes

- CARET
- MASKL:C
- MULTILINE
- SELECTION

Canvas Attributes

- BORDER

SCROLLBAR

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.
FONT_CB - Action generated to retrieve the font 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

Storage

Before mapped to the native system, all attributes are stored in the hash table, independently from the size of the matrix or its operation mode. The action attributes like ADDLIN and DELCOL will NOT work.

When the matrix is mapped NOT in callback mode then the cell values and mark state are moved from the hash table to an internal storage at the matrix. Other cell attributes remains on the hash table. Cell values with indices greater than (NUMLIN,NUMCOL) are ignored. When mapped in callback mode cell values stored in the hash table are ignored.

Size

If you do not plan to use ADDLIN nor ADDCOL, and plan to set sparse cell values, then you must set NUMLIN and NUMCOL before mapping.

If you do not plan to set SIZE or RASTERSIZE, then set NUMCOL_VISIBLE and NUMLIN_VISIBLE so the Natural size will give better results. The Natural size is calculated using only the title cells size plus the size of NUMCOL_VISIBLE and NUMLIN_VISIBLE cells, but it is also affected if SCROLBAR is enabled. The natural height is the sum of the line heights from line 0 to NUMLIN_VISIBLE (inclusive), or line 0 plus the sum from line NUMLIN-NUMLIN_VISIBLE to NUMLIN if NUMLIN_VISIBLE_LAST is defined. The natural width is the sum of the column width from column 0 to NUMCOL_VISIBLE (inclusive), or column 0 plus the sum from column NUMCOL-NUMCOL_VISIBLE to NUMCOL if NUMCOL_VISIBLE_LAST is defined. Notice that since NUMCOL_VISIBLE and NUMLIN_VISIBLE do not include the titles then NUMCOL_VISIBLE+1 columns and NUMLIN_VISIBLE+1 lines are included in the sum.

The height of a line L depends on several attributes, first it checks the HEIGHT*L* attribute, then checks RASTERHEIGHT*L*, then when USETITLESIZE=YES or not in

callback mode the height of the title text for the line or if L=0 it searches for the heighest column title, if still could not define a height then if L!=0 it will use HEIGHTDEF, if L=0 then height will be 0. A similar approach is valid for the column width. The width of a column C first checks the WIDTH*C* attribute, then checks RASTERWIDTH*C*, then when USETITLESIZE=YES or not in callback mode the width of the title text for the column or if C=0 it searches for the widest line title, if still could not define a width then if C!=0 it will use WIDTHDEF, if C=0 then height will be 0.

When the scrollbars are enabled if the matrix area is greatter than the visible area then scroolbars will be displayed so the cells can be scrolled to be visible area. But the scrooling position is not free, the first cell at the top right corner is always aligned at the begining of the cell. So the scroll increment is a cell and it can not be broken. One problem that raises from this policy is that depending on the visible size the last cell may never be completely visible. So solve the matrix size must be ajusted so all the last cells are visible within the visible area, or the natural size can be computed using the last cells instead of the first cells using the attributes NUMCOL_VISIBLE_LAST and NUMLIN_VISIBLE_LAST.

Any cell can have more than one text line, just use the \n control character. Multiple text lines will be considered when calculating the title cell size based on its contents. The contents of ordinary cells (not a title) do not affect the cell size.

Titles

A matrix might have titles for lines and columns. Titles are not scrollable, non editable and presented with a different default background color. A matrix will have a line of titles if an attribute of the "*L*:0" type is defined, where L is a line number. It will have a column of titles if an attribute of the "0:*C*" type is defined, where C is a column number.

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

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, after it is mapped the matrix never verifies attributes of type L:C again.

If VALUE_CB is defined and VALUE_EDIT_CB is not defined when the matrix is mapped then READONLY will be set to YES.

Keyboard Navigation

Keyboard navigation 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 fist column in the line.
- **Home Home**: Moves the focus to the upper left corner of the visible page.
- **Home Home Home**: Moves the focus to the upper left corner of the first page of the matrix.
- **End**: Moves the focus to the last column in the line.
- **End End**: Moves the focus to the lower right corner of the visible page.
- **End End End**: Moves the focus to the lower right corner of the last page in the matrix.
- **F2, Enter** or **Space**: enters editing mode.
- **Del**: remove the selected cells contents.

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 **F2, Enter** or **Space** is pressed, the current cell enters the edition mode with the current text of the cell. Double-clicking a cell also enters the edition mode (in Motif the user must click again to the edit control get the focus).

When using the keyboard to change the focus cell if the limit of the visible area is reached then the cells are automatically scrolled. Also if a cell partially visible is edited then first it is scrolled to the visible area.

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

- **Left, Right, Up and Down arrows**: if the caret is at the extremes of the text being edited then leave the edition mode and moves the focus accordingly. The value is confirmed.
- **Ctrl + arrows**: leave the edition mode and moves the focus accordingly independent of caret position. The value is confirmed.
- **Enter**: leave the edition mode. The value is confirmed. Moves the focus to the cell bellow.
- **Esc**: leave the edition mode. The new value is ignored and the old value remains.

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

Marks

When a mark mode is set the cells can be marked using mouse.

A marked cell will have its background attenuated to indicate that it is marked. A title cell appears marked only when MARKMODE=LIN, COL or LINCOL.

Cells can be selected individually or can be restricted to lines 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, if MARKMODE=CELL then all the cells of the line or column will be marked. Continuous areas are marked holding and dragging the mouse or holding the **Shift** key when clicking at the end of the area. Segmented areas are marked or unmarked holding the **Ctrl** key, the mark state is inverted. Clicking on the cell 0:0 will select all the items depending on MARKMODE, except for LINCOL.

Examples

Browse for Example Files

Inflation	January 2000	February 2000
Medicine Pharma	5.6 3.33	4.5
Food	2.2	8.1
Energy	7.2	3.4

1:1	1:2	1:3
2:1	2:2	2:3
3:1	3:2	3:3

See Also

IupCanvas

IupMatrix Attributes (all non inheritable with exceptions)

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".

FOCUS_CELL: Defines the current cell. Two numbers in the "***L***:***C***" format, (***L***>0 and ***C***>0, a title cell can NOT be the current cell). Default: "1:1".

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

READONLY: disables the editing of all cells. EDITION_CB and VALUE_EDIT_CB will not be called anymore. The L:C attribute will still be able to change the cell value. (since 3.0)

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).

USETITLESIZE: Use the title size to define the cell size if necessary. See WIDTHn and HEIGHTn. Default: NO. (since 3.0)

ACTIVE, EXPAND, FONT, X, Y, POSITION, MINSIZE, MAXSIZE, WID, TIP, SIZE, RASTERSIZE, ZORDER, VISIBLE: also accepted.

Cell Attributes (no redraw)

(These attributes are only updated in the display when you set the REDRAW attribute.)

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.

BGCOLOR: Background color of the matrix. (inheritable)
BGCOLOR*:*C*: Background color of column C.
BGCOLOR*L*:*: Background color of line L.
BGCOLOR*L*:*C*: Background color of the cell in line L and column C.

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 the global attribute TXTBGCOLOR 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. (inheritable)
FGCOLOR*:*C*: Text color of column C.
FGCOLOR*L*:*: Text color of line L.
FGCOLOR*L*:*C*: Text color of the cell in line L and column C.

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 the global attribute TXTFGCOLOR for cells or the global attribute DLGFGCOLOR for titles.

FONT: Character font of the text. (inheritable)
FONT*L*:*: Text font of the cells in line L.
FONT*:C*: Text font of the cells in column C.
FONT*L*:*C*: Text font of the cell in line L and column C.

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.

FRAMECOLOR: Sets the color to be used in the frame lines. (inheritable)
FRAMEVERTCOLOR*L*:*C*: Color of the vertical right frame of the cell. When not defined the FRAMECOLOR is used. For a title cell defines right and left frames.
FRAMEHORIZCOLOR*L*:*C*: Color of the horizontal bottom frame of the cell. When not defined the FRAMECOLOR is used. For a title cell defines bottom and top frames.

VALUE: Allows setting or verifying the value of the current cell. Is the same as obtaining the current cell from FOCUS_CELL value and using it as the attribute name for "L:C". 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.

Column Attributes (no redraw)

ALIGNMENT*C*: Alignment of the cells in column C (C >= 0). Possible values: "ALEFT", "ACENTER" or "ARIGHT". Default: "ALEFT" for n=0 and "ACENTER" for n>0. Not used for the column title. The 0:0 corner is always ACENTER, and the other column titles are also centered but if they do not fit in the cell then the alignment is changed to ALEFT.

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

Column Size Attributes

For all columns if WIDTHn is not defined, then RASTERWIDTHn is used. If also not defined, then depending on the circumstances a logic is used to find the column width.

If it is the title column (n=0), then if USETITLESIZE=YES or not in callback mode, it will search for the maximum width among the titles of all lines. Finally if the conditions are not true or the maximum width of the column is 0, then the column of line titles is hidden.

If it is a regular column (n>0), then if USETITLESIZE=YES, then it will use the width of the title of the column. Finally if the condition is not true or the width of the title of the column is 0, then the default value WIDTHDEF is used.

RASTERWIDTHn: Same as WIDTHn but in pixels. Has lower priority than WIDTHn.

WIDTHn: Width of column n in SIZE units, where n is the number of the column (n>=0). If the width value is 0, the column will not be shown on the screen. It does not includes the decoration size occupied by the frame lines.

WIDTHDEF: Default column width in SIZE units. Not used for the title column. Default: 80 (width corresponding to 20 characters).

Line Size Attributes

For all lines if HEIGHTn is not defined, then RASTERHEIGHTn is used. If also not defined, then depending on the circumstances a logic is used to find the line height.

If it is the title line (n=0), then if USETITLESIZE=YES or not in callback mode, it will search for the maximum height among the titles of all columns. Finally if the conditions are not true or the maximum height of the line is 0, then the line of column titles is hidden.

If it is a regular line (n>0), then if USETITLESIZE=YES, then it will use the height of the title of the line. Finally if the condition is not true or the height of the title of the line is 0, then the default value HEIGHTDEF is used.

HEIGHTn: Height of line n in SIZE units, where n is the number of the line (n>=0). If the height value is 0, the line will not be shown on the screen. It does not includes the decoration size occupied by the frame lines.

HEIGHTDEF: Default line height in SIZE units. Not used for the title line. Default: 8 (height corresponding to 1 line).

RASTERHEIGHTn: Same as HEIGHTn but in pixels. Has lower priority than HEIGHTn.

Number of Cells Attributes

NUMCOL: Defines the number of columns in the matrix. Must be an integer number. Default: "0". It does not includes the title column. If changed after map will add empty cells or discard cells at the end.

NUMCOL_VISIBLE: When set defines the number of visible columns to be counted when calculating the **Natural** size, not counting the title column. Not used elsewhere. The **Natural** size will always include the title column if any. Also it will always use the first columns of the matrix, except if **NUMCOL_VISIBLE_LAST**=YES then it will use the last columns. The remaining columns will be accessible only by using the scrollbar. When retrieved returns the current number of visible lines. Default: "4".

NUMLIN: Defines the number of lines in the matrix. Must be an integer number. Default: "0". It does not includes the title line. If changed after map will add empty cells or discard cells at the end.

NUMLIN_VISIBLE: When set defines the number of visible lines to be counted when calculating the **Natural** size, not counting the title line. Not used elsewhere. The **Natural** size will always include the title line if any. Also it will always use the first lines of the matrix, except if **NUMLIN_VISIBLE_LAST**=YES then it will use the last lines. The remaining lines will be accessible only by using the scrollbar. When retrieved returns the current number of visible lines. Default: "3".

Mark Attributes

AREA: Defines if the area to be **interactively** marked by the user must be continuous or not, valid only if MARKMULTIPLE=YES. Possible values: "CONTINUOUS" or "NOT_CONTINUOUS". Default: "CONTINUOUS".

MARKMODE: Defines the entity that can be marked: none, lines, columns, (lines or columns), and cells. Possible values: "NO", "LIN", "COL", "LINCOL" or "CELL". Default: "NO" (no mark).

MARK*L:C*: marks a cell, a line or a column depending on MARKMODE, and returns cell, line or column mark state according to MARKMODE. Can be "1" or "0". If MARKMODE=LIN,COL,LINCOL use 0 to mark only the other element (ex: "0:3" set/get for column 3). Even when MARKMODE=LIN,COL,LINCOL you can specify a single cell address. (since 3.0)

MARKED: String of '0' or '1' characters, informing which cells are marked (indicated by value '1'). Use NULL to clear all marks, returns NULL if no marks. The format of this character vector depends on the value of the MARKMODE attribute: if its value is CELL, the vector will have NUMLIN x NUMCOL positions, corresponding to all the cells in the matrix starting with all the cells of the first line, then the second line and so on. 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. If you change the other mark attributes the marked cells are cleared. When setting the attribute the LIN and COL notation can be used even if MARKMODE=CELL. MULTIPLE and AREA are NOT considered when setting MARKED or MARKL:C.

MULTIPLE: Defines if more than one entity defined by MARKMODE can be **interactively** marked. Possible values: "YES" or "NO". Default: "NO".

Action Attributes

ADDCOL (write-only): Adds a new column to the matrix after 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 operation mode. Can NOT add a title column. Ignored if set before map.

ADDLIN (write-only): Adds a new line to the matrix after 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 operation mode. Can NOT add a title line. Ignored if set before map.

DELCOL (write-only): 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 operation mode. Can NOT remove a title column, ***C***>0. Ignored if set before map.

DELLIN (write-only): 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 operation mode. Can NOT remove a title line, ***L***>0. Ignored if set before map.

EDIT_MODE: When set to YES, programmatically 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 scroll to a line or a column, use a value such as "***L***:*" or "*:***C**" (where ***L***>0 and ***C***>0).

SHOW (write-only): If necessary scroll the visible area to make the given cell visible. To scroll to a line or a column, use a value such as "***L***:*" or "*:***C**" (where ***L***>0 and ***C***>0). (since 3.0)

REDRAW (write-only): 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 the attributes: L:C, ALIGNMENTn, BGCOLOR*, FGCOLOR*, FONT*, VALUE, FRAME*COLOR. Global and size attributes always automatically redraw the matrix.

Text Edition Attributes

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

MASK*L:C*: Defines a mask that will filter text input. The MASK auxiliary attributes are also available adding the line and column at the end of the attribute name.

MULTILINE: allows the edition of multiple lines. Use Shift+Enter to add lines. Enter will end the editing.

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

Canvas Attributes (inheritable)

BORDER: Changed to NO.

SCROLLBAR: Changed to YES.

