YaST2 Documentation

SUSE Linux AG

YaST2 Team, SUSE Linux AG

YaST2 Documentation:

SUSE Linux AG
by YaST2 Team

Table of Contents

List of Figures

List of Tables

List of Examples

Preface

Administering a Linux system at the lowest level is sometimes not an easy task. If it should be done manually, it requires a very experienced and knowledgeable person, willing to browse and edit very many configuration files. As a result there have been many efforts to create intelligent tools that provide rather automatic means to accomplish this challenging and (at the same time) tedious task.

One of these tools is `YaST`, the SuSE Linux™ installer. Being the result of a rather long period of development, it is by now a very large and capable system, well suited to install and administer a SuSE Linux™ system. While the internal functionality of `YaST` is quite multifaceted and therefore not exactly easy to understand, it should not be kept as a secret. Rather the world shall be encouraged to make use of the mechanisms `YaST` can provide.

This goal can be achieved because `YaST` is not a closed monolithic system but has a high degree of modularity. In fact it consists largely of modules that could as well be created by people not related to `YaST` development. For example hardware vendors could provide a `YaST` module for customizing specific system settings related to their particular piece of hardware. From the user's point of view this would be much more comfortable than editing configuration files by hand.

Of course this can't be done without some knowledge of the `YaST` internal functionality. So this document tries to lighten things up by advancing from the unsubtle connections in the beginning to more and more detailed explanations towards the end. However, describing the particularities of this matter in full detail would easily fill several heavy books which in turn would contradict the goal of introductional simplification. Furthermore some of these “details” are subject to moderate change service, which would render “static” documentation like this one outdated rather quickly.

To alleviate these problems, this text very often refers to the “official” `YaST` developers documentation that can be found in `/usr/share/doc/packages/yast2*` (especially towards the end). Aside from the references to be found in the following text, this location provides very valuable information regarding the whole `YaST` environment. To have access to these files the following packages must be installed:

- yast2-devel
- yast2-core-devel
- liby2util-devel
- yast2-packagemanager-devel

Note

This document is available in HTML and PDF. However, the primary target format is HTML because of the many references to the `YaST` developers documentation. When viewed with a web browser these links are functional and provide easy access to the respective files. Unfortunately in the PDF format the links are *not* functional rendering this document not very useful when it comes to the details.

1. What's Inside

This document is subdivided into the following chapters:

Introduction. A brief explanation of the intention and nature of this document.

YaST - The Big Picture. This is a short depiction of the `YaST` installer and the `YaST` environment as such. The architectural peculiarities of `YaST` are explained as far as it is necessary to understand the elucidations that follow thereafter.

The YaST Language - YCP. This chapter is dedicated to the `YaST` language that constitutes most of the high-level functionality of `YaST`. Sections covering the basic language elements are accompanied by others that deal with user interface creation and program structure.

SCR Details. In this chapter the `YaST` *System Configuration Repository* (SCR) is explained in some concise detail. It shows how to access configuration data and hardware data from within `YaST` modules in a consistent way.

YaST Modules. Some explanations regarding the different types of `YaST` *modules* as well as some rules for writing them.

Appendix A. References. Throughout this document there are numerous references to the `YaST` developers documentation. To ease access to these links this appendix is mostly a dense index of those references.

2. Style Conventions

Throughout this document some conventions regarding the typeface of printed text are used:

- *Emphasised* text is used to denote important parts of the text.
- Product names, file names and paths are printed using `literal` typeface. Furthermore cut-outs from programs that are embedded in the normal text flow are printed this way.
- Commands and command lines are printed using the **command** typeface.
- Keyboard keys are denoted as in **CTRL-C**.
- The description of programming language elements is displayed as shown below.

  Synopsis: **while (***condition***)** *loop_body*

  The parts of the language construct are printed like **commands** while any arguments are printed *emphasized*.

Chapter 1. Introduction

YaST is the installation program used by SuSE Linux™ to install Linux on a system and to administer this installed system thereafter. The notation “program” is a bit misleading here because in fact YaST consists of many components and layers. Therefore one may as well ragard YaST as an installation and administering environment.

Among the most important components in this environment are the YaST-modules which are usually written in a YaST-specific language called *YaST Control Language* (YCP). With exception of some rare cases the whole high-level functionality of YaST is formulated in YCP. When YaST is running, the YCP-modules are interpreted by the low-level YaST-components and the YCP-code makes use of the infrastructure provided by them. The possibility to add such modules at any time realizes the concept of extensibility that is inherent in YaST.

This concept of extensibilty by means of modules has been designed into YaST from the very beginning. In fact YaST-modules are the layer of YaST the user comes in contact with. Nearly every dialog on screen during the installation is realized as a YaST-module and there are also modules that act behind the scenes to care of specific pieces of hardware, e.g. the keyboard.

The YaST-modules mentioned so far come ready-made with the distribution and provide the core functionality to install and administer a SuSE Linux™-system, but this need not and should not be the end. The extension facility is intended to be also used by “third party people”, e.g. harware vendors, who want to contribute YaST-functionality in some way.

To pave the way for this intention to come alive this document will provide some insight into the inner mechanics of YaST. The primary goal of the following elucidations is to make available the information that is needed to write YaST-modules that conform to the programming paradigm imposed by YaST.

What This Document Is. This document explains in some detail how to extend the functionality of YaST by means of modules. Of course every module of more than trivial functionality will have to make use of the features that are provided by the YaST-core-components and other YaST-modules. While the official YaST developers documentation is the primary knowledge base for all YaST related information, it is a bit overwhelming for everyone who tries to get into the matter for the first time. Therefore this document tries to provide a gentle introduction by explaining things rather explicitely in the beginning and getting more and more concise towards the end. By providing very many references to the developers documentation this document can be thought of as a “guided index” to this voluminous material.