IupMatrix Callbacks

Interaction

ACTION_CB: Action generated when a keyboard event occurs.

```
int function(Ihandle *ih, int c, int lin, int col, int edition, char* after); [in C]
elem:action_cb(c, lin, col, edition, after: number) -> (ret: number) [in Lua]
```

ih: 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.
edition: 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 *ih, int lin, int col, char *status); [in C]
elem:click_cb(lin, col: number, status:string) -> (ret: number) [in Lua]
```

ih: 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.
status: Status of the mouse buttons and some keyboard keys at the moment the event is generated. The same macros used for BUTTON_CB can be used for this status.

To avoid the display update 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 *ih, int lin, int col, char *status); [in C]
elem:click_cb(lin, col: number, status:string) -> (ret: number) [in Lua]
```

ih: 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.
status: Status of the mouse buttons and some keyboard keys at the moment the event is generated. The same macros used for BUTTON_CB can be used for this status.

To avoid the display update return IUP_IGNORE.

MOUSEMOVE_CB: Action generated to notify the application that the mouse has moved over the matrix.

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

ih: 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 *ih, int lin, int col); [in C]
elem:enteritem_cb(lin, col: number) -> (ret: number) [in Lua]
```

ih: 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 *ih, int lin, int col); [in C]
elem:leaveitem_cb(lin, col: number) -> (ret: number) [in Lua]
```

ih: 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 *ih, int lin, int col); [in C]
elem:scrolltop_cb(lin, col: number) -> (ret: number) [in Lua]
```

ih: 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 *ih, 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]
```

ih: 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 *ih, 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]
```

ih: 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.

FONT_CB: Action generated to verify the font of a cell. Called both for common cells and for line and column titles. (since 3.0)

```
char* function(Ihandle* ih, int lin, int col); [in C]
elem:value_cb(lin, col: number) -> (ret: string) [in Lua]
```

ih: Identifier of the matrix interacting with the user.
lin, **col**: Coordinates of the cell.

Must return a font or NULL to use the the attribute defined font.

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 *ih, 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]
```

ih: 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 *ih, int lin, int col); [in C]
elem:dropcheck_cb(lin, col: number) -> (ret: number) [in Lua]
```

ih: 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 *ih, Ihandle *drop, int lin, int col); [in C]
elem:drop_cb(drop: ihandle, lin, col: number) -> (ret: number) [in Lua]
```

ih: 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 *ih, 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]
```

ih: 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. Not called if READONLY=YES.

```
int function(Ihandle *ih, int lin, int col, int mode, int update);  [in C]
elem:edition_cb(lin, col, mode, update: number) -> (ret: number) [in Lua]
```

ih: 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.
update: used when mode=0 to identify if the value will be updated whe the callback returns with IUP_DEFAULT. (since 3.0)

When **mode**=1 editing is allowed if the callback returns IUP_DEFAULT, so to make the cell read-only return IUP_IGNORE.

When **mode**=0 the new value is accepted only if the callback returns IUP_DEFAULT. The VALUE attribute returns the new value that will be updated in the cell. If the callback returns IUP_CONTINUE the edit mode is ended and the new value will not be updated, so the application can set a different value during the callback (usefull to format the new value). If the callback returns IUP_IGNORE the editing is not ended.

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. In this case **update** is 0.

This callback is also called if the user cancel the editing with **Esc** (in this case **update** is 0)**,** 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* ih, int lin, int col); [in C]
elem:value_cb(lin, col: number) -> (ret: string) [in Lua]
```

ih: Identifier of the matrix interacting with the user.
lin, **col**: Coordinates of the cell.

Must return the string to be redrawn.

IMPORTANT: The existance of this callback defines the callback operation mode of the matrix when it is mapped.

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). Not called if READONLY=YES. This callback is usually set in callback mode, but also works in normal mode.

```
int function(Ihandle *ih, int lin, int col, char* newval); [in C]
elem:value_edit_cb(lin, col: number, newval: string) -> (ret: number) [in Lua]
```

ih: Identifier of the matrix interacting with the user.
lin, **col**: Coordinates of the cell.
newval: String containing the new cell value

IMPORTANT: if VALUE_CB is defined and VALUE_EDIT_CB is not defined when the matrix is mapped it will be read-only.

MARK_CB: Action generated to verify the selection state of a cell. Called only for common cells, only when MARKMODE=CELL and only in callback mode.

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

ih: 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 "**MARK*L: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, only when MARKMODE=CELL and only in callback mode.

```
int function(Ihandle *ih, int lin, int col, int marked); [in C]
elem:markedit_cb(lin, col, marked: number) -> (ret: number) [in Lua]
```

ih: 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 "**MARK*L:C***" will be updated. So if you define the **MARKEDIT_CB** the "**MARK*L: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 "**MARK*L: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.

IupGauge (Deprecated since 3.0, will be removed in a future version)

Use the IupProgressBar control of the main library.

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]
```

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

Attributes

DASHED: Changes the style of the gauge for a dashed pattern. Default is "NO".

FGCOLOR: Controls the gauge and text color. The default is "64 96 192".

MAX (non inheritable): Contains the maximum value. Default is "1".

MIN (non inheritable): Contains the minimum value. Default is "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. Default value: "0x0". (since 3.0)

SHOW_TEXT: Indicates if the text inside the Gauge is to be shown or not. If the gauge is dashed the text is never shown. Possible values: "YES" or "NO". Default: "YES".

SIZE (non inheritable): The initial size is "120x14". Set to NULL to allow the automatic layout use smaller values.

TEXT (non inheritable): Contains a text to be shown inside the Gauge when SHOW_TEXT=YES. If it is NULL, the percentage calculated from VALUE will be used. If the gauge is dashed the text is never shown.

VALUE (non inheritable): Contains a number between "MIN" and "MAX", controlling the current position.

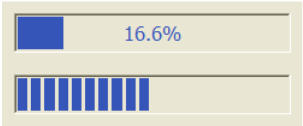
ACTIVE, BGCOLOR, EXPAND, FONT, X, Y, POSITION, MINSIZE, MAXSIZE, WID, TIP, RASTERSIZE, ZORDER, VISIBLE: also accepted.

Callbacks

MAP_CB, UNMAP_CB: common callbacks are supported.

Examples

Browse for Example Files



The Two Types of Gauge

See Also

IupCanvas

- Creation
- Attributes
- Callbacks
- Notes
- Examples
- See Also

(Old) IupTabs (Deprecated since 3.0, will be removed in a future version)

This control was moved to the main library under the same name, see IupTabs.

If you still need to use the old version, then you must call the function **IupOldTabsOpen**(). It will replace the registration of the new IupTabs by the old IupTabs. It does NOT work in GTK.

This is an additional control that depends on the CD library. It is included in the Controls Library.

It inherits from IupCanvas.

It contains a IupZbox to control the groups of controls. The **IupZbox** is a child of the **IupCanvas**, so all the attributes set in the Tabs will affect its children by attribute inheritance. To acess the Zbox simply call IupGetNextChild or IupGetChild for the Tabs.

Old Attributes (not supported in the new IupTabs)

ALIGNMENT (non inheritable): propagates the attribute to the Zbox. See the IupZbox documentation.

FONT_ACTIVE (non inheritable): Indicates the font to be used when the tab is selected.

FONT_INACTIVE (non inheritable): Indicates the font to be used when the tab is inactive.

TABSIZE (non inheritable) **(at children or element)**: 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 Tabs element, thus affecting all tabs, or to any tab child. If both are defined, the size of the tab child will have priority over the global size.

REPAINT (non inheritable): Since the element depends on attributes that are set in its children, this attribute updates the element internal and visual states, then redraw the Tabs.

ACTIVE, BGCOLOR, FONT, X, Y, POSITION, MINSIZE, MAXSIZE, WID, SIZE, RASTERSIZE, ZORDER, VISIBLE: also accepted.

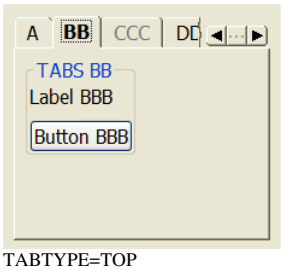
Notes

A Tab is inactive when its child is inactive. After setting the ACTIVE attribute on the child, set UPDATE on the **IupTabs** so its visual state is updated. (not supported in the new IupTabs)

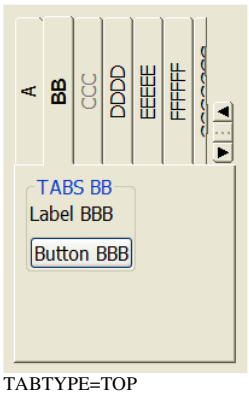
In the new IupTabs, the IUP_IGNORE return code in the callback is not supported anymore.

Examples

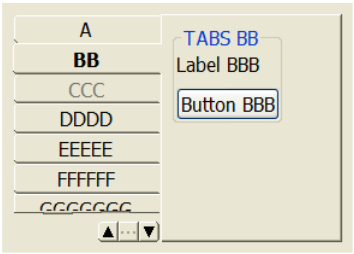
Browse for Example Files



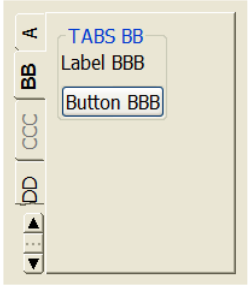
TABTYPE=TOP
TABORIENTATION=HORIZONTAL



TABTYPE=TOP
TABORIENTATION=VERTICAL



TABTYPE=LEFT
TABORIENTATION=HORIZONTAL



TABTYPE=LEFT
TABORIENTATION=VERTICAL

See Also

IupCanvas, IupTabs (3.0)

(Old) IupVal (Deprecated since 3.0, will be removed in a future version)

This control was moved to the main library under the same name, see IupVal.

If you still need to use the old version, then you must call the function **IupOldValOpen()**. It will replace the registration of the new IupVal by the old IupVal.

This is an additional control that depends on the CD library. It is included in the Controls Library.

It inherits from IupCanvas.

Old Attributes (not supported in the new IupVal)

HANDLER_IMAGE (non inheritable): Handler image name. When defined the handle will not be drawn an this image will be used instead.

HANDLER_IMAGE_INACTIVE (non inheritable): Inactive handler image name. Used when **HANDLER_IMAGE** is used and the control is inactive.

Old Callbacks

MOUSEMOVE_CB: Called each time the user moves the valuators thumb keeping the mouse button pressed. The value of VALUE is passed as parameter.

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

ih: identifier of the element that activated the event.
val: the valuator value.

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 *ih, double val); [in C]
elem:button_press_cb(val: number) -> (ret: number) [in Lua]
```

ih: identifier of the element that activated the event.
val: the valuator value.

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.

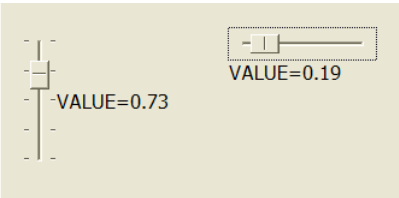
```
int function(Ihandle *ih, double val); [in C]
elem:button_release_cb(val: number) -> (ret: number) [in Lua]
```

ih: identifier of the element that activated the event.
val: the valuator value.

Notes

When the keyboard arrows are pressed and released, or the mouse wheel is rotated, the mouse press and the mouse release callbacks are called, in this order. If you hold the key down a mouse move callback is also called. In these cases the value is incremented by 10% of the interval max-min.

Examples



See Also

IupCanvas, IupVal (3.0)

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 iupluagl.h file must also be included in the source code. The program must be linked to the lua control library (iupluagl).

To link with the OpenGL libraries in Windows add: opengl32.lib. In UNIX add before the X-Windows libraries: -LGL.

The Lua binding does not include OpenGL functions. But you can use any OpenGL binding available.

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.

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.

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".

BUFFER: Indicates if the buffer will be single "SINGLE" or double "DOUBLE". Default is "SINGLE".

BUFFER_SIZE: Indicates the number of bits for representing the color indices (valid only for INDEX). The system default is 8 (256-color palette).

COLOR: Indicates the color model to be adopted: "INDEX" or "RGBA". Default is "RGBA".

COLORMAP (read-only)**:** Returns "Colormap" in UNIX and "HPALETTE" in Win32, if COLOR=INDEX.

CONTEXT (read-only)**:** Returns "GLXContext" in UNIX and "HGLRC" in Win32.

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.

ERROR (read-only): If an error is found, returns a string containing a description of the error in English.

RED_SIZE, **GREEN_SIZE** and **BLUE_SIZE**: Indicate the number of bits for representing each color component (valid only for RGBA). The system default is usually 8 for each component (True Color support).

REFRESHCONTEXT (write-only) [Windows Only]: action attribute to refresh the internal device context when it is not owned by the window class. The IupCanvas of the Win32 driver will always create a window with an owned DC, but GTK in Windows will not. (since 3.0)

STENCIL_SIZE: Indicates the number of bits in the stencil buffer. Value 0 means the stencil 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.

VISUAL (read-only)**:** Returns "XVisualInfo*" in UNIX and "HDC" in Win32.

Callbacks

The **IupGLCanvas** element understands all callbacks defined for a conventional canvas, see IupCanvas.

Addicionally:

RESIZE_CB: By default the resize callback sets:

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

Auxiliary Functions

These are auxiliary functions based on the WGL and XGL extensions. Check the respective documentations for more information.

```
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.

```
void IupGLUseFont(Ihandle* ih, int first, int count, int list_base); [in C]
iup.GLUseFont(ih:ihandle, first, count, list_base: number) [in Lua]
```

Creates a bitmap display list from the current FONT attribute. (since 3.0)

```
void IupGLWait(int gl) [in C]
iup.GLWait(gl: number) [in Lua]
```

If gl is non zero it will call glFinish or glXWaitGL, else will call GdiFlush or glXWaitX. (since 3.0)

Notes

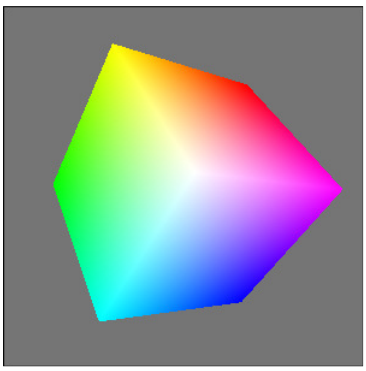
In Windows XP, if the COMPOSITE attribute is enabled then the hardware acceleration will be disabled.

The **IupGLCanvas** works with the GTK base driver in Windows and in UNIX (X-Windows).

Not available in SunOS510x86 just because we were not able to compile OpenGL code in our installation.

Examples

Browse for Example Files



See Also

IupCanvas

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 writen 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, **iuppplotlua_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. Al other plot parameters are configured by attributes.

If no attribute is set, the default values are choosed 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 occours.

The **dataset area** is delimited by a margin. Data is only plotted inside the dataset area. Axis and main title are positioned indepent of this margin. It is very important to set the margins when using axis automatic scaling or the axis itih maybe 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 the 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.

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* ih, int strXdata); [in C]
iup.PPlotBegin(ih: 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* ih, float x, float y); [in C]
iup.PPlotAdd(ih: 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* ih, const char* x, float y); [in C]
iup.PPlotAddStr(ih: 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* ih); [in C]
iup.PPlotEnd(ih: 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 *ih, int index, int sample_index, float x, float y); [in C]
void IupPPlotInsertStr(Ihandle *ih, int index, int sample_index, const char* x, float y);
iup.IupPPlotInsert(ih: ihandle, index, sample_index, x, y: number) [in Lua]
iup.IupPPlotInsertStr(ih: 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* ih, float x, float y, int *ix, int *iy); [in C]
iup.PPlotTransform(ih: 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* ih, cdCanvas* cnv); [in C]
iup.PPlotPaintTo(ih: ihandle, cnv: cdCanvas) [in Lua]
```

Plots to the given CD canvas instead of the display canvas.

Attributes

REDRAW (write-only) (non inheritable): 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 (non inheritable): 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] (non inheritable): 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 used 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 (non inheritable): the title. Located always at the top center area.

TITLEFONTSIZE, TITLEFONTSTYLE (non inheritable): 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 (non inheritable): margin of the dataset area. Default: "15", "15", "30", "15".

Legend Configuration (non inheritable)

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.

Grid Configuration (non inheritable)

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.

Dataset Management (non inheritable)

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 becames the current dataset. The index starts at 0. All "DS_*" attributes are dependent on this value.

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.

Axis Configuration (non inheritable)

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.

ACTIVE, X, Y, POSITION, MINSIZE, MAXSIZE, WID, TIP, SIZE, RASTERSIZE, ZORDER, VISIBLE: also accepted.

Callbacks

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 *ih, 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 coodinate value of the sample
y: Y coodinate 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 *ih); [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 te sample is about to be changed.

```
int function(Ihandle *ih, 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 coodinate value of the sample
y: Y coodinate 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 *ih); [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 te sample is about to be changed.

```
int function(Ihandle *ih, 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 coodinate value of the sample
y: Y coodinate value of the sample
new_x: the new X coodinate value of the sample
new_y: the new Y coodinate value of the sample

If the return value is IUP_IGNORE then the sample is not edited. The application can changed 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 *ih); [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 addicional information in the plot. Predraw has no restrictions, but Postdraw is clipped to the dataset area. To position elements in plot units, use the **IupPPlotTransform** function.

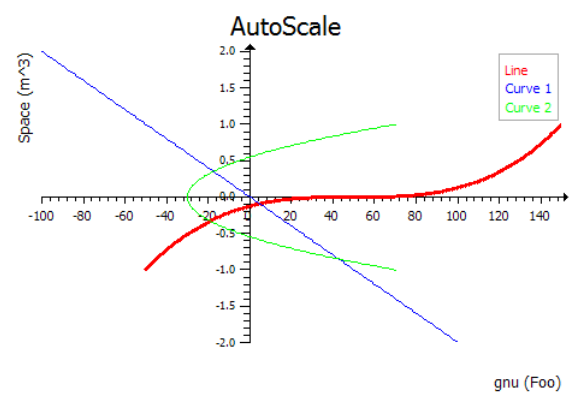
```
int function(Ihandle *ih, cdCanvas* cnv); [in C]
elem:predraw_cb(cnv: cdCanvas) -> (ret: number) [in Lua]
elem:postdraw_cb(cnv: cdCanvas) -> (ret: number) [in Lua]
```

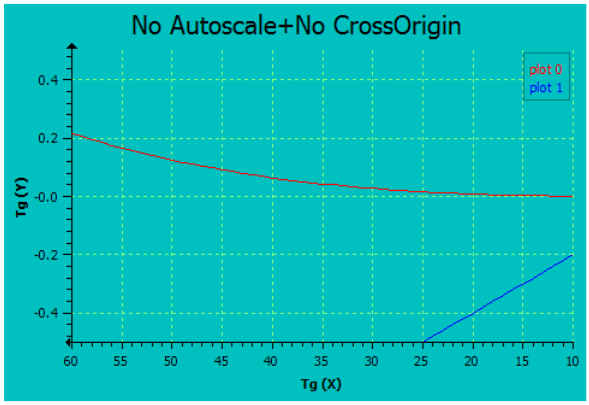
cnv: the CD canvas where the draw operation occours.

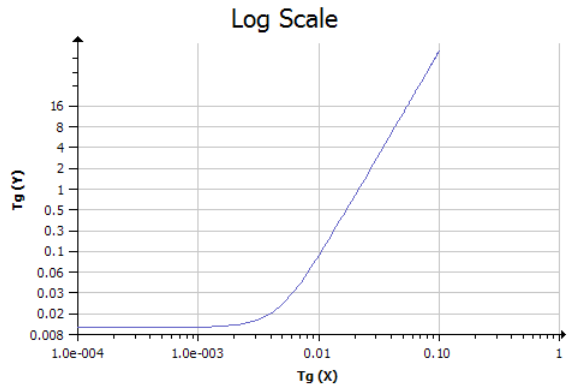
MAP_CB, UNMAP_CB, GETFOCUS_CB, KILLFOCUS_CB, ENTERWINDOW_CB, LEAVEWINDOW_CB, K_ANY, HELP_CB: All common callbacks are supported.

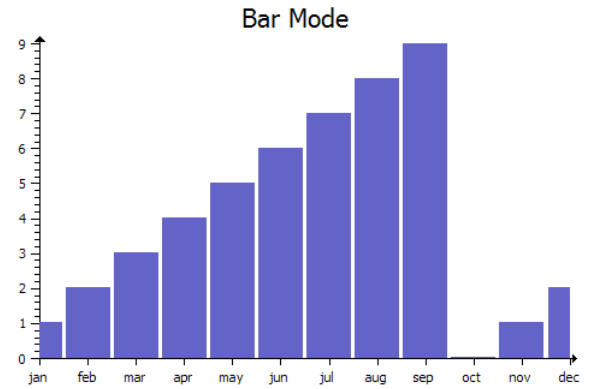
Examples

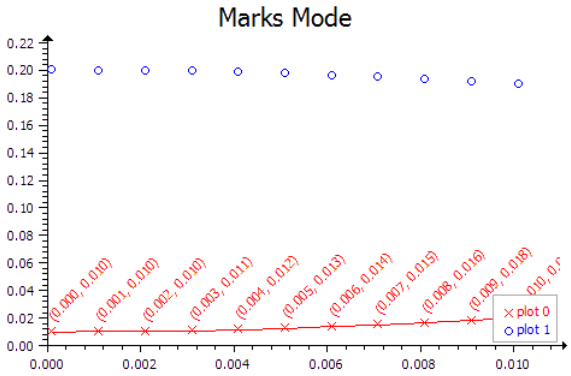
Browse for Example Files

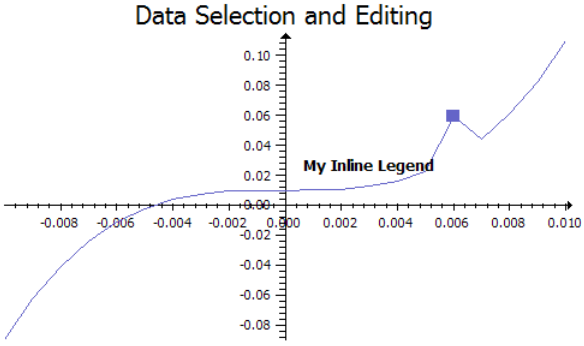












See Also

IupCanvas

- Creation
- Attributes
- Callbacks
- Notes
- Examples
- See Also

IupOleControl [Windows only]

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

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

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

Initialization and usage

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

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

Attributes

DESIGNMODE: activates the design mode. Some controls behave differently when in design mode. See this article for more information about design mode. Can be YES or NO. Default value: "NO".

DESIGNMODE_DONT_NOTIFY: sets the design mode, but do not notify the native control.

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

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

Some IupCanvas attributes may also work, like:

ACTIVE, X, Y, POSITION, MINSIZE, MAXSIZE, WID, TIP, SIZE, RASTERSIZE, ZORDER, VISIBLE

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, call elem:CreateLuaCOM so a LuaCOM object is created and stored in the **"elem.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.

Some **IupCanvas** callbacks may also work, like:

MAP_CB, UNMAP_CB, GETFOCUS_CB, KILLFOCUS_CB, ENTERWINDOW_CB, LEAVEWINDOW_CB.

Additional Methods in Lua

```
elem:CreateLuaCOM()
```

If LuaCOM is loaded and the IUNKNOWN is valid then set:

```
elem.com = luacom.CreateLuaCOM(luacom.ImportIUnknown(elem.iunknown))
```

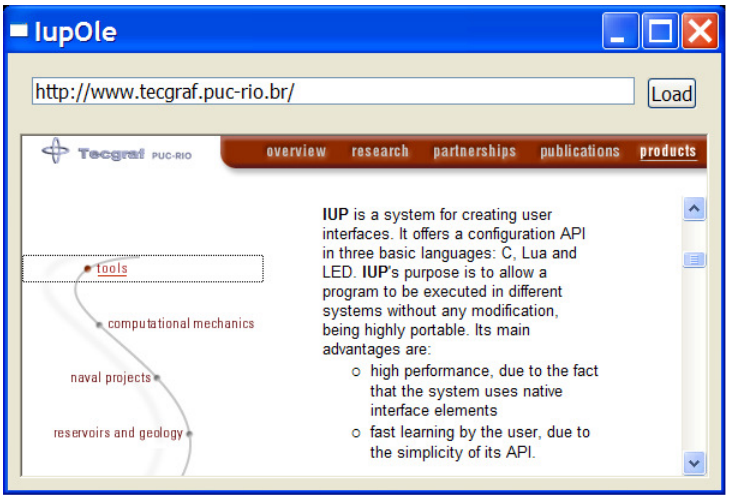
Notes

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 for Example Files



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.

FONT (up to 2.x)

Character font of the text shown in the element. See FONT definition since IUP 3.0.

Value

Pre-defined font name (see Font Names) or a string in the native format.

In **Windows**, the native format is as a string with the following format:

"**`name:attributes:size`**"

`name`: The name the user will see (Times New Roman, MS Sans Serif, etc.).
`attributes`: Can be empty, or a list separated by commas with the following names: `BOLD ITALIC UNDERLINE STRIKEOUT`
`size`: Size in pixels

Examples:

```
"Times New Roman::10"
"Ms Sans Serif:ITALIC:20"
"Courier New:BOLD,STRIKEOUT:15"
```

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

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

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

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

Affects

All elements with an associated text.

Note

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.

Font Names

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"

See Also

IupMapFont, IupUnMapFont.

IupMapFont (Deprecated since 3.0)

Retrieves the name of a native font, given the name of the old IUP FONT names. See the old Font Names table for a list of names.

Parameters/Return

```
char* IupMapFont(const char *iupfont); [in C]
iup.MapFont(iupfont : string) -> (driverfont : string) [in Lua]
```

Returns: the name of the native font.

See Also

IupUnMapFont

IupUnMapFont (Deprecated since 3.0)

Retrieves the name of the old IUP FONT names, given the native font. See the old Font Names table for a list of names.

Parameters/Return

```
char* IupUnMapFont(const char *driverfont); [in C]
iup.UnMapFont(driverfont :string) -> (iupfont : string) [in Lua]
```

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

- 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 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.image{width = width: number, height = height: number, pixels = pixels: 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 discart it after the call.
pixel0, **pixel1**, **pixel2**, ...: Value of the pixels. But for **IupImageRGB** and **IupImageRGBA** in fact will be one value for each color channel (pixel_r_0, pixel_g_0, pixel_b_0, pixel_r_1, pixel_g_1, pixel_b_1, pixel_r_2, pixel_g_2, pixel_b_2, ...).
line0, **line1**: unnamed tables, one for each line containing pixels values. See Notes bellow.
colors: table named colors containing the colors indices. See Notes bellow.

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

Attributes

"0" Color in index 0.
"1" Color in index 1.
"2" Color in index 2.
...
"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.

BPP (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 internal 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.

Images created with the **IupImage*** constructors can be reused in different elements.

The images should be destroyed when they are no longer necessary, by means of the **IupDestroy** function. To destroy an image, it cannot be in use, i.e the controls where it is used should be destroyed first. Images that were associated with controls by names are automatically destroyed in IupClose.

Please observe the rules for creating cursor images: CURSOR.

Usage

Images are used in elements such as buttons and labels by attributes that points to names registered with IupSetHandle. You can also use **IupSetAttributeHandle** to shortcut the set of an image as an attribute. For example:

```
Ihandle* image = IupImage(width, height, pixels);

IupSetHandle("MY_IMAGE_NAME", image);
IupSetAttribute(label, "IMAGE", "MY_IMAGE_NAME");
or
IupSetAttributeHandle(label, "IMAGE", image); // an automatic name will be created internally
```

In Windows, names of resources in RC files linked with the application are also accepted. In GTK, names of GTK Stock Items are also accepted. In Motif, names of bitmaps installed on the system are also accepted. For example:

```
IupSetAttribute(label, "IMAGE", "TECGRAF_BITMAP");  // available in the "etc/iup.rc" file
or
IupSetAttribute(label, "IMAGE", "gtk-open");  // available in the GTK Stock Items
```

In all drivers, a path to a file name can also be used as the attribute value (since 3.0). But the available file formats supported are system dependent. The Windows driver supports BMP, ICO and CUR. The GTK driver supports the formats supported by the GDK-PixBuf library, such as BMP, GIF, JPEG, PCX, PNG, TIFF and many others. The Motif driver supports the X-Windows bitmap. For example:

```
IupSetAttribute(label, "IMAGE", "../etc/tecgraf.bmp");
```

Colors

In Motif, the alpha channel in RGBA images is composed with the control BGCOLOR by IUP prior to setting the image at the control. In Windows and in GTK, the alpha channel is composed internally by the system. But in Windows for some controls the alpha must be composed prior also, it includes: **IupItem** and **IupSubmenu** always; and **IupToggle** when NOT using Visual Styles. This implies that if the control background is not uniform then probably there will be a visible difference where it should be transparent.