What This Document Is Not. This document will not explain the “binary” particularities related to YaST, i.e. there will be no implementation notes on how the low-level machinery of YaST is realized. Because YaST implies the module concept for extensibility, only this approach is promoted here. The “engine” that executes these modules and how it is assembled is subject to the following explanations only in so far as it is neccessary to understand the interaction of the various components.

The Audience Of This Document. So this document is intended for all people who want to make use of the YaST-functionality by providing modules for a specific task. Additionally the information presented herein might be interesting for all those who want to adept something about the whys and wherefores related to YaST as such. Furthermore, to get most out of this reading, the reader is supposed to have some programming experience in a structured programming language, ideally C/C++. Some expertise in functional programming could also be helpful but is not really necessary.

Chapter 2. YaST - The Big Picture

To be able to create a `YaST`-module it is necessary to have understood how the extensive `YaST`-world is structured, which components there are, what they do and how they do it. Therefore prior to going into closer detail we'll step back from the blackboard and have a look at the big picture first. By doing so you will have the opportunity to get an overview of the ample terrain `YaST` is living on. While you don't have to understand each and every byte `YaST` consists of, having seen the whole issue will ease your understanding of the details we will come across.

2.1. Overview

`YaST` has been invented to have an extensible and fairly standardized means to install and manage SuSE Linux™ on a system. Basically `YaST` serves three main purposes:

- Installation of SuSE Linux™ on a system
- Configuration of the installed system
- Administration of the installed system

To provide a solution to the resulting demands that has a lifetime extensible well into the future this solution had to be flexible and maintainable. Consequently some key concepts determined the design of `YaST`. In particular it was the strict separation of:

- The user interface
- The functional code doing the job
- The data representing the current state of the system

Furthermore `YaST` has some very specific attributes that make it unique for the user as well as for those people who are developing it or contributing to it. The following sections outline some of the features of the `YaST` installer that should be seen as a guiding line for module development.

2.1.1. Access To The System

Managing a SuSE Linux™ system requires direct low-level access to the system which generally means reading and writing configuration data. Of course this could be done manually by a knowledgeable person using a conventional editor. A more comfortable and in most cases safer way is to use `YaST`. Consequently `YaST` must be able to handle this configuration data on the system level. By handling the *original* data `YaST` activities take into account manual editing that might also occur. Thus nobody is *forced* to use `YaST` exclusively for configuration tasks.

In `YaST` the access to system configuration data is realized by means of a special component (or layer if you prefer), the *System Configuration Repository* (SCR) (see below and Chapter 5, *SCR Details*). The SCR component basically consists of a number of so-called *agents* that have been created to accomplish a specific kind of access. For example there is an agent to run shell-commands and there is another one that reads and writes ASCII-files of a specific format. Additionally there are agents that provide access to the system hardware e.g. by taking hold on the proc-file-system.

All these agents are gathered together under a common hood, the SCR-API that can be used from within the `YaST`-modules in a consistent way. In summary the SCR provides kind of a *view* on all kinds of data, either YaST2 internal data, original system config files or hardware data.

2.1.2. Reasonable Suggestions

YaST implies lots of artificial intelligence to provide reasonable suggestions for the various tasks. During installation the target system is thoroughly analyzed with respect to its hardware components and in most cases YaST succeeds in suggesting a proper configuration for them.

These suggestions are presented in an overview dialog that shows the main characteristics of the system to be installed and how YaST would handle them. If you are satisfied with these automatically generated settings you can simply accept them. If not, each of the system configuration categories can be “activated” to be changed manually. This is where the *workflows* come into their own.

2.1.3. Workflows

If you decide to change a specific configuration category this is usually being done in a workflow. Workflows are used to lead you through the steps neccessary to accomplish a specific task. The steps are generally small to avoid an information “overflow”. At the end of the sequence the task has been accomplished and the changes are made permanent in the system.

As was stated above, you are not *forced* to do it this way. You could as well edit configuration files by hand but YaST can offer as much help as possible for this. Sometimes a workflow has multiple branches for “novice” and “expert” modes. The novice mode fills in the default values and tries to determine as much as possible automatically. The expert mode offers full control and allows to enter even unreasonable values.

By providing pre-configured workflows and configuration data, it is possible to automate almost arbitrary configuration tasks with YaST. From adding a user, to installing a completely configured SuSE Linux™ on specific hardware, nearly everything is possible.

2.1.4. Modules And The YCP-Language

Every workflow is assembled from rather small steps, implemented by means of *YaST modules* written in a YaST-specific scripting language, the *YaST Control Language* (YCP). These YaST-modules are then called in a predefined sequence to complete a specific task.

In fact it is possible to even write modules in *bash* and *perl* as long as the module need not have a user interface, i.e. it is not interactive. Such non-interactive modules typically handle specific problems like controlling a particular piece of hardware and can be called from within YCP-modules. This building block approach makes constructing complex workflows easy and maintainable.

2.1.5. User Interface

The YCP language is also used to control the user interface (UI) presented on screen. The UI displays the information already known by the system and retrieves the information entered by the user.

There are two modes of operation:

- *Text mode for console-based service*

  In text mode the user interface is presented in the NCurses environment that provides windowing capabilities and entry forms on a text-based console. Mouse support is neither possible nor necessary here because all dialogs can be operated using only the keyboard.

- *Graphics mode for X11-based service*

  In graphics mode the well-known Qt-system is used to present the dialogs in a graphical way using a running X11-server. Operating these dialogs follows the common habits of graphical user interfaces.

It is important to notice here that both methods principally use the same YaST-specific YCP-API to build the dialogs. While there are some (rare) cases where the YCP-code has to distinguish these modes, the dialogs are usually programed for both worlds in in one single source with the same code.