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 cian)
 7 = 192,192,192 (gray)
 8 = 128,128,128 (dark gray)
 9 = 255,  0,  0 (red)
10 =   0,255,  0 (green)
11 = 255,255,  0 (yellow)
12 =   0,  0,255 (blue)
13 = 255,  0,255 (magenta)
14 =   0,255,255 (cian)
15 = 255,255,255 (white)
```

For images with more than 16 colors, 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.

Samples

You can obtain several images from the IupImageLib, a library of pre-defined images. To view the images you can use the **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.

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 Old Constructor

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:

```
img = iup.image{
  { 1,2,3,3,3,3,3,3,2,1 },
  { 2,1,2,3,3,3,3,3,2,1,2 },
  { 3,2,1,2,3,3,3,2,1,2,3 },
  { 3,3,2,1,2,3,2,1,2,3,3 },
  { 3,3,3,2,1,2,1,2,3,3,3 },
  { 3,3,3,3,2,1,2,3,3,3,3 },
  { 3,3,3,2,1,2,1,2,3,3,3 },
  { 3,3,2,1,2,3,2,1,2,3,3 },
  { 3,2,1,2,3,3,3,2,1,2,3 },
  { 2,1,2,3,3,3,3,3,2,1,2 },
  { 1,2,3,3,3,3,3,3,3,2,1 };
  colors = {
    "0 1 0",       -- index 1
    "255 0 0",     -- index 2
    "255 255 0"    -- index 3
  }
}
```

Using this constructor the image data can NOT has 0 indices. Notice that the indexing of the unamed tables is different than the **colors** table. The first value in the **colors** table is relative to the color index 1, but the first value of the unamed tables is relative to the pixel 0.

After the image is created in Lua, the unnamed 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" function, for example:

```
lin0 = rawget(img, 1)  -- line index 0
lin1 = rawget(img, 2)  -- line index 1
lin2 = rawget(img, 3)  -- line index 2
...
pixel0 = lin0[1]       -- column index 0
pixel1 = lin0[2]       -- column index 1
pixel3 = lin0[3]       -- column index 3
...
```

IupLua New Constructors

The new constructors since IUP 3 are different and must contains explicit values for **width**, **height** and **pixels**. Also the indexing of the **colors** table is the same of the **pixels** table, the first value is the color index 0. For example:

```
img = iup.image{
  width = 11,
  height = 11,
  pixels = {
   1,2,0,0,0,0,0,0,2,1,
   2,1,2,0,0,0,0,2,1,2,
   0,2,1,2,0,0,0,2,1,2,0,
   0,0,2,1,2,0,2,1,2,0,0,
   0,0,0,2,1,2,1,2,0,0,0,
   0,0,0,0,2,1,2,0,0,0,0,
   0,0,0,2,1,2,1,2,0,0,0,
   0,0,2,1,2,0,2,1,2,0,0,
   0,2,1,2,0,0,0,2,1,2,0,
   2,1,2,0,0,0,0,0,2,1,2,
   1,2,0,0,0,0,0,0,0,2,1},
  colors = {
    "255 255 0"    -- index 0
    "0 1 0",       -- index 1
    "255 0 0",     -- index 2
  }
}
```

Although in Lua they are still referenced as index 1, so img.**colors**[1] returns the color of the index 0 in the image.

Here is the same image but using 24bpp:

```
img = iup.imagergb{
  width = 11,
```

```
  height = 11,
  pixels = {
    0,255,0,  255,  0,0,  255,255,0,  255,255,0,  255,255,0,  255,255,0,  255,255,0,  255,255,0,  255,255,0,  255,  0,0,    0,255,0,
  255,  0,0,    0,255,0,  255,  0,0,  255,255,0,  255,255,0,  255,255,0,  255,255,0,  255,255,0,  255,  0,0,    0,255,0,  255,  0,0,
  255,255,0,  255,  0,0,    0,255,0,  255,  0,0,  255,255,0,  255,255,0,  255,255,0,  255,  0,0,    0,255,0,  255,  0,0,  255,255,0,
  255,255,0,  255,255,0,  255,  0,0,    0,255,0,  255,  0,0,  255,255,0,  255,  0,0,    0,255,0,  255,  0,0,  255,255,0,  255,255,0,
  255,255,0,  255,255,0,  255,255,0,  255,  0,0,    0,255,0,  255,  0,0,    0,255,0,  255,  0,0,  255,255,0,  255,255,0,  255,255,0,
  255,255,0,  255,255,0,  255,255,0,  255,255,0,  255,  0,0,    0,255,0,  255,  0,0,  255,255,0,  255,255,0,  255,255,0,  255,255,0,
  255,255,0,  255,255,0,  255,255,0,  255,  0,0,    0,255,0,  255,  0,0,    0,255,0,  255,  0,0,  255,255,0,  255,255,0,  255,255,0,
  255,255,0,  255,255,0,  255,  0,0,    0,255,0,  255,  0,0,  255,255,0,  255,255,0,  255,  0,0,    0,255,0,  255,  0,0,  255,255,0,
  255,255,0,  255,  0,0,    0,255,0,  255,  0,0,  255,255,0,  255,255,0,  255,255,0,  255,  0,0,    0,255,0,  255,  0,0,  255,255,0,
  255,  0,0,    0,255,0,  255,  0,0,  255,255,0,  255,255,0,  255,255,0,  255,255,0,  255,  0,0,    0,255,0,  255,  0,0,  255,255,0,
    0,255,0,  255,  0,0,  255,255,0,  255,255,0,  255,255,0,  255,255,0,  255,255,0,  255,255,0,  255,  0,0,    0,255,0}
}
```

Then at the **pixels** table we have:

```
r0 = img.pixels[1]   g0 = img.pixels[2]   b0 = img.pixels[3]
r1 = img.pixels[4]   g1 = img.pixels[5]   b1 = img.pixels[6]
r3 = img.pixels[7]   g3 = img.pixels[8]   b3 = img.pixels[9]
...
```

If the image was created in C then there is no way to access its pixels values in Lua, except as an userdata using the WID attribute.

Examples

Browse for Example Files

See Also

IupLabel, IupButton, IupToggle, IupDestroy.

IupImageLib (up to 2.x)

A library of pre-defined images for buttons and labels. See IupImageLib since IUP 3.0.

Initialization

To generate an application that uses this function, the program must be linked to the functions library (iupimglib.lib on Windows and libiupimglib.a on Unix). The iupcontrols.h file must also be included in the source code.

The library is quite large because of the images. To avoid using all the images get the source code and extract only the image you need.

Reference

```
void IupImageLibOpen(void); [in C]
iup.ImageLibOpen()  [in Lua]
```

This function loads all the images in the library.

In Lua, when require"iupluaimglib" is used this function will be automatically called.

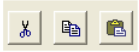
```
void IupImageLibClose(void); [in C]
iup.ImageLibClose()  [in Lua]
```

This function releases all the images of the library.

Usage

The following names are defined after the library initialization. The images do NOT include the button borders, this is just a preview for buttons!

The "BGCOLOR" color value is set and the colors are distributed so that the automatic disable color algorithm works fine. Images for buttons have size 20x20, small images are 11x11, and label images have height=30.

Names	Images
"IUP_IMGBUT_TEXT" "IUP_IMGBUT_NEW" "IUP_IMGBUT_NEWSPRITE" "IUP_IMGBUT_OPEN" "IUP_IMGBUT_CLOSE" "IUP_IMGBUT_CLOSEALL" "IUP_IMGBUT_SAVE"	
"IUP_IMGBUT_CUT" "IUP_IMGBUT_COPY" "IUP_IMGBUT_PASTE"	
"IUP_IMGBUT_PRINT" "IUP_IMGBUT_PREVIEW" "IUP_IMGBUT_SEARCH" "IUP_IMGBUT_HELP"	
"IUP_IMGBUT_REDO" "IUP_IMGBUT_UNDO" "IUP_IMGBUT_ONELEFT" "IUP_IMGBUT_ONERIGHT" "IUP_IMGBUT_TENLEFT" "IUP_IMGBUT_TENRIGHT"	
"IUP_IMGBUT_ZOOM" "IUP_IMGBUT_ZOOMIN" "IUP_IMGBUT_ZOOMOUT" "IUP_IMGBUT_NOZOOM" "IUP_IMGBUT_YZ" "IUP_IMGBUT_XY"	

"IUP_IMGBUT_XZ"
"IUP_IMGBUT_FIT"
"IUP_IMGBUT_AXIS"
"IUP_IMGBUT_CUBE"

"IUP_IMGBUT_TILE"
"IUP_IMGBUT_CASCADE"

"IUP_IMGBUT_STOP"
"IUP_IMGBUT_PLAY"
"IUP_IMGBUT_PREVIOUS"
"IUP_IMGBUT_NEXT"
"IUP_IMGBUT_PLAYBACKWARD"
"IUP_IMGBUT_FOWARD"
"IUP_IMGBUT_REWIND"

"IUP_IMGBUT_GREENLEFT"
"IUP_IMGBUT_GREENUP"
"IUP_IMGBUT_GREENRIGHT"
"IUP_IMGBUT_GREENDOWN"

"IUP_IMGBUT_CONFIGURE"
"IUP_IMGBUT_VIDEO"

"IUP_IMGSML_SINGLELEFT"
"IUP_IMGSML_DOUBLELEFT"
"IUP_IMGSML_SINGLERIGHT"
"IUP_IMGSML_DOUBLERIGHT"
"IUP_IMGSML_DOWN"
"IUP_IMGSML_LEFT"
"IUP_IMGSML_RIGHT"
"IUP_IMGSML_UP"

"IUP_IMGLBL_TECGRAF"
"IUP_IMGLBL_BR"
"IUP_IMGLBL_LUA"
"IUP_IMGLBL_TECGRAFPUCRIO"
"IUP_IMGLBL_PETROBRAS"

See Also

IupImage

IupImageLib (since 3.0)

A library of pre-defined images for buttons and labels. See IupImageLib up to IUP 2.x.

Initialization

To generate an application that uses this function, the program must be linked to the functions library (**iupimglib**.lib on Windows and **libiupimglib**.a on Unix).

The library is large and could increase the size of the executable. To avoid linking with the library get the source code and extract only the image you need.

Reference

```
void IupImageLibOpen(void); [in C]
iup.ImageLibOpen()  [in Lua]
```

This function register the names but do not load the images. The images will be loaded only if they are used in a control. The loaded images will be automatically released at IupClose.

In Lua, when require"iupluaimglib" is used this function will be automatically called.

Usage

The following names can be used after the library initialization. The name are NOT registered using **IupSetHandle**, they will be automatically loaded when associated with a control.

Bitmap Group

These bitmaps are 16x16-8bpp (Motif), 16x16-32bpp (Win32) or 24x24-32bpp (GTK) pixels size images that can be used in Buttons, usually inside toolbars. Do not set RASTERSIZE of buttons so they can expand when switching to GTK.

Name	Image (Motif)	Image (Win32)	Image (GTK)
"IUP_ActionCancel"			
"IUP_ActionOk"			
"IUP_ArrowDown"			
"IUP_ArrowLeft"			
"IUP_ArrowRight"			
"IUP_ArrowUp"			
"IUP_EditCopy"			

"IUP_EditCut"			
"IUP_EditErase"			
"IUP_EditFind"			
"IUP_EditPaste"			
"IUP_EditRedo"			
"IUP_EditUndo"			
"IUP_FileClose"			
"IUP_FileCloseAll"			
"IUP_FileNew"			
"IUP_FileOpen"			
"IUP_FileProperties"			
"IUP_FileSave"			
"IUP_FileSaveAll"			
"IUP_FileText"			
"IUP_FontBold"			
"IUP_FontDialog"			
"IUP_FontItalic"			
"IUP_MediaForward"			
"IUP_MediaGotoBegin"			
"IUP_MediaGoToEnd"			
"IUP_MediaPause"			
"IUP_MediaPlay"			
"IUP_MediaRecord"			
"IUP_MediaReverse"			
"IUP_MediaRewind"			
"IUP_MediaStop"			
"IUP_MessageError"			
"IUP_MessageHelp"			
"IUP_MessageInfo"			
"IUP_NavigateHome"			
"IUP_NavigateRefresh"			
"IUP_Print"			
"IUP_PrintPreview"			
"IUP_ToolsColor"			
"IUP_ToolsSettings"			
"IUP_ToolsSortAscend"			
"IUP_ToolsSortDescend"			
"IUP_ViewFullScreen"			
"IUP_WindowsCascade"			
"IUP_WindowsTile"			
"IUP_Zoom"			
"IUP_ZoomActualSize"			
"IUP_ZoomIn"			
"IUP_ZoomOut"			
"IUP_ZoomSelection"			

Icon Group

These icons are 32x32 pixels size (or just 32 pixels height) images that can be used in Labels, usually inside toolbars.

Name	Image (Generic)
"IUP_Tecgraf"	
"IUP_PUC-Rio"	
"IUP_BR"	
"IUP_Lua"	

Name	Image (Generic)
"IUP_TecgrafPUC-Rio"	
"IUP_Petrobras"	

Logo Group

These logos are 48x48 pixels size (or just 48 pixels height) images that can be used in Labels, usually inside dialogs.

Name	Image (Generic)
"IUP_LogoTecgraf"	
"IUP_LogoPUC-Rio"	
"IUP_LogoBR"	
"IUP_LogoLua"	

Name	Image (Generic)
"IUP_LogoTecgrafPUC-Rio"	
"IUP_LogoPetrobras"	

Other

Here are other images available in the IUP stock library, commonly used by the respective systems. All images are 48x48 pixels size and 32bpp.

Name	Image (Win32)	Image (GTK)
"IUP_LogoMessageSecurity"		
"IUP_LogoMessageWarning"		
"IUP_LogoMessageInfo"		
"IUP_LogoMessageError"		
"IUP_LogoMessageHelp"		

Name	Image (Win32)
"IUP_DeviceCamera"	
"IUP_DeviceCD"	
"IUP_DeviceCellPhone"	
"IUP_DeviceComputer"	
"IUP_DeviceFax"	
"IUP_DeviceMP3"	
"IUP_DeviceNetwork"	
"IUP_DevicePDA"	
"IUP_DevicePrinter"	
"IUP_DeviceScanner"	
"IUP_DeviceSound"	
"IUP_DeviceVideo"	

Notes

All 8bpp images are from the old ImageLib and since Motif does not have any stock images, we selected this set to be used in Motif. Although the IUP Motif driver supports 32bpp images.

Not available in AIX43.

All "Win32" images copyright Microsoft and were extracted from the Visual Studio 2005 Image Library. Their use **must** be used consistently with their description in the Visual Studio 2005 Image Library, and so consistently with the IUP name. These files are available with Microsoft Visual Studio 2005 for redistribution under the Visual Studio 2005 license.

GTK stock images are released under the GTK license.

PUC-Rio, Tecgraf/PUC-Rio, Petrobras and Lua images are copyright of the respective companies or owners.

Lua image graphic design by A. Nakonechnyj. Copyright © 1998. All rights reserved.

See Also

IupImage

IUP-IM Functions

Functions to load/save an **IupImage** from/to a 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 iupluaim.h file must also be included in the source code. The program must be linked to the lua functions library (iupluaim 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.

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* ih, const char* file_name, const char* format); [in C]
iup.SaveImage(ih: ihandle, file_name, format: string) -> (ret: boolean) [in Lua]
```

ih: handle of the **IupImage**.
file_name: Name of the file to be loaded.
format: format descriptor for IM. For ex: "BMP", "JPEG", "GIF", "TIFF", "PNG", "PNM", "PCX", "ICO", etc.

Returns: zero if failed. When failed a message box describing the error is displayed.

Native Handle to imImage

```
imImage* IupGetNativeHandleImage(void* handle); [in C]
iup.GetNativeHandleImage(handle: userdata) -> (image: imImage) [in Lua]
```

handle: image native handle. In Win32 is a **HANDLE** of a DIB. In GTK is a **GdkPixbuf***. In Motif is a **Pixmap**. Its memory is released after the **imImage** is created.

Returns: the **imImage*** handle. Usefull when pasting data from a **IupClipboard**.

You must include the "im_image.h" header before the "iupim.h" to enable these functions.

imImage to Native Handle

```
imImage* IupGetImageNativeHandle(imImage* image); [in C]
iup.GetImageNativeHandle(image: imImage) -> (handle: userdata) [in Lua]
```

image: the **imImage*** handle. Must be a bitmap image.

Returns: the image native handle. In Win32 is a **HANDLE** for a DIB. In GTK is a **GdkPixbuf***. In Motif is a **Pixmap**. Usefull when copying data to a **IupClipboard**.

You must include the "im_image.h" header before the "iupim.h" to enable these functions.

See Also

IupImage, IupSaveImageAsText, IupClipboard

IupSaveImageAsText (since 3.0)

Saves the **IupImage** as a text file to be reused in other programs.

It does NOT depends on the IM library.

Parameters/Return

```
int IupSaveImageAsText(Ihandle* ih, const char* file_name, const char* format, const char* name); [in C]
iup.SaveImageAsText(ih: ihandle, file_name, format[, name]: string) -> (ret: boolean) [in Lua]
```

ih: handle of the **IupImage**.
file_name: Name of the file to be loaded.
format: text format. Can be: "LED", "LUA" or "C".
name: name of the image. Can be NULL.

Returns: zero if failed, non zero value if sucess.

Notes

If name is NULL and the **IupImage** is associated with a name then that name is used, if no name is associated then "image" will be used.

See Also

IupImage, IUP-IM Functions

Keyboard

The application can control the focus using the functions **IupGetFocus** and **IupSetFocus**. When the focus is changed the application is notified trough the callbacks GETFOCUS_CB and KILLFOCUS_CB.

Keyboard navigation in the dialog uses the "Tab" and "Shilf+Tab" keys to change the keyboard focus from one control to another. The exception is when the focus is at an **IupMultiline** control, to change focus the combination "Ctrl+Tab" must be used, because "Tab" is a valid input for the text. 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. The order is the same implemented by the functions **IupNextField** and **IupPreviousField**. To remove the Tab stop from a control use the CANFOCUS attribute.

Arrows can also be used for navigation between buttons and toggles. This is necessary because when an **IupToggle** is inside an **IupRadio** the "Tab" keys will navigate only to the selected toggle.

In Windows, the focus feedback only appears after the user presses a key (except for the **IupText** where the feedback is the caret). Before pressing a key if you click in a control the focus feedback will be NOT be shown although it will be in focus. **IupMatrix** and other additional controls will always show their focus feedback. In GTK and Motif the focus feedback is always shown for the control that has the focus.

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

Usualy the application will process keyboard input in the **IupCanvas** 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, it is called only when the key is pressed. 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 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 somes 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 **iup_isXkey(key)** macro informs if a given key is an extended code. The **iup_isShiftXkey(key)** macro informs if a given key is an extended code using the Shift modifier, the **iup_isCtrlXkey(key)** macro for the Ctrl modifier, the **iup_isAltXkey(key)** macro for the Alt modifier, and the **iup_isSysXkey(key)** macro for the Sys modifier. These macros are also available in Lua as a function with the same name (iup.isprint(key), iup.isXkey(key), and so on).

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: 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) -> (next: ihandle) [in Lua]
```

ih: identifier of the interface element.

Returns: the element that received the focus or NULL if not found.

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) -> (previous: ihandle) [in Lua]
```

ih: identifier of the interface element.

Returns: the element that received the focus or NULL if not found.

See Also

IupNextField.

IupGetFocus

Returns the identifier of the interface element that has the keyboard focus, i.e. the element that will receive keyboard events.

Parameters/Return

```
Ihandle* IupGetFocus(void); [in C]
iup.GetFocus() -> elem: ihandle [in Lua]
```

See Also

IupSetFocus

IupSetFocus

Defines the interface element that will receive the keyboard focus, i.e., the element that will receive keyboard events. But this will be processed only after the con

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.

Notes

The value returned by **IupGetFocus** will be updated only after the main loop regain the control and the control actually receive the focus. So if you call **IupGetFocus** right after **IupSetFocus** the return value will be different. You could call **IupFlush** between the two functions to obtain the same value.

See Also

IupGetFocus.

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

AUTOTOGGLE (non inheritable): enables the automatic toggle of VALUE state when the item is activated. Default: NO. (since 3.0)

KEY (non inheritable): Underlines a key character in the submenu title. It is updated only when TITLE is updated. **Deprecated (since 3.0)**, use the mnemonic support directly in the TITLE attribute.

HIDEMARK [Motif and GTK Only]: If enabled the item cannot be checked, since the check box will not be shown. If all items in a menu enable it, then no empty space will be shown in front of the items. Normally the unmarked check box will not be shown, but since GTK 2.14 the unmarked check box is always shown. If your item will not be marked you must set HIDEMARK=YES, since this is the most common case we changed the default value to YES for this version of GTK, but if VALUE is defined the default goes back to NO. Default: NO. (since 3.0)

IMAGE [Windows and GTK Only] (non inheritable): Image name of the check mark image when VALUE=OFF. In Windows, an item in a menu bar cannot have a check mark. Ignored if item in a menu bar. A recommended size would be 16x16 to fit the image in the menu item. In Windows, if larger than the check mark area it will be cropped.

IMPRESS [Windows and GTK Only] (non inheritable): Image name of the check mark image when VALUE=ON.

TITLE (non inheritable): Item text. The "&" character can be used to define a mnemonic, the next character will be used as key. Use "&&" to show the "&" character instead on defining a mnemonic. When in a menu bar an item that has a mnemonic can be activated from any control in the dialog using the "Alt+key" combination.

The text also 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", but notice that the shortcut key (also known as Accelerator or Hot Key) still has to be implemented. To implement a shortcut use the K_* callbacks in the dialog.

TITLEIMAGE (non inheritable): Image name of the title image. In Windows, it appears before of the title text and after the check mark area (so both title and title image can be visible). In Motif, it must be at least defined during map, it replaces the text, and only images will be possible to set (TITLE will be hidden). In GTK, it will appear on the check mark area. (since 3.0)

VALUE (non inheritable): Indicates the item's state. When the value is ON, a mark will be displayed to the left of the item. Default: OFF. An item in a menu bar cannot have a check mark. When IMAGE is used, the checkmark is not shown. See the item AUTOTOGGLE attribute and the menu RADIO attribute.

WID (non inheritable): In Windows, returns the HMENU of the parent menu.

ACTIVE: also accepted.

Callbacks

ACTION: Action generated when the item is selected. IUP_CLOSE will be processed. Even if inside a popup menu when IUP_CLOSE is returned, the current popup dialog or the main loop will be closed.

HIGHLIGHT_CB: Action generated when the item is highlighted.

MAP_CB, UNMAP_CB, HELP_CB: common callbacks are supported.

Notes

Menu items are activated using the Enter key.

In Motif and GTK, the text font will be affected by the dialog font when the menu is mapped.

Since GTK 2.14 to have a menu item that can be marked you must set the VALUE attribute to ON or OFF, or set HIDEMARK=NO, before mapping the control.

Examples

Browse for Example Files

See the **IupMenu** element for screenshots.

See Also

IupSeparator, IupSubmenu, IupMenu.

IupMenu

Creates a menu element, which groups 3 types of interface elements: item, submenu and separator. Any other interface element defined inside a menu will be an error.

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.

Attributes

BGCOLOR: the background color of the menu, affects all items in the menu. (since 3.0)

RADIO (non inheritable): enables the automatic toggle of one child item. When a child item is selected the other item is automatically deselected. The menu acts like a **IupRadio** for its children. Submenus and their children are not affected.

WID (non inheritable): In Windows, returns the HMENU of the menu.

Callbacks

OPEN_CB: Called just before the menu is opened.

MENUCLOSE_CB: Called just after the menu is closed.

MAP_CB, UNMAP_CB: common callbacks are supported.

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**. If you replace a menu bar of a dialog, the previous menu is unmapped.

Any item inside a menu bar can retrieve attributes from the dialog using **IupGetAttribute**. It is not necessary to call **IupGetDialog**.

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"}
}
```

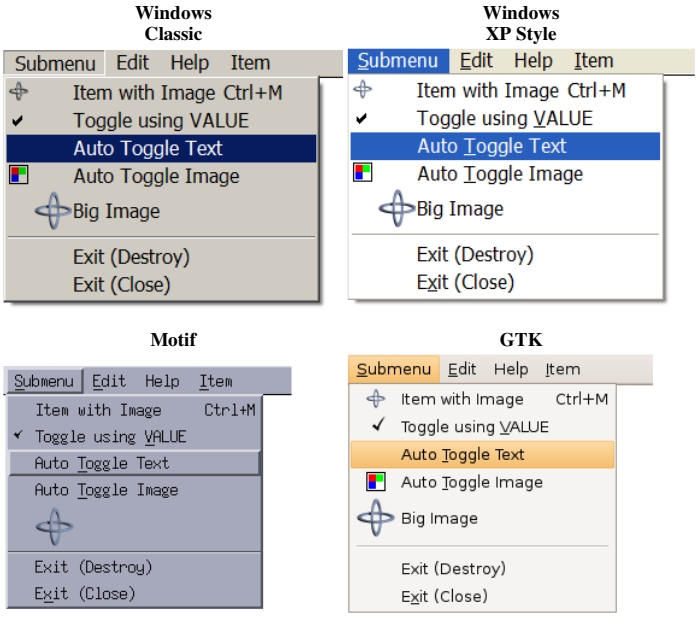
The same example using the cleaner syntax:

```
mnu = iup.menu
{
  {
```

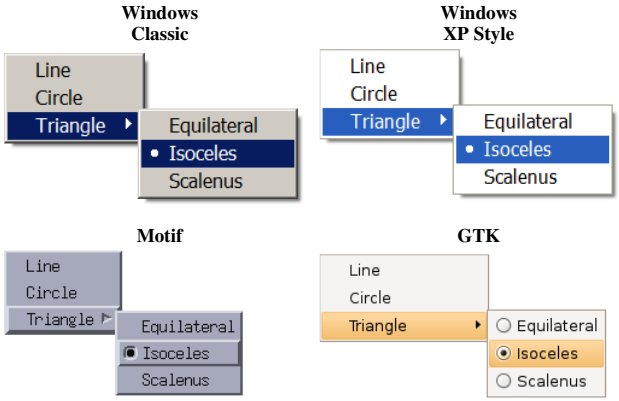
```
    "IupSubMenu 1",
    iup.menu
    {
      {"IupItem 1 Checked";value="ON"},
      {},
      {"IupItem 2 Disabled";active="NO"}
    }
  },
  {"IupItem 3"},
  {"IupItem 4"}
}
```

Examples

Browse for Example Files



The **IupItem** check is affected by the RADIO attribute in **IupMenu**:



See Also

IupDialog, IupItem, IupSeparator, IupSubmenu, IupPopup, IupDestroy

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 for Example Files

See Also

IupItem, IupSubMenu, IupMenu.

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 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

IMAGE [Windows and GTK Only] (non inheritable): Image name of the submenu image. In Windows, an item in a menu bar cannot have a check mark. Ignored if submenu in a menu bar. A recommended size would be 16x16 to fit the image in the menu item. In Windows, if larger than the check mark area it will be cropped. (since 3.0)

KEY (non inheritable): Underlines a key character in the submenu title. It is updated only when TITLE is updated. **Deprecated**, use the mnemonic support directly in the TITLE attribute.

TITLE (non inheritable): Submenu Text. The "&" character can be used to define a mnemonic, the next character will be used as key. Use "&&" to show the "&" character instead on defining a mnemonic.

WID (non inheritable): In Windows, returns the HMENU of the parent menu and it is actually created only when its child menu is mapped.

ACTIVE: also accepted.

Callbacks

HIGHLIGHT_CB: Action generated when the submenu is highlighted.

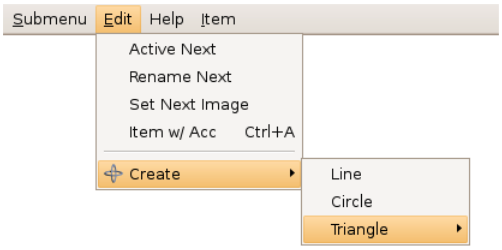
MAP_CB, UNMAP_CB: common callbacks are supported.

Notes

In Motif and GTK, the text font will be affected by the dialog font when the menu is mapped.

Examples

Browse for Example Files



See the **IupMenu** element for more screenshots.

See Also

IupItem, IupSeparator, IupMenu.

KEY

Associates a key to a menu item or submenu. Such key works as a shortcut when the menu is open, this is not a hot key.

Value

String containing a key description. Its is a string representation of an IUP key code. Please refer to the Keyboard Codes table for a list of the possible values.

Default: NULL

Notes

IUP automatically underlines the first appearance of the chosen menu letter. For such, the chosen letter must necessarily be a part of the menu text.

In Windows, when used will also set an underscore on the respective letter of the submenu title.

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.

In the menu bar, some systems automatically associate the ALT+<letter> combination for the chosen letter. This is valid for the Windows driver, but not for the Motif driver.

Be careful not to misuse this attribute in relation to K_ANY or K_* callbacks.

Affects

IupItem, IupSubMenu.

HIGHLIGHT_CB

Callback triggered every time the user selects an **IupItem** or **IupSubmenu**.

Callback

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

ih: identifier of the element that activated the event.

Affects

IupItem, IupSubmenu

OPEN_CB

Called just before the menu is opened.

Callback

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

ih: identifier of the element that activated the event.

Affects

IupMenu

MENUCLOSE_CB

Called just after the menu is closed.

Callback

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

ih: identifier of the element that activated the event.

Affects

IupMenu

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.

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.

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. Only the list of names need to be allocated. Each name will point to an internal string.
max_n: maximum number of names the table can receive.

Returns: the number of names loaded to the table. If names==NULL or max_n==0 then returns the actual number of names.

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. Only the list of names need to be allocated. Each name will point to an internal string.
max_n: maximum number of names the table can receive.

Returns: the number of names loaded to the table. If names==NULL or max_n==0 then returns the actual number of names.

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.

IupClipboard (since 3.0)

Creates an element that allows access to the clipboard. Each clipboard should be destroyed using IupDestroy, but you can use only one for entire application because it does not store any data inside. Or you can simply create and destroy everytime you need to copy or paste.

Creation

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

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

Attributes

IMAGE (write-only): name of an image to copy (set) or paste (get), to or from the clipboard. (GTK 2.6)

IMAGEAVAILABLE (read-only): informs if there is any image available at the clipboard. (GTK 2.6)

NATIVEIMAGE: native handle of an image to copy or paste, to or from the clipboard. In Win32 is a **HANDLE** of a DIB. In GTK is a **GdkPixbuf***. In Motif is a **Pixmap**. The returned handle in a paste must be released after used (GlobalFree(handle), g_object_unref(pixbuf) or XFreePixmap(display, pixmap)). After copy, do NOT release the given handle. See IUP-IM Functions for utility functions on image native handles. (GTK 2.6)

TEXT: copy or paste text to or from the clipboard.

TEXTAVAILABLE (read-only): informs if there is any text available at the clipboard.

Examples

```
Ihandle* clipboard = IupClipboard();
IupSetAttribute(clipboard, "TEXT", IupGetAttribute(text, "VALUE"));
IupDestroy(clipboard);
```

```
Ihandle* clipboard = IupClipboard();
IupSetAttribute(text, "VALUE", IupGetAttribute(clipboard, "TEXT"));
IupDestroy(clipboard);
```


IupTimer

Creates a timer which periodicaly 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.
>
> Returns: IUP_CLOSE will be processed.

Examples

Browse for Example Files

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]
iup.user{} -> (elem: ihandle) [in Lua]
user() [in LED]
```

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

Attributes

CLEARATTRIBUTES (write-only, non inheritable): it will clear all attributes stored internally and remove all references. (since 3.0)

IupHelp

Opens the given URL. In UNIX executes Netscape, Safari (MacOS) or Firefox (in Linux) passing the desired URL as a parameter. In Windows executes the shell "open" operation on the given URL.