2.1.6. Summary

In summary `YaST` provides the following features, some of them having already been mentioned above:

- *System access*

  `YaST` provides thorough probing of the system hardware and presents the information gathered thereby via the SCR-API. The SCR is also the means for reading and writing configuration files.

- *Reasonable Suggestions*

  Based on the system analysis and predefined configuration data, `YaST` is able to provide reasonable suggestions for almost any configuration task.

- *Workflows*

  Management of particular configuration categories is usually realized in form of workflows that split up the whole task into small steps.

- *Modules and YCP*

  The steps constituting a workflow are usually realized as `YaST`-modules that are written in the *YaST Control Language* (`YCP`)

- *User interface*

  The user interface of `YaST` is realized by means of a specific API from within the `YCP`-modules. This API supports a text-based console-mode as well as a graphical X11-mode.

- *Internationalisation*

  `YaST` provides support for various languages.

- *Multi-platform support*

  `YaST` provides support for various platforms like Intel™ (x86), Apple™, IBM™ (s390) etc.

2.2. YaST Architecture

YaST2 is a modular system for Linux installation and system administration. The design goals include:

- Flexibility
- Extensibility
- Maintainability
- Network transparency

  support administration of remote hosts or virtual machines on mainframes, machines without CD/DVD drives, rack-mounted machines

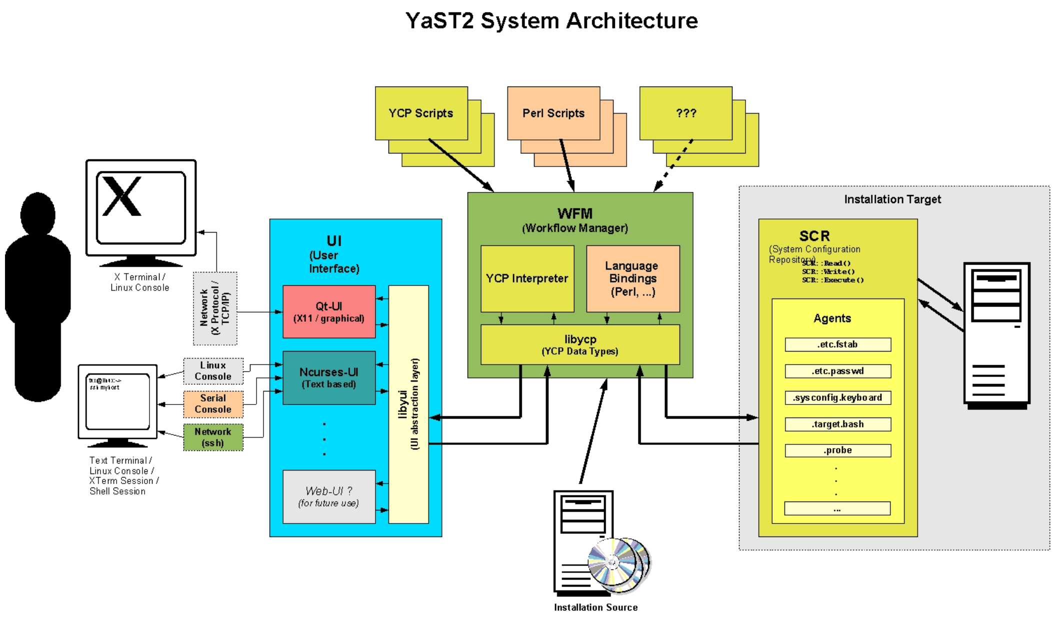
- User interface independence

  must run in graphical and text-only environments and serial consoles

- Cover the whole range from novice users to expert system administrators

To achieve the above design goals, YaST2 is split up into a number of components for each individual task:

Figure 2.1. The YaST Architecture



There is the core engine and to run scripts written in YCP (YaST2's own scripting language), Perl or (in future releases) other scripting languages.

The engine and scripts together form a YaST2 Module for the user.

2.2.1. The SCR (System Configuration Repository)

Even though in most scenarios there is only one single machine, it is important to distinguish between the installation source machine and the installation target machine:

- The installation source machine is the machine that holds the installation media - usually CDs or DVDs - and a mini-Linux called "inst-sys" that is copied from one of those installation media to that machine's RAM disk to have a basic operating system to work with on a "bare metal" machine (a machine that doesn't have an operating system installed yet). Most of that inst-sys is read-only, there is only limited disk space for temporary files, and since everything runs from a RAM disk the writable part of it is very volatile.

- The installaton target on the other hand is the machine that is to be installed or administered. That may be the same machine as the installation source machine (in fact, this is very common for PC installation or administration tasks), but it might as well be two distinct machines - a virtual machine on a mainframe computer or a remote rack-mounted machine without any display adapter or CD/DVD drives.

All communication with the installation target is handled via the System Configuration Repository (SCR) to guarantee the network abstraction design goal. This is much easier said than done, however: YaST2 module developers always have to keep in mind that it is strictly forbidden to access system files (or any other system resources, for that matter) directly, even if there may be very convenient CPAN Perl modules to do that. Rather, SCR is to be used instead - always. Otherwise everything might run fine if installation source and target are the same machine, but break horribly if they are not.

SCR in itself is also modularized: All calls are handled by "agents" that each know how to handle a particular configuration "path" like `"/etc/fstab"` or `"/etc/passwd"`. That may be a simple file, but it may also be a directory hierarchy like "probe" - this particular agent handles all kinds of hardware probing, from mouse and display adapters to storage device controllers (like SCSI or IDE controllers), disks attached to each individual controller or partitions on those disks. Paths are denoted like ".etc.fstab" for SCR. YCP even has a special data type "path" for just this case (a special kind of string with some special operations).

SCR agents handle no more than three calls:

- SCR::Read()
- SCR::Write()
- SCR::Exectute()

The first argument is always the path to handle, but there may be any number of additional parameters, depending on the agent.

While Read() and Write() are obvious, Execute() may not be: This is intended for some kinds of agents that actually run a program on the installation target. In particular, the ".target.bash" agent does that - it runs a "bash" shell on the target machine and accepts a shell command as an argument. This is the tool of choice for tasks such as starting or stopping system services on the target machine - and again, the distinction between installation source and installaton target machine becomes very important: You want to start or stop the service on the (possibly remote) target machine, not on the machine that happens to hold the installation media.

SCR agents can easily added when needed. There are frameworks available to write SCR agents in C++, in Perl, or as Bash shell scripts as well as several ready-made parsers for different file formats like the ".ini" file parser that can handle files with "key = value" pairs or the "anyagent" that generalizes that concept even more using regular expressions. Those parsers return YCP lists and maps ready for further processing.

Typically, a YaST2 module for a specific installation or administration task includes a set of YCP or Perl scripts as well as some SCR agents to handle its particular configuration files.

2.2.2. The UI (User Interface)

Given the wide variety of machines that can possibly be handled with YaST2, it is important to keep the user interface (UI) abstraction in mind - very much like the SCR, the UI does not necessarily run on the installation target machine. It doesn't even need to run on the same machine as the WFM.

The UI provides dialogs with "widgets" - user interface elements such as input fields, selection lists or buttons. It is transparent to the calling application if those widgets are part of a graphical toolkit such as Qt, or text based (using the NCurses library) or something completely else. An input field for example only guarantees that the user can enter and edit some value with it. A button only provides means to notify the application when the user activated it - by mouse click (if the UI supports using pointing devices such as a mouse), by key press or however else.

The UI has a small number of built-in functions - for example:

- UI::OpenDialog() accepts a widget hierarchy as an argument and opens a dialog with those widgets
- UI::CloseDialog() closes a dialog

- UI::QueryWidget() returns a widget's property such as the current value of an input field or selection box
- UI::ChangeWidget() changes a widget's property
- UI::UserInput() waits until the user has taken some action such as activate a button - after which the application can call UI::QueryWidget() for each widget in the dialog to get the current values the user entered. The application does not have to handle every key press in each input field directly - the widgets are self-sufficient to a large degree.

There is virtually no low-level control for the widgets - nor is it necessary or even desired to have that. You don't specify a button's width or height - you specify its label to be "Continue", for example, and it will adapt its dimensions accordingly. If desired, more specific layout constraints can be specified: For example, buttons can be arranged in a row with equal width each. The UI will resize them as needed, giving them additional margins if necessary.

The existing UIs provide another layer of network abstraction: The graphical UI uses the Qt toolkit which is based on the X Window System's Xlib which in turn uses the X protocol (usually) running on top of TCP/IP. X Terminals can be used as well as a Linux console (that may be the installation source machine or the installation target machine or another machine connected via the network) running the X Window System or even X servers running on top of other operating systems.

The NCurses (text based) UI requires no more than a shell session - on a text terminal (serial console or other), on a Linux console, in an XTerm session, via ssh or whatever.

Currently, there is no web UI, but YaST2's concepts would easily allow for that if it proves useful or necessary.

2.2.3. YaST Core Engine

The component broker is the central piece of YaST. It acts as a dispatcher for all other components: When a (YCP, Perl or whatever) script calls a function, the broker determines what component handles that function call based on the respective namespace identifier. It is transparent to the caller what programming language a function is written in; the component broker handles that kind of dispatching. The caller only needs to know the function name, its namespace and (or course) the required parameters.

For example, calls like UI::OpenDialog() go to the UI (the user interface), SCR::Read() to the SCR (the system configuration repository). Even scripts can provide namespaces via modules in YCP or Perl.

All communication between the different parts of YaST core is done via a predefined set of YCP data types - simple data types like string, integer, boolean etc., but also compound data types like maps (key / value pairs, also known as "hashes" in other programming languages) or lists (like arrays or vectors in other programming languages). For complex data structures, maps, lists and simple data types can be nested to any degree.

2.2.4. External Programs

The core-engine of `YaST` consists of some binary components (modules) that are interconnected via `YaST`-specific protocols. There are *clients* as well as *servers* that are responsible for specific tasks that may have to be accomplished during a `YaST`-session. According to the well-known client-server-paradigm often used in software technology, `YaST`-servers are program modules that *passively* await connections from certain clients to process their requests. Clients on the other hand are *active* components that send requests to the servers thereby initiating certain actions.

For example the SCR and the UI act as server components that process client-requests on demand. An example for a client module is the *stdio-component* that can be used to connect the `YaST`-internal communication with a terminal.

Because this architectural speciality is meant to be used only by the `YaST` core developers to establish and maintain the low-level machinery we will not go into more detail here. Instead we will focus on the advocated method of extending `YaST` at the "open end" by creating `YCP`-modules.

Chapter 3. The YaST Language - YCP

The `YaST`-language `YCP` is a scripting language to be interpreted by the `YCP`-engine (`YCP` interpreter) specially designed for manipulation with a system configuration. Its syntax is very similar to C programming language. Because `YCP` can make use of the whole infrastructure that `YaST` provides, the actions that can be accomplished with `YCP` are very powerful.

`YCP` has the usual features of procedural languages and some more, partially originating from the functional programming paradigm:

- Control structures like if/then/else, foreach-loops.
- Compound data types like strings, lists and maps.
- Function definition (procedures)
- Variable scopes
- Name spaces
- Include files
- UNIX command execution (via the YaST infrastructure)

On the following pages we will explore the `YCP` language definition and find out how to use `YCP` to write "programs" that can be executed by `YaST`.

3.1. The First YCP Program

Probably the best way to get into the matter is by means of a simple example.

3.1.1. YCP Source

The following little program opens a window that displays the string "Hello, World!" and provides a push button for termination.