In UNIX you can change the used browser setting the environment variable IUP_HELPAPP or using the global attribute "HELPAPP".

It is a non synchronous operation, i.e. the function will return just after execute the command and it will not wait for its result.

Parameters/Return

```
int IupHelp(const 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 (deprecated since 3.0, will be removed in a future version)

Since IUP 3.0, IupText and IupMatrix have several MASK* attributes to support masks. See the MASK attribute for more information.

Functions to associate an input mask to a **IupText** or a **IupMatrix** element.

These functions are included in the Controls Library.

Functions

```
int iupMaskSet(Ihandle *ih, const char *mask, int autofill, int casei);
int iupMaskMatSet(Ihandle *ih, const char *mask, int autofill, int casei, int lin, int col);
```

These functions are responsible for setting the mask to be used.

ih: 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 (NOT supported since 3.0)
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 *ih, int autofill, int min, int max);
int iupMaskSetFloat(Ihandle *ih, int autofill, float min, float max);
int iupMaskMatSetInt(Ihandle *ih, int autofill, int min, int   max, int lin, int col);
int iupMaskMatSetFloat(Ihandle *ih, 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.

ih: 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 (NOT supported since 3.0)
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 *ih);
int iupMaskMatRemove(Ihandle *ih, int lin, int col);
```

These functions are responsible for removing the mask from the control.

ih: Ihandle of IupText or IupMatrix
lin, col: Line and column numbers in the matrix

```
int iupMaskCheck (Ihandle *ih);
int iupMaskMatCheck(Ihandle *ih, int lin, int col);
```

These functions verify if what was typed by the user is valid for the defined mask. For IupMatrix they work only when in edition mode.

ih: 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 *ih, char **val);
int iupMaskGetFloat(Ihandle *ih, float *fval);
int iupMaskGetInt(Ihandle *ih, int *ival);
int iupMaskMatGet(Ihandle *ih, char **val, int lin, int col);
int iupMaskMatGetFloat(Ihandle *ih, float  *fval, int lin, int col);
int iupMaskMatGetDouble(Ihandle *ih, double *dval, int lin, int col);
int iupMaskMatGetInt(Ihandle *ih, 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. For IupMatrix they work only when in edition mode.

ih: Ihandle of IupText or IupMatrix
val, fval, ival: Pointers used to complete the return value
lin, col: Line and column numbers in the matrix.

They return 1 if the text satisfies the mask, or 0 otherwise.

Notes

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 for Example Files

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

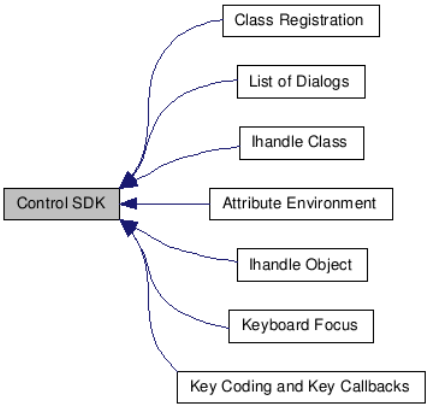
- __IUPXXX_H (same file name, upper case, "__" preffix and replace "." by "_")

Documentation

- In the header, using Doxygen commands.
- Check an existant header for example.

Control SDK

Collaboration diagram for Control SDK:



Modules

Attribute Environment
Ihandle Class
List of Dialogs
Keyboard Focus
Key Coding and Key Callbacks
Ihandle Object
Class Registration

Detailed Description

Control Creation Guide

Generated on Thu Oct 1 14:02:32 2009 for IUP by doxygen *1.6.1*

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 Iup**Xxx**Open(void)
{
  iupRegisterClass(iupXxxGetClass());
}
```

The function iupXxxGetClass 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* iupXxxGetClass(void)
{
  /* (1) - initialize the class */
  Iclass* ic = iupClassNew(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 = iXxxCreateMethod;
  ic->Map = iXxxMapMethod;
  ic->Destroy = iXxxDestroyMethod;
  ic->ComputeNaturalSize = iXxxComputeNaturalSizeMethod;
  ic->SetCurrentSize = iXxxSetCurrentSizeMethod;
  ic->LayoutUpdate = iXxxLayoutUpdateMethod;
  ic->DisplayUpdate = iXxxDisplayUpdateMethod;
  ic->SetPosition = iXxxSetPositionMethod;
```

```
  /* (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, NULL, IUP_NOT_MAPPED, IUP_NO_INHERIT);
  iupClassRegisterAttribute(ic, "RASTERSIZE", iupGetRasterSizeAttrib, iupDlgSetRastersizeAttrib, NULL, IUP_NOT_MAPPED, IUP_NO_INHERIT);
  iupClassRegisterAttribute(ic, "WID", iupGetWidAttrib, iupNoSetAttrib, NULL, IUP_MAPPED, IUP_NO_INHERIT);
  iupClassRegisterAttribute(ic, "FONT", NULL, iupdrvSetFontAttrib, IupGetGlobal("DEFAULTFONT"), IUP_NOT_MAPPED, IUP_NO_INHERIT);

  /* Common, but only after Map */
  iupClassRegisterAttribute(ic, "ACTIVE", iupGetActiveAttrib, iupSetActiveAttrib, "YES", IUP_MAPPED, IUP_INHERIT);
  iupClassRegisterAttribute(ic, "VISIBLE", iupGetVisibleAttrib, iupSetVisibleAttrib, "YES", IUP_MAPPED, IUP_NO_INHERIT);
  iupClassRegisterAttribute(ic, "ZORDER", NULL, iupdrvSetZorderAttrib, NULL, IUP_MAPPED, IUP_NO_INHERIT);

  /* only the default value. */
  iupClassRegisterAttribute(ic, "BORDER", NULL, NULL, "YES", IUP_NOT_MAPPED, 0);

  return ic;
}
```

You can use the iupXxxGetClass equivalent function of other controls to initialize a new base class for a new control that inherits the functionalities of the base class.

You can also use the Base Class methods and attribute functions to simplify your iupXxxGetClass.

If the control is a native control then it usually will have separate modules for each driver. The iupXxxGetClass function could call a iupdrvXxxInitClass(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 pratice to have a convenience function to create the control:

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

Control Exported Functions

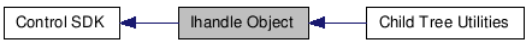
The file header with the exported functions should look like this:

```
#ifndef __IUPXXX_H
#define __IUPXXX_H

#ifdef __cplusplus
extern "C" {
#endif

void IupXxxOpen(void);

Ihandle* IupXxx(void);

#ifdef __cplusplus
}
#endif

#endif
```

Ihandle Object

[Control SDK]

Collaboration diagram for Ihandle Object:



Data Structures

struct Ihandle_

Modules

Child Tree Utilities

Defines

#define IUP_EXPAND_WIDTH (IUP_EXPAND_W1 | IUP_EXPAND_W0)
#define IUP_EXPAND_HEIGHT (IUP_EXPAND_H1 | IUP_EXPAND_H0)
#define IUP_EXPAND_BOTH (IUP_EXPAND_WIDTH | IUP_EXPAND_HEIGHT)
#define iupALLOCCTRLDATA() ((IcontrolData*)calloc(1, sizeof(IcontrolData)))

Typedefs

typedef struct _InativeHandle InativeHandle
typedef struct _IcontrolData IcontrolData

Enumerations

enum Iexpand {
IUP_EXPAND_NONE = 0x00, **IUP_EXPAND_H0** = 0x01, **IUP_EXPAND_H1** = 0x02, **IUP_EXPAND_W0** = 0x04,
IUP_EXPAND_W1 = 0x08
}

Functions

void ** iupObjectGetParamList (void *first, va_list arglist)
int iupObjectCheck (Ihandle *ih)

Detailed Description

Object handle for all the elements.

See iup_object.h

Define Documentation

#define IUP_EXPAND_WIDTH (IUP_EXPAND_W1 | IUP_EXPAND_W0)

Expand configuration

#define IUP_EXPAND_HEIGHT (IUP_EXPAND_H1 | IUP_EXPAND_H0)

Expand configuration

#define IUP_EXPAND_BOTH (IUP_EXPAND_WIDTH | IUP_EXPAND_HEIGHT)

Expand configuration

#define iupALLOCCTRLDATA ( ) ((IcontrolData*)calloc(1, sizeof(IcontrolData)))

IcontrolData allocation utility.

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

enum Iexpand

Expand configuration

Function Documentation

void** iupObjectGetParamList (void * *first*,
va_list *arglist*
)

Utility that 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.

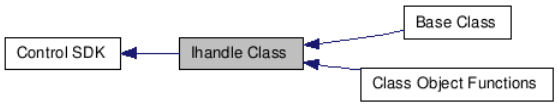
int iupObjectCheck (Ihandle * *ih*)

Checks if the handle is still valid based on the signature.

Ihandle Class

[Control SDK]

Collaboration diagram for Ihandle Class:



Data Structures

struct Iclass_

Modules

Class Object Functions
Base Class

Typedefs

typedef enum _InativeType InativeType
typedef enum _IchildType IchildType
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)
typedef enum _IattribFlags IattribFlags

Enumerations

enum _InativeType {
IUP_TYPEVOID, IUP_TYPECONTROL, IUP_TYPECANVAS, IUP_TYPEDIALOG,
IUP_TYPEIMAGE, IUP_TYPEMENU
}
enum _IchildType { **IUP_CHILDNONE**, **IUP_CHILD_ONE**, **IUP_CHILDMANY** }
enum _IattribFlags {
IUPAF_DEFAULT = 0, IUPAF_NO_INHERIT = 1, IUPAF_NO_DEFAULTVALUE = 2, IUPAF_NO_STRING = 4,
IUPAF_NOT_MAPPED = 8, IUPAF_HAS_ID = 16, IUPAF_READONLY = 32, IUPAF_WRITEONLY = 64
}

Functions

Iclass * iupClassNew (Iclass *ic_parent)
void iupClassRelease (Iclass *ic)
void iupClassRegisterAttribute (Iclass *ic, const char *name, IattribGetFunc get, IattribSetFunc set, const char *default_value, const char *system_default, int flags)
void iupClassRegisterAttributeId (Iclass *ic, const char *name, IattribGetIdFunc get, IattribSetIdFunc set, int flags)
void iupClassRegisterGetAttribute (Iclass *ic, const char *name, IattribGetFunc *get, IattribSetFunc *set, const char **default_value, const char **system_default, int *flags)
void iupClassRegisterCallback (Iclass *ic, const char *name, const char *format)
char * iupClassCallbackGetFormat (Iclass *ic, const char *name)

Detailed Description

See iup_class.h

Typedef Documentation

typedef enum _InativeType InativeType

Known native types.

typedef enum _IchildType IchildType

Possible number of children.

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 has_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 hash table (except inheritble attributes that are always stored in the hash table).
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 has_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, ex: "1" = "IDVALUE1". Used by iupClassRegisterAttribute.

typedef enum _IattribFlags IattribFlags

Attribute flags. Used by iupClassRegisterAttribute.

Enumeration Type Documentation

enum _InativeType

Known native types.

Enumerator:

IUP_TYPEVOID	No native representation - HBOX, VBOX, ZBOX, FILL, RADIO (handle==(void*)-1 always)
IUP_TYPECONTROL	Native controls - BUTTON, LABEL, TOGGLE, LIST, TEXT, MULTILINE, FRAME, others
IUP_TYPECANVAS	Drawing canvas, also used as a base control for custom controls.
IUP_TYPEDIALOG	DIALOG
IUP_TYPEIMAGE	IMAGE
IUP_TYPEMENU	MENU, SUBMENU, ITEM, SEPARATOR

enum _IchildType

Possible number of children.

enum _IattribFlags

Attribute flags. Used by iupClassRegisterAttribute.