Example 3.1. "Hello World" in YCP

```
{
    string message = "Hello, World!";

    UI::OpenDialog(
                `VBox(
                      `Label( message ),
                      `PushButton("&OK")
                     )
              );

    UI::UserInput();

    UI::CloseDialog();
}
```

In the following this code will be explained shortly in a line-by-line manner thereby touching some topics we will examine in detail later on.

- `{`

  The opening curly opens a so-called *block* in `YCP`. Blocks are used to "glue" several `YCP`-

statements together to form an entity that can be handled just like a single statement.

- `string message = "Hello, World!";`

  In this line we define a variable named “message” that is of type string. In `YCP` any variable definition *must* imply a value assignment to avoid all errors that might occur due to uninitialized variables. Here we assign the constant string “Hello, World!”. Furthermore the terminating semicolon is mandatory in `YCP` to indicate the end of a statement (just like C).

- `UI::OpenDialog(`

  This command opens a dialog on screen. Because we want to display something, the code describing our dialog has to be sent to the UI. This is being done by the leading name space identifier `UI::`. The (single) parameter that is supplied here determines the content of the dialog.

- `` `VBox( ``

  This is a UI-statement related to the geometry of the dialog to be defined. As the name indicates it opens a (virtual) *vertical box* that displays all content in a column-wise manner. (Geometry management is described in more detail in Section 3.9, “Controlling The User Interface”).

  The leading back-quote introduces a `YCP`-feature that stems from the functional programming paradigm. In `YCP`-speak the `` `VBox() `` is a *term*. In `YCP`, terms are used as a structured constants and are typically passed to functions provided by YaST infrastructure as parameters as is done here with `OpenDialog()`.

- `` `Label( message ), ``

  Displaying strings in `YCP` is done by means of *Labels*. This statement gets one parameter, the string variable we defined in the beginning. Because it is the first of two parameters passed to `` `VBox() `` this line is not terminated with a semicolon but with a comma. As in most programming languages commas are used to separate parameters in `YCP`.

- `` `PushButton("&OK") ``

  This statement displays a labelled push button. Since it is the next element in the enclosing `` `VBox() ``, it is displayed immediately below the preceding label. The & in the label string is a `YaST` feature declaring the subsequent character to be a key-shortcut. As a result the button can not only be clicked with the mouse but also be activated by typing **ALT-O**.

- `)` and `);`

  The next two lines first close the open `` `VBox() `` and then the open `OpenDialog()`. Because `` `VBox() `` is passed as a parameter to `OpenDialog()` there is no need to terminate the statement with a semicolon. `OpenDialog()` on the other hand *is* a statement in the UI and hence *must* be terminated with a semicolon.

- `UI::UserInput();`

  Here we hand over control to the UI which then awaits some sort of user input. In this case it simply waits for the push button to be pressed by the user. Consequently our program blocks at this point until the user really does it.

- `UI::CloseDialog();`

  After all the UI-related action has finished, i.e. when `UI::UserInput()` returns, we want to remove the dialog we just created. This is done here.

- `}`

  Indicating the end of the block, the closing curly bracket ends our little `YCP`-program.

3.1.2. The YCP compiler

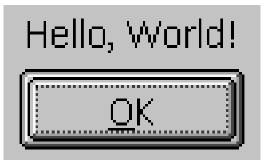
Section not written yet...

3.1.3. Running YCP

Now we can start the program using YaST. For this, we will use a script `/sbin/yast2`. It is an envelope for easier setup of a running YaST environment.

So if you are reading this document with a browser, you could copy-and-paste the program listed above into a file `hello.ycp`, and then run **/sbin/yast2 hello.ycp** which should render the following "spectacular" result.

Figure 3.1. Output of the "Hello, World!"-program


Hello, World!

Starting off with this simple example we will now explore the more subtle details of `YCP`. Since all programming is about handling of data there must be a way to hold it in variables of different types. In the next section you will get to know the various data types that `YCP` knows about.

3.2. YCP Data Types

Just like any other high-level programming language YCP has typed variables to hold data of different kinds:

- Data type void
- Data type symbol
- Data type boolean
- Data type integer
- Data type float
- Data type string
- Data type byteblock
- Data type list
- Data type map
- Data type term
- Data type path
- Data type block
- Data type symbol
- Data type any

3.2.1. Data Type *void (nil)*

This is the most simple data type. It has only one possible value: `nil`. Declaring variables of this type doesn't make much sense but it is very useful to declare functions that need not return any useful value. `nil` is often also returned as an error flag if functions fail in doing their job somehow.

3.2.2. Data Type *symbol*

A symbol is a literal constant. It is denoted by a single backquote and a letter or underscore optionally followed by further letters, underscores or digits.

Example 3.2. Symbol constants

```
`literal
`next
```

3.2.3. Data Type *boolean*

In contrast to C/C++, a `YCP` boolean is a real data type with the dedicated values `true` and `false`. Comparison operations like `<` or `==` evaluate to a boolean value. The `if (...)` statement expects a boolean value as the result of the decision clause.

3.2.4. Data Type *integer*

This is a machine independent signed integer value that is represented internally by a 64 bit value. The valid range is from `-2^63` through `2^63-1`. Integer constants are written just as you would expect. You can write them either decimal or hexadecimal by prefixing them with `0x`, or octal by prefixing them with `0` (just like in C/C++).

Example 3.3. Integer constants

```
1
-17049349
0x9fa0
0xDEADBEEF
0233
```

3.2.5. Data Type *float*

Floating point numbers. Because they are represented via the C datatype `double` the valid range is machine dependent. Constants are written just as you would expect. The decimal point is mandatory only if no exponent follows. Then there must be at least one digit leading the decimal point. The exponent symbol may be `e` or `E`.

Example 3.4. Float constants

```
1.0
-0.0035
1e30
-0.128e-17
```

3.2.6. Data Type *string*

Represents a character string of almost arbitrary length (limited only by memory restrictions). String constants consist of UNICODE characters encoded in UTF8. They are enclosed in double quotes.

The backslash can be used to mark special characters:

Table 3.1. Special characters in *strings*

Representation	Meaning
`\n`	Newline (ASCII 10)
`\t`	Tabulator
`\r`	Carriage Return (ASCII 13)
`\b`	Backspace
`\f`	Formfeed
\abc	ASCII character represented by the octal value *abc*. Note that unlike in C, there must be exactly 3 octal digits!
\X	The character *X* itself.

A backslash followed by a newline makes both the backslash and the newline being ignored. Thus you can split a string constant over multiple lines in the `YCP` code.

Example 3.5. String constants

“This string ends with a newline character.\n”
“This is also a newline: \012”

3.2.7. Data Type *byteblock*

A byteblock simply is a sequence of zero or more bytes. The ASCII syntax for a byteblock is `#[hexstring]`. The *hexstring* is a sequence of hexadecimal values, lower and upper case letters are both allowed. A byte block consisting of the three bytes 1, 17 and 254 can thus be written as `#[0111fE]`.

In most cases, however, you will not write a byteblock constant directly into the `YCP` code. You can use the SCR to read and write byteblocks.

Example 3.6. Byteblock constants

#[ ]
#[42]
#[0111fE]
#[03A6f298B5]

3.2.8. Data Type *list*

A list is a finite sequence of values. These values need not neccessarily have the same data type. List constants are denoted by square brackets. In contrast to C it is possible to use complex expressions as list members when defining a list constant. The empty list is denoted by `[]`.

Example 3.7. List constants

```
[ ]
[ 1, 2, true ]
[ variable, 17 + 38, some_function(x, y) ]
[ "list", "of", "strings" ]
```

Accessing the list elements is done by means of the index operator as in `my_list[1]:"error"`. The list elements are numbered starting with 0, so index 1 returns the second element. After the index operator there *must* be a colon denoting a following *default value* that is returned if the list access fails somehow. The default value should have the type that is expected for the current list access, in this case the string “error”.

Note 1: A list preserves order of its elements when iterating over them.

Note 2: There is also another method for accessing lists, originating from the early days of `YaST`. The command `select(my_list, 1, "error")` also returns the second element of `my_list`. While this still works, it is deprecated by now and may be dropped in the future.

3.2.9. Data Type *map*

A `YCP`-map is an associative array. It is a list of key-value-pairs with the keys being non-ambiguous, i.e. there are no two keys being exactly equal. While you can use values of any type for keys and values, you should restrict the *keys* to be of type *string* because from experience other types tend to complicate the code. *Values* of arbitrary type on the other hand make the map a very flexible data container. Maps are denoted with `$[ key_0:value_0, key_1:value_1, ...]`. The empty map is denoted by `$[]`.

Note: A map does not reserve order of its elements when iterating over them.

Example 3.8. Map constants

```
$[ ]
$[ "/usr": 560, "/home" : 3200 ]
$[ "first": true, "2": [ true, false ], "number" : 8+9 ]
```

Accessing the map elements is done by means of the index operator as in `my_map["os_type"]:"linux"`. This returns the value associated with the key `"os_type"`. As with lists, a default value *must* be appended (after a colon) that is returned if the given key does not exist. Again it should have the type that is expected for the current access, in this case the string “linux”.

You may have noticed that the syntax for accessing maps kind of resembles that of accessing lists. This is due to the fact that lists are realized as maps internally with constant keys 0, 1, 2, and so on.

Note: There is also another method for accessing maps, originating from the early days of `YaST`. The command `lookup(my_map, "os_type", "linux")` also returns the value associated with the given key. While this still works, it is deprecated by now and may be dropped in the future.

3.2.10. Data Type *path*

A path is something special to `YCP` and similar to paths in TCL. It is a sequence of path elements separated by dots. A path element can contain any characters except `\x00`. If it contains something else than `a-zA-Z0-9_-` it must be enclosed in double quotes. The *root path*, i.e. the root of the tree is denoted by a single dot. Paths can be used for multiple purposes. One of their main tasks is the selection of data from complex data structures like the SCR-tree (see Section 5.2, “SCR Tree”).

The backslash can be used to mark a special characters:

Table 3.2. Special characters in *paths*

Representation	Meaning
`\n`	Newline (ASCII 10)
`\t`	Tabulator
`\r`	Carriage Return (ASCII 13)
`\b`	Backspace
`\f`	Formfeed
`\`*xXX*	ASCII character represented by the hexadecimal value *XX*.
`\`*X*	The character *X* itself.

Example 3.9. Path constants

```
.
.17
.etc.fstab
."\nHello !\n".World
."\xff" == ."\xFF"
."\x41" == ."A"
."" != .
```

3.2.11. Data Type *term*

A term is something you won't find in C, Perl, Pascal or Lisp but you will find it in functional programming languages like Prolog for example. It is a list plus a symbol, with the list written between normal brackets. The term `` `alpha(17, true) `` denotes a symbol `` `alpha `` and the list `[ 17, true ]` as *parameters* for that symbol. This looks pretty much like a function call.

You can also use the term as a parameter in another function call, for example to specify a user dialog.

Example 3.10. Term constants

```
`like_function_call(17, true)
`HBox(`Pushbutton(`Id(`ok), "OK"), `Textentry(`Id(`name), "Name"))
```

3.2.12. Data Type *any*

In the previous sections you have seen the data types `YCP` knows about. In most cases you will (and should) assign a certain data type to every variable you declare. However, there might be cases when the type of a variable is not really clear at coding time, e.g. (in some rare cases) if you access the SCR to get some hardware data. While you should try very hard to avoid this situation, there might be cases where you can't.