Enumerator:

IUPAF_DEFAULT	inheritable, can has a default value, is a string, can call the set/get functions only if mapped, no ID
IUPAF_NO_INHERIT	is not inheritable
IUPAF_NO_DEFAULTVALUE	can not has a default value
IUPAF_NO_STRING	is not a string
IUPAF_NOT_MAPPED	will call the set/get functions also when not mapped
IUPAF_HAS_ID	can has an ID at the end of the name, automatically set by iupClassRegisterAttributeId
IUPAF_READONLY	is read-only, can not be changed
IUPAF_WRITEONLY	is write-only, usually an action

Function Documentation

Iclass* iupClassNew (Iclass * *ic_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. Calls the Iclass::Release method.
Called from iupRegisterFinish.

```
void iupClassRegisterAttribute ( Iclass *          ic,
                                 const char *      name,
                                 IattribGetFunc    get,
                                 IattribSetFunc    set,
                                 const char *      default_value,
                                 const char *      system_default,
                                 int               flags
                               )
```

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, is inheritable can has a default value and is a string.
Since there is only one attribute function table per class tree, if you register the same attribute in a child class, then it will replace the parent registration.
If an attribute is not inheritable or not a string then it MUST be registered. Internal attributes (starting with "_IUP") can never be registered.

```
void iupClassRegisterAttributeId ( Iclass *          ic,
                                   const char *      name,
                                   IattribGetIdFunc  get,
                                   IattribSetIdFunc  set,
                                   int               flags
                                 )
```

Same as iupClassRegisterAttribute for attributes with Ids.

```
void iupClassRegisterGetAttribute ( Iclass *          ic,
                                    const char *      name,
                                    IattribGetFunc *  get,
                                    IattribSetFunc *  set,
                                    const char **     default_value,
                                    const char **     system_default,
                                    int *             flags
                                  )
```

Returns the attribute handling functions.

```
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
- "v" = (void*) - generic pointer
- "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.

Class Object Functions

[Ihandle Class]

Collaboration diagram for Class Object Functions:

Functions

int iupClassObjectCreate (Ihandle *ih, void **params)
int iupClassObjectMap (Ihandle *ih)
void iupClassObjectUnMap (Ihandle *ih)
void iupClassObjectDestroy (Ihandle *ih)
Ihandle * iupClassObjectGetInnerContainer (Ihandle *ih)
void * iupClassObjectGetInnerNativeContainerHandle (Ihandle *ih, Ihandle *child)
void iupClassObjectChildAdded (Ihandle *ih, Ihandle *child)
void iupClassObjectChildRemoved (Ihandle *ih, Ihandle *child)
void iupClassObjectLayoutUpdate (Ihandle *ih)
void iupClassObjectComputeNaturalSize (Ihandle *ih, int *w, int *h, int *children_expand)
void iupClassObjectSetChildrenCurrentSize (Ihandle *ih, int shrink)
void iupClassObjectSetChildrenPosition (Ihandle *ih, int x, int y)
int iupClassObjectDlgPopup (Ihandle *ih, int x, int y)

Detailed Description

Stubs for the class methods. They implement inheritance and check if method is NULL.

See iup_class.h

Function Documentation

int iupClassObjectCreate (Ihandle * *ih*,
void ** *params*
)

Calls Iclass::Create method.

int iupClassObjectMap (Ihandle * *ih*)

Calls Iclass::Map method.

void iupClassObjectUnMap (Ihandle * *ih*)

Calls Iclass::UnMap method.

void iupClassObjectDestroy (Ihandle * *ih*)

Calls Iclass::Destroy method.

Ihandle* iupClassObjectGetInnerContainer (Ihandle * *ih*)

Calls Iclass::GetInnerContainer method. The parent class is ignored. If necessary the child class must handle the parent class internally.

void* iupClassObjectGetInnerNativeContainerHandle (Ihandle * *ih*,
Ihandle * *child*
)

Calls Iclass::GetInnerNativeContainerHandle method. Returns ih->handle if there is no inner parent. The parent class is ignored. If necessary the child class must handle the parent class internally.

void iupClassObjectChildAdded (Ihandle * *ih*,
Ihandle * *child*
)

Calls Iclass::ChildAdded method.

void iupClassObjectChildRemoved (Ihandle * *ih*,
Ihandle * *child*
)

Calls Iclass::ChildRemoved method.

void iupClassObjectLayoutUpdate (Ihandle * *ih*)

Calls Iclass::LayoutUpdate method.

void iupClassObjectComputeNaturalSize (Ihandle * *ih*,
int * *w*,
int * *h*,
int * *children_expand*
)

Calls Iclass::ComputeNaturalSize method.

void iupClassObjectSetChildrenCurrentSize (Ihandle * *ih*,
int *shrink*
)

Calls Iclass::SetChildrenCurrentSize method.

void iupClassObjectSetChildrenPosition (Ihandle * *ih*,
int *x*,
int *y*
)

Calls Iclass::SetChildrenPosition method.

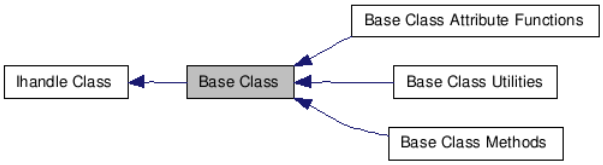
int iupClassObjectDlgPopup (Ihandle * *ih*,
int *x*,
int *y*
)

Calls Iclass::DlgPopup method.

Base Class

[Ihandle Class]

Collaboration diagram for Base Class:



Modules

- Base Class Methods
- Base Class Attribute Functions
- Base Class Utilities

Functions

void iupBaseRegisterCommonAttrib (Iclass *ic)
void iupBaseRegisterVisualAttrib (Iclass *ic)
void iupBaseRegisterCommonCallbacks (Iclass *ic)
void iupBaseContainerUpdateExpand (Ihandle *ih)
void iupBaseComputeNaturalSize (Ihandle *ih)
void iupBaseSetCurrentSize (Ihandle *ih, int w, int h, int shrink)

void iupBaseSetPosition (Ihandle *ih, int x, int y)

Detailed Description

See iup_classbase.h

Function Documentation

void iupBaseRegisterCommonAttrib (Iclass * *ic*)

Register all common base attributes:
WID
SIZE, RASTERSIZE, POSITION
FONT (and derived)

All controls that are positioned inside a dialog must register all common base attributes.

void iupBaseRegisterVisualAttrib (Iclass * *ic*)

Register all visual base attributes:
VISIBLE, ACTIVE
ZORDER, X, Y
TIP (and derived)

All controls that are positioned inside a dialog must register all visual base attributes.

void iupBaseRegisterCommonCallbacks (Iclass * *ic*)

Register all common callbacks:
MAP_CB, UNMAP_CB, GETFOCUS_CB, KILLFOCUS_CB, ENTERWINDOW_CB, LEAVEWINDOW_CB, K_ANY, HELP_CB.

void iupBaseContainerUpdateExpand (Ihandle * *ih*)

Updates the expand member of the IUP object from the EXPAND attribute. Should be called in the beginning of the ComputeNaturalSize for a container.

void iupBaseComputeNaturalSize (Ihandle * *ih*)

Initializes the natural size using the user size, then if a container then update the "expand" member from the EXPAND attribute, then call iupClassObjectComputeNaturalSize for containers if they have children or call iupClassObjectComputeNaturalSize for non-containers if user size is not defined. Must be called for each children in the container.
First call is in iupLayoutCompute.

```
void iupBaseSetCurrentSize ( Ihandle * ih,
                             int       w,
                             int       h,
                             int       shrink
                           )
```

Update the current size from the available size, the natural size, expand and shrink. Call iupClassObjectSetChildrenCurrentSize for containers if they have children. Must be called for each children in the container.
First call is in iupLayoutCompute.

```
void iupBaseSetPosition ( Ihandle * ih,
                          int       x,
                          int       y
                        )
```

Set the current position and update children position for containers. Call iupClassObjectSetChildrenPosition for containers if they have children. Must be called for each children in the container.
First call is in iupLayoutCompute.

Base Class Methods

[Base Class]

Collaboration diagram for Base Class Methods:

Functions

void iupdrvBaseLayoutUpdateMethod (Ihandle *ih)
void iupdrvBaseUnMapMethod (Ihandle *ih)
int iupBaseTypeVoidMapMethod (Ihandle *ih)

Detailed Description

See iup_classbase.h

Function Documentation

void iupdrvBaseLayoutUpdateMethod (Ihandle * *ih*)

Driver dependent Iclass::LayoutUpdate method.

void iupdrvBaseUnMapMethod (Ihandle * *ih*)

Driver dependent Iclass::UnMap method.

int iupBaseTypeVoidMapMethod (Ihandle * *ih*)

Native type void Iclass::Map method.

Base Class Attribute Functions

[Base Class]

Collaboration diagram for Base Class Attribute Functions:

Functions

char * **iupBaseGetWidAttrib** (Ihandle *ih)
int **iupBaseSetNameAttrib** (Ihandle *ih, const char *value)
int **iupBaseSetRasterSizeAttrib** (Ihandle *ih, const char *value)
int **iupBaseSetSizeAttrib** (Ihandle *ih, const char *value)
char * **iupBaseGetSizeAttrib** (Ihandle *ih)
char * **iupBaseGetRasterSizeAttrib** (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)
char * **iupdrvBaseGetXAttrib** (Ihandle *ih)
char * **iupdrvBaseGetYAttrib** (Ihandle *ih)
int **iupdrvBaseSetTipAttrib** (Ihandle *ih, const char *value)
int **iupdrvBaseSetTipVisibleAttrib** (Ihandle *ih, const char *value)
int **iupdrvBaseSetBgColorAttrib** (Ihandle *ih, const char *value)
int **iupdrvBaseSetFgColorAttrib** (Ihandle *ih, const char *value)
char * **iupBaseNativeParentGetBgColorAttrib** (Ihandle *ih)
char * **iupBaseContainerGetExpandAttrib** (Ihandle *ih)
int **iupdrvBaseSetCursorAttrib** (Ihandle *ih, const char *value)
char * **iupdrvBaseGetClientSizeAttrib** (Ihandle *ih)
char * **iupdrvBaseGetTitleAttrib** (Ihandle *ih)
int **iupdrvBaseSetTitleAttrib** (Ihandle *ih, const char *value)

Detailed Description

Used by the controls for iupClassRegisterAttribute.

See iup_classbase.h

Base Class Utilities

[Base Class]

Collaboration diagram for Base Class Utilities:

Defines

#define **iupMAX**(_a, _b) ((_a)>(_b)?(_a):(_b))
#define **iupROUND**(_x) ((int)((_x)>0? (_x)+0.5: (_x)-0.5))
#define **iupCOLOR8TO16**(_x) ((unsigned short)(_x*257))
#define **iupCOLOR16TO8**(_x) ((unsigned char)(_x/257))
#define **iupBYTECROP**(_x) ((unsigned char)((_x)<0?0:((_x)>255)?255:(_x)))

#define **IUP_ALIGN_ABOTTOM** IUP_ALIGN_ARIGHT
#define **IUP_ALIGN_ATOP** IUP_ALIGN_ALEFT

Enumerations

enum { **IUP_ALIGN_ALEFT**, **IUP_ALIGN_ACENTER**, **IUP_ALIGN_ARIGHT** }
enum { **IUP_SB_NONE**, **IUP_SB_HORIZ**, **IUP_SB_VERT** }

Functions

int **iupBaseGetScrollbar** (Ihandle *ih)
char * **iupBaseNativeParentGetBgColor** (Ihandle *ih)
void **iupBaseCallValueChangedCb** (Ihandle *ih)

Detailed Description

See iup_classbase.h

Class Registration

[Control SDK]

Collaboration diagram for Class Registration:

Functions

Iclass * iupRegisterFindClass (const char *name)
void iupRegisterClass (Iclass *ic)

Detailed Description

All controls are registered so the creation using IupCreate can work.

See iup_register.h

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]

Collaboration diagram for Attribute Environment:

Defines

#define iupAttribIsInternal(_name) ((_name[0] == '_' && _name[1] == 'I' && _name[2] == 'U' && _name[3] == 'P')? 1: 0)

Functions

int iupAttribIsPointer (Ihandle *ih, const char *name)
void iupAttribSetStr (Ihandle *ih, const char *name, const 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 * iupAttribGet (Ihandle *ih, const char *name)
char * iupAttribGetInherit (Ihandle *ih, const char *name)
char * iupAttribGetInheritNativeParent (Ihandle *ih, const char *name)
char * iupAttribGetStr (Ihandle *ih, const char *name)
int iupAttribGetInt (Ihandle *ih, const char *name)
int iupAttribGetBoolean (Ihandle *ih, const char *name)
float iupAttribGetFloat (Ihandle *ih, const char *name)
void iupAttribSetHandleName (Ihandle *ih)

Detailed Description

When attributes are not stored at the control they are stored in a hash table (see Hash Table).

As a general rule use:
- IupGetAttribute, IupSetAttribute, ... : when care about control implementation, hash table, inheritance and default value
- iupAttribGetStr,Int,Float: when care about inheritance, hash table and default value
- iupAttribGet,... : ONLY access the hash table These different functions have very different performances and results. So use them wiselly.

See iup_attrib.h

Define Documentation

#define iupAttribIsInternal (_name) ((_name[0] == '_' && _name[1] == 'I' && _name[2] == 'U' && _name[3] == 'P')? 1: 0)

Returns true if the attribute name if in the internal format "_IUP...".

Function Documentation

int iupAttribIsPointer (Ihandle * *ih*,
const char * *name*
)

Returns true if the attribute name is a known pointer.

void iupAttribSetStr (Ihandle * *ih*,
const char * *name*,
const char * *value*
)

Sets the attribute only in the hash table as a pointer. It ignores children.

void iupAttribStoreStr (Ihandle * *ih*,
const char * *name*,
const char * *value*
)

Sets the attribute only in the hash table 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 hash table as a string. The string is internally duplicated. Use same format as sprintf. It ignores children.

void iupAttribSetInt (Ihandle * *ih*,
const char * *name*,
int *num*
)

Sets an integer attribute only in the hash table. 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 hash table. It will be stored as a string. It ignores children.

char* iupAttribGet (Ihandle * *ih*,
const char * *name*
)

Returns the attribute from the hash table only.

char* iupAttribGetInherit (Ihandle * *ih*,
const char * *name*
)

Returns the attribute from the hash table only, but if not defined then checks in its parent tree.

char* iupAttribGetInheritNativeParent (Ihandle * *ih*,
const char * *name*
)

Returns the attribute from the hash table of a native parent. Don't check for default values. Don't check at the element. Used for BGCOLOR and BACKGROUND attributes.

```
char* iupAttribGetStr ( Ihandle *      ih,
                        const char *  name
                      )
```

Returns the attribute from the hash table as a string, but if not defined then checks in its parent tree if allowed by the control implementation, if still not defined then returns the registered default value if any.

```
int iupAttribGetInt ( Ihandle *      ih,
                      const char *  name
                    )
```

Same as iupAttribGetStr but returns an integer number. Checks also for boolean values.

```
int iupAttribGetBoolean ( Ihandle *      ih,
                          const char *  name
                        )
```

Same as iupAttribGetStr but checks for boolean values. Use iupStrBoolean.

```
float iupAttribGetFloat ( Ihandle *      ih,
                          const char *  name
                        )
```

Same as iupAttribGetStr but returns an floating point number.

```
void iupAttribSetHandleName ( Ihandle *  ih  )
```

Set an internal name to a handle.

Child Tree Utilities

[Ihandle Object]

Collaboration diagram for Child Tree Utilities:

Functions

Ihandle * iupChildTreeGetNativeParent (Ihandle *ih)
InativeHandle * iupChildTreeGetNativeParentHandle (Ihandle *ih)
void iupChildTreeAppend (Ihandle *parent, Ihandle *child)

Detailed Description

Some native containers have an internal native child that will be the actual container of the children. This native container is returned by iupClassObjectGetInnerNativeContainerHandle.

Some native elements need an extra parent, the ih->handle points to the main element itself, NOT to the extra parent. This extra parent is stored as "_IUP_EXTRAPARENT".

See iup_childtree.h

Function Documentation

```
Ihandle* iupChildTreeGetNativeParent ( Ihandle *  ih  )
```

Returns the native parent. It simply excludes containers that are from IUP_TYPEVOID classes.