To solve this problem, you may assign the type *any* to your variable which makes it accept assignments of any other valid type. However, because variables of type *any* are highly deprecated in `YaST` by now, this “feature” will eventually be dropped in the near future.

3.3. More YCP types

Section not written yet...

3.3.1. Data Type *block*

Basically a block is a sequence of YCP statements enclosed in curly brackets. It can be a whole YCP program as was the case with the outermost block in *hello.ycp* from Section 3.1.1, “YCP Source”. What is special about blocks in YCP is that they represent a value and therefore can be assigned to a variable. It is sometimes really useful to have those blocks as YCP values because this makes it possible to use them as parameters to function calls. Of course the syntactical structure of blocks can become rather complex which leads to a description of the whole language itself. Therefore we put this into a section of its own: Section 3.8, “YCP Program Structure”.

For now the following examples should suffice.

Example 3.11. Block constants

{ return 17; }
{ integer a = 5; return a + 8; }

3.4. YCP Type System

Section not written yet...

3.4.1. Data Type *any* And Type Checking

In Data type any you have seen the data type `any`. Because the value of type `any` can not be assigned to a variable of any other type. FIXME. So it is importnat to check its type with `is(...)` and then re-assigning it to a variable of the correct type. The following example shows how this should be done.

Example 3.12. Type checking and data type *any*

```
//
// Hypothetical example:
// ---------------------
// We don't know whether the SCR will return integers or floats...
//
any     any_var   = 0;
integer int_var   = 0;
float   float_var = 0.0;
boolean int_case  = false;

any_var = SCR::Read(...);

if ( is( any_var, integer ) )
{
   int_var  = any_var;
   int_case = true;
}
else if ( is( any_var, float ) )
{
   float_var = any_var;
   int_case  = false;
}
else
{
   // Error...
}

if ( int_case )
{
   // Use int_var...
}
else
{
   // Use float_var...
}
```

As this is *very* cumbersome, you should try to avoid this oddity in any case. If it is ineluctable, do it as shown above to stay compatible with future YaST behaviour.

3.5. YCP Expression Evaluation

From the interpreters point of view any YCP value is an expression and thus can be evaluated. *How* the evaluation is done in a particular case depends on the data type of the expression.

Because the *block* data type is somewhat special with respect to evaluation it will be explained first. The other basic data types will follow thereafter.

3.5.1. Evaluation Of Blocks

A YCP *block* is a sequence of *statements* enclosed in curly brackets. Upon evaluation (execution), all the statements in the block are evaluated one by one. Because blocks are also a valid data type in YCP, they can have a value (see Data type block). If a block contains the special statement `return(...)`, then the returned value replaces the block upon evaluation.

The following code sample shows a block with some statements.

```
{
   integer n = 1;

   while (n <= 10)
   {
      y2milestone("Number: %1", n);
      n = n + 1;
   }

   y2milestone("Returned number: %1", n);

   return n;
}
```

It calculates the numbers 1 through 10 and prints these numbers into the log file. The statement `y2milestone(...)` used for this is explained in YaST2 Logging along with YCP-logging as such. For now we are interested in the output that is written to the log file. As can be seen below the loop is executed 10 times and the counter has the value 11 after the loop. Finally the last statement `return(...)` determines the value of the whole block, in this case `11`.

```
...ycp/block_01.ycp:6 Number: 1
...ycp/block_01.ycp:6 Number: 2
...ycp/block_01.ycp:6 Number: 3
...ycp/block_01.ycp:6 Number: 4
...ycp/block_01.ycp:6 Number: 5
...ycp/block_01.ycp:6 Number: 6
...ycp/block_01.ycp:6 Number: 7
...ycp/block_01.ycp:6 Number: 8
...ycp/block_01.ycp:6 Number: 9
...ycp/block_01.ycp:6 Number: 10
...ycp/block_01.ycp:10 Returned number : 11
```

3.5.2. Evaluation Of Basic Data Types

The basic YCP data types we got to know in Section 3.2, “YCP Data Types” are evaluated in a rather straight forward way as will be shown in the following list.

Evaluation of basic data types

- *Simple data types*

  Most of the YCP data types can't be “evaluated” at all, as they simply evaluate to themselves. This holds for the simple types `void`, `boolean`, `integer`, `float`, `string`, `symbol` and `path`.

- *list*

  When evaluating a list, the interpreter evaluates all the list elements thereby forming a new list.

  ```
  {
     list list_var = [ 1 + 1, true || false, "foo" + "bar" ];
     y2milestone("list_var: %1", list_var );
  }
  ```

  yields the log file entry

  ```
  ...ycp/list_eval.ycp:4 list_var: [2, true, "foobar"]
  ```

- *map*

  A map is handled similar to a list. The values (but not the keys) are evaluated to form a new map.

  ```
  {
     map map_var = $[ "one" : `one, `two : "one" + "one" ];
     y2milestone("map_var: %1", map_var );
  }
  ```

  yields the log file entry

  ```
  ...ycp/map_eval.ycp:4 map_var: $["one":`one, `two:"oneone"]
  ```

- *term*

  Upon evaluation, term parameters are evaluated to form a new term.

  ```
  {
     term term_var = `val ( 1 + 1, true || false, "foo" + "bar" );
     y2milestone("term_var: %1", term_var );
  }
  ```

  yields the log file entry

  ```
  ...ycp/term_eval.ycp:4 term_var: `val(2, true, "foobar")
  ```

All the evaluations we have seen above are closely related to operators that may be used within an expression to act on the variables. The next section will give an overview of the operators that can be used in YCP.

3.6. YCP Operators

As any other programming language YCP knows a lot of operators that can be used to act on data.

3.6.1. Comparison Operators

These are binary operators for comparison of two values. The result is always boolean.