```
InativeHandle* iupChildTreeGetNativeParentHandle ( Ihandle *  ih  )
```

Returns the native parent handle. Uses iupChildTreeGetNativeParent and iupClassObjectGetInnerNativeContainerHandle.

```
void iupChildTreeAppend ( Ihandle *  parent,
                          Ihandle *  child
                        )
```

Adds the child directly to the parent tree.

Generated on Thu Oct 1 14:02:32 2009 for IUP by doxygen 1.6.1

List of Dialogs

[Control SDK]

Collaboration diagram for List of Dialogs:

Functions

void iupDlgListAdd (Ihandle *ih)
void iupDlgListRemove (Ihandle *ih)
Ihandle * iupDlgListFirst (void)
Ihandle * iupDlgListNext (void)
void iupDlgListVisibleInc (void)
void iupDlgListVisibleDec (void)
int iupDlgListVisibleCount (void)

Detailed Description

See iup_dlglist.h

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.

Generated on Thu Oct 1 14:02:33 2009 for IUP by doxygen 1.6.1

Keyboard Focus

[Control SDK]

Collaboration diagram for Keyboard Focus:

Functions

int iupFocusCanAccept (Ihandle *ih)
void iupCallGetFocusCb (Ihandle *ih)
void iupCallKillFocusCb (Ihandle *ih)
Ihandle * iupFocusNextInteractive (Ihandle *ih)

Detailed Description

See iup_focus.h

Function Documentation

int iupFocusCanAccept (Ihandle * *ih*)

Utility to check if a control can have the keyboard input focus. To receive the focus must be interactive, has CANFOCUS=YES, is mapped, is visible and is active.

void iupCallGetFocusCb (Ihandle * *ih*)

Call GETFOCUS_CB and FOCUS_CB.

void iupCallKillFocusCb (Ihandle * *ih*)

Call KILLFOCUS_CB and FOCUS_CB.

Ihandle* iupFocusNextInteractive (Ihandle * *ih*)

Returns the next interactive brother. Independs if it can receive the focus.

Generated on Thu Oct 1 14:02:33 2009 for IUP by doxygen *1.6.1*

Key Coding and Key Callbacks

[Control SDK]

Collaboration diagram for Key Coding and Key Callbacks:

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)
int iupKeyProcessNavigation (Ihandle *ih, int key, int shift)

Detailed Description

See iup_key.h

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 occoured.

int iupKeyCallKeyPressCb (Ihandle * *ih*,
int *code*,
int *press*
)

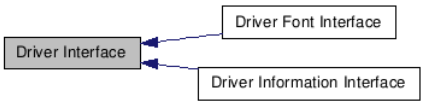
Calls the KEYPRESS_CB callback. Should be called when a keyboard event occoured.

int iupKeyProcessNavigation (Ihandle * *ih*,
int *key*,
int *shift*
)

Process Tab, DEFAULTENTER and DEFAULTESC in key press events.

Driver Interface

Collaboration diagram for Driver Interface:



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 iupdrvSetFocus (Ihandle *ih)
void iupdrvSetVisible (Ihandle *ih, int enable)
void iupdrvSetActive (Ihandle *ih, int enable)
void iupdrvDisplayUpdate (Ihandle *ih)
void iupdrvDisplayRedraw (Ihandle *ih)
void iupdrvReparent (Ihandle *ih)
void iupdrvDrawFocusRect (Ihandle *ih, void *gc, int x, int y, int w, int h)
int iupdrvGetScrollbarSize (void)
void iupdrvActivate (Ihandle *ih)
int iupdrvMenuGetMenuBarSize (Ihandle *ih)

Detailed Description

Each driver must export the symbols defined here.

See iup_drv.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 atribute, 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 releative.

int iupdrvIsVisible (Ihandle * *ih*)

Returns true if the element is visible.

int iupdrvIsActive (Ihandle * *ih*)

Returns true if the element is active.

void iupdrvSetFocus (Ihandle * *ih*)

Actually changes the focus to the given element.

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*)

Post a redraw of a control.

void iupdrvDisplayRedraw (Ihandle * *ih*)

Force a redraw of a control.

void iupdrvReparent (Ihandle * *ih*)

Reparent the native control.

void iupdrvDrawFocusRect (Ihandle * *ih*,
void * *gc*,
int *x*,
int *y*,
int *w*,
int *h*
)

Draws a focus rectangle

int iupdrvGetScrollbarSize (void)

Size of the scroolbar.

void iupdrvActivate (Ihandle * *ih*)

Activates a control.

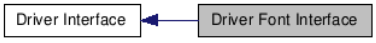
int iupdrvMenuGetMenuBarSize (Ihandle * *ih*)

Returns the height of a menu bar.

Driver Font Interface

[Driver Interface]

Collaboration diagram for Driver Font Interface:



Functions

void iupdrvFontGetCharSize (Ihandle *ih, int *charwidth, int *charheight)
int iupdrvFontGetStringWidth (Ihandle *ih, const char *str)
void iupdrvFontGetMultiLineStringSize (Ihandle *ih, const char *str, int *w, int *h)
char * iupdrvGetSystemFont (void)
int iupdrvSetStandardFontAttrib (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)

Detailed Description

Each driver must export the symbols defined here.

See iup_drvfont.h

Function Documentation

```
void iupdrvFontGetCharSize ( Ihandle *  ih,
                             int *      charwidth,
                             int *      charheight
                           )
```

Retrieve the character size for the selected font. Should be used only to calculate the SIZE attribute.

```
int iupdrvFontGetStringWidth ( Ihandle *      ih,
                               const char *  str
                             )
```

Retrieve the string width for the selected font.

```
void iupdrvFontGetMultiLineStringSize ( Ihandle *      ih,
                                        const char *  str,
                                        int *         w,
                                        int *         h
                                      )
```

Retrieve the multi-lined string size for the selected font.
Width is the maximum line width.
Height is charheight*number_of_lines (this will avoid line size variations).

```
char* iupdrvGetSystemFont ( void    )
```

Returns the System default font.

```
int iupdrvSetStandardFontAttrib ( Ihandle *      ih,
                                  const char *  value
                                )
```

STANDARDFONT 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]

Collaboration diagram for Driver Information Interface:

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)

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

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 DialogGetDecoration.

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*)

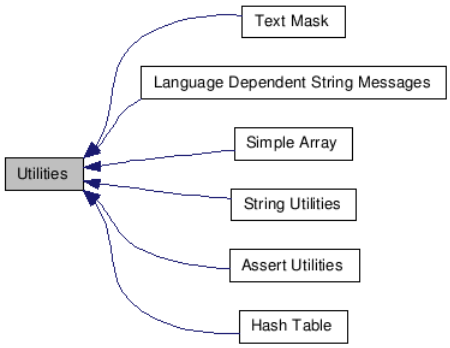
Returns true if the given name is an existant directory.

int iupdrvMakeDirectory (const char * *name*)

Creates a new direcotry.

Utilities

Collaboration diagram for Utilities:



Modules

Simple Array
Assert Utilities
Text Mask
String Utilities
Language Dependent String Messages
Hash Table

Simple Array

[Utilities]

Collaboration diagram for Simple Array:

Functions

Iarray * iupArrayCreate (int start_count, int elem_size)

void iupArrayDestroy (Iarray *iarray)
void * iupArrayGetData (Iarray *iarray)
void * iupArrayInc (Iarray *iarray)
void * iupArrayAdd (Iarray *iarray, int new_count)
int iupArrayCount (Iarray *iarray)

Detailed Description

Expandable array using a simple pointer.

See iup_array.h

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 * *iarray*)

Destroys the array.

void* iupArrayGetData (Iarray * *iarray*)

Returns the pointer that contains the array.

void* iupArrayInc (Iarray * *iarray*)

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 * *iarray*,
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.

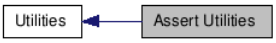
int iupArrayCount (Iarray * *iarray*)

Returns the actual number of elements in the array.

Assert Utilities

[Utilities]

Collaboration diagram for Assert Utilities:



Defines

#define iupASSERT(_expr) ((_expr)? (void)0: iupAssert(#_expr, __FILE__, __LINE__, NULL))
#define iupERROR(_msg) iupError(_msg)

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

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.

Generated on Thu Oct 1 14:02:32 2009 for IUP by doxygen *1.6.1*

String Utilities

[Utilities]

Collaboration diagram for String Utilities:

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)
int iupStrLineCount (const char *str)
const char * iupStrNextLine (const char *str, int *len)
int iupStrCountChar (const char *str, int c)
char * iupStrCopyUntil (char **str, int c)
void iupStrCopyN (char *dst_str, int dst_max_size, const char *src_str)
char * iupStrGetMemory (int size)
char * iupStrGetMemoryCopy (const char *str)
void iupStrLower (char *dstr, const char *sstr)
int iupStrToRGB (const char *str, unsigned char *r, unsigned char *g, unsigned char *b)
int iupStrToRGBA (const char *str, unsigned char *r, unsigned char *g, unsigned char *b, unsigned char *a)
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)
int iupStrReplace (char *str, char src, char dst)
void iupStrToUnix (char *str)
char * iupStrToMac (const char *str)
char * iupStrToDos (const char *str)
void iupStrRemove (char *value, int start, int end, int dir)
char * iupStrInsert (const char *value, const char *insert_value, int start, int end)
char * iupStrProcessMnemonic (const char *str, char *c, int action)

Detailed Description

See iup_str.h

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.

```
int iupStrLineCount ( const char *  str   )
```

Returns the number of lines in a string. It works for UNIX, DOS and MAC line ends.

```
const char* iupStrNextLine ( const char *  str,
                             int *         len
                           )
```

Returns the a pointer to the next line and the size of the current line. It works for UNIX, DOS and MAC line ends. The size does not includes the line end. If str is NULL it will return NULL.

```
int iupStrCountChar ( const char *  str,
                      int           c
                    )
```

Returns the number of repetitions of the character occours 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.

```
void iupStrCopyN ( char *        dst_str,
                   int           dst_max_size,
                   const char *  src_str
                 )
```

Copy the string to the buffer, but limited to the max_size of the buffer. buffer is always porperly ended.

```
char* iupStrGetMemory ( int  size   )
```

Returns a buffer with the specified size+1.
The buffer is resused after 50 calls. It must NOT be freed. Use size=-1 to free all the internal buffers.

```
char* iupStrGetMemoryCopy ( const char *  str   )
```

Returns a buffer that contains a copy of the given buffer using iupStrGetMemory.

```
void iupStrLower ( char *        dstr,
                   const char *  sstr
                 )
```

Converts a string into lower case.

```
int iupStrToRGB ( const char *       str,
                  unsigned char *  r,
                  unsigned char *  g,
                  unsigned char *  b
                )
```

Extract a RGB triple from the string. Returns 0 or 1.

```
int iupStrToRGBA ( const char *       str,
                   unsigned char *  r,
                   unsigned char *  g,
                   unsigned char *  b,
                   unsigned char *  a
                 )
```

Extract a RGBA quad from the string, alpha is optional. Returns 0, 3 or 4.

```
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. Supports UNIX and Windows directory separators.

```
char* iupStrFileGetTitle ( const char *  file_name  )
```

Returns the file title of a file name. Supports UNIX and Windows directory separators.

```
char* iupStrFileGetPath ( const char *  file_name  )
```

Returns the file path of a file name. Supports UNIX and Windows directory separators.

```
char* iupStrFileMakeFileName ( const char *  path,
                               const char *  title
                             )
```

Concat path and title addind '/' between if path does not have it.

```
int iupStrReplace ( char *  str,
                    char    src,
                    char    dst
                  )
```

Replace a character in a string. Returns the number of occurrences.

```
void iupStrToUnix ( char *  str  )
```

Convert line ends to UNIX format in place (one
per line).

```
char* iupStrToMac ( const char *  str  )
```

Convert line ends to MAC format (one per line). If returned pointer different than input it must be freed.

```
char* iupStrToDos ( const char *  str  )
```

Convert line ends to DOS/Windows format (the sequence
per line). If returned pointer different than input it must be freed.

```
void iupStrRemove ( char *  value,
                    int     start,
                    int     end,
                    int     dir
                  )
```

Remove the interval from the string. Done in place.

```
char* iupStrInsert ( const char *  value,
                     const char *  insert_value,
                     int           start,
                     int           end
```

)

Remove the interval from the string and insert the new string at the start.

char* iupStrProcessMnemonic (const char * *str*,
char * *c*,
int *action*
)

Process the mnemonic in the string. If not found returns str. If found returns a new string. Action can be:

- 1: replace & by c
- -1: remove & and return in c
- 0: remove &

Language Dependent String Messages

[Utilities]

Collaboration diagram for Language Dependent String Messages:

Functions

void iupStrMessageShowError (Ihandle *parent, const char *message)
char * iupStrMessageGet (const char *message)

Detailed Description

String database that is dependend of the selected language.

See iup_strmessage.h

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.

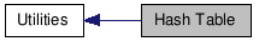
char* iupStrMessageGet (const char * *message*)

Returns a common string from a registered coded message. The returned string depends on the global LANGUAGE attribute.

Hash Table

[Utilities]

Collaboration diagram for Hash Table:



Typedefs

typedef enum _Itable_IndexTypes Itable_IndexTypes
typedef enum _Itable_Types Itable_Types

Enumerations

enum _Itable_IndexTypes { IUPTABLE_POINTERINDEXED = 10, IUPTABLE_STRINGINDEXED }
enum _Itable_Types { IUPTABLE_POINTER, IUPTABLE_STRING, IUPTABLE_FUNCPOINTER }

Functions

Itable * iupTableCreate (Itable_IndexTypes indexType)
Itable * iupTableCreateSized (Itable_IndexTypes indexType, unsigned int initialSizeIndex)
void iupTableDestroy (Itable *n)
void iupTableClear (Itable *it)
int iupTableCount (Itable *it)
void iupTableSet (Itable *n, const char *key, void *value, 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, 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)

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

Typedef Documentation

typedef enum _Itable_IndexTypes Itable_IndexTypes

How the table key is interpreted.

typedef enum _Itable_Types Itable_Types

How the value is interpreted.

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 (Itable_IndexTypes *indexType*)

Creates a hash table with an initial default size. This function is equivalent to iupTableCreateSized(0);

Itable* iupTableCreateSized (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. Calls iupTableClear.

void iupTableClear (Itable * *it*)

Removes all items in the table. 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*,
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*,
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(). iupTableGetCur() is faster then 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:

- _ -

- _IattribFlags : iup_class.h
- _IchildType : iup_class.h
- _InativeType : iup_class.h
- _Itable_IndexTypes : iup_table.h
- _Itable_Types : iup_table.h