Operator	**Datatype**	**Description**
==	almost all	True if operands are equal, otherwise false.

Operator	Datatype	Description
<	almost all	True if left operand is smaller than the right one, otherwise false.
>	almost all	True if left operand is greater than the right one, otherwise false.
<=	almost all	True if left operand is smaller or equal to the right one, otherwise false.
>=	almost all	True if left operand is greater or equal to the right one, otherwise false.
!=	almost all	True if operands are not equal, otherwise false.

3.6.2. Boolean Operators

These are logical operators, that works with boolean datatype, two are binary one is unary. The result is always boolean.

Operator	Datatype	Description
&&	boolean	True if both operands are true, otherwise false (logical and).
||	boolean	True if at least one of the operands is true, otherwise false (logical or).
!	boolean	True if the operand if false, otherwise false (logical not).

3.6.3. Bit Operators

These are bit operators that works with integer, two are binary one is unary. The result is always integer.

Operator	Datatype	Description
&	integer	Bits of the result number are product of the bits of the operands (bit and).
|	integer	Bits of the result number are count of the bits of the operands (bit or).
~	integer	Bits of the result number are reverted bits of operand (bit not).
<<	integer	Bits of the result number are left shifted bits of the operands (bit shift left).
>>	integer	Bits of the result number are right shifted bits of the operands (bit shift right).

3.6.4. Math Operators

There math operators works with numeric data types (integer and float) and also with string. All are binary (except unary minus).

Operator	Datatype	Description
+	integer, float, string	The result is sum of the numbers or concatenation of the strings.
-	integer, float	The result is difference of the numbers.
*	integer, float	The result is product of the numbers.
/	integer, float	The result is quotient of the numbers (number class is preserved, thus quotient of integers produce integer, etc).
%	integer	The result is modulo.
unary -	integer, float	The result is negative number.

3.6.5. Triple Operator

This is the operator known from C language (*condition ? expression : expression*). The first operand is expression that can evaluate to boolean, types of second and third operands are code dependent. The result of the triple operator expression is the second operand in the case the first operand (condition) evaluates to true, the third one otherwise.

Code	Result	Comment
(3 > 2) ? true : false	true	The expression (3 > 2) evaluates to true, the result is true
contains ([1, 2, 3], 5) ? "yes" : "no"	"no"	The expression contains ([1, 2, 3], 5) evaluates to false, the result is "no"
(size ([]) > 0) ? 1 : -1	-1	The expression size ([]) > 0 evaluates to false, the result is -1

Note

Using brackets makes code cleaner, but is not necessary (according to operators precedence).

Note

With the introduction of the index operator (a = mapvar["key"]:default), the sequence "]:" became a lexical token usable only for indexing, so watch out when using the triple operator with lists and maps. Use parentheses or white space.

3.6.6. Operators Precedence

The table of operators precedence (from lowest to highest).

Direction	Operators
right	=
left	?:
left	||
left	&&
left	== !=
left	< <= > >=
left	+ -
left	* / %
left	<< >>
left	|
left	&
prefix	! ~ -

3.6.7. The bracket operator

3.6.7.1. Introduction

The bracket operator is the use of '[' and ']' like accessing arrays in C.

In YCP, this operator is used to ease handling with (possibly nested) lists and maps.

The bracket operator can be applied to any list or map variable and should be used in favour of (deeply) nested lookup() and select() cascades.

3.6.7.2. Access variant

The access variant of the bracket operator is used for accessing elements of a list or a map. It effectively replaces `select` for lists and `lookup` for maps.

3.6.7.2.1. Accessing lists

General syntax:

for simple lists:

<list-var>`[`*<index>*`]:`*<default-value>*

for nested lists:

<list-var>`[`*<index>*`,`*<index>* `<`*, ...>*`]:`*<default-value>*

index must be an integer and counts from 0 up to the number of elements-1.

It will return the *default-value* if you try to access an out-of-bounds element.

Note

Note that there must be no space between the closing bracket and the colon.

Examples:

```
{
  list l = [1, 2, 3];
  integer three = l[2]:0;         // == 3
  integer zero = l[42]:0;         // default value
  list ll = [[1,2], [3,4], [5,6]];
  return (ll[1,0]:0 == three); // returns true
}
```

3.6.7.2.2. Accessing maps

General syntax:

for simple maps:

<map-var>`[`*<key>*`]:`*<default-value>*

for nested lists:

<map-var>`[`*<key>*`,`*<key>* `<`*, ...>*`]:`*<default-value>*

key must have an allowed type for maps, integer, string, or symbol.

It will return *default-value* if you try to access an non existing key.

Note

Note that there must be no space between the closing bracket and the colon.

Examples:

```
{
  map m = $["a":1, "b":2, "c":3];
  integer three = m["c"]:0;         // == 3
  integer zero = m["notthere"]:0; // default value
  map mm = $["a":$[1:2], "b":$[3:4], "c":$[5:6]];
  return (mm["b",0]:0 == three); // returns true
}
```

3.6.7.2.3. Mixed map/list access

Since the bracket operator applies to list and maps, you can use it to access nested lists and maps. But be careful not to mix up the index/key types.

Examples:

```
{
  map map_of_lists = $["a":[1, 2, 3], "b":[4,5,6], "c":[7,8,9]];
  integer three = map_of_lists["a",2]:0;         // == 3
  list list_of_maps = [$[1:2], $[3:4], $[5:6]];
  return (list_of_maps[1,0]:0 == three);         // returns true
}
```

3.6.7.3. Assign variant

The bracket operator can also be used on the left side of an assignment (lvalue).

This changes the list or map element **in place (!!)** and must be used with care.

If the map or list element does not exist, it will be created. The bracket operator can therefore replace `add` and `change`.

Creating a new list element will extend the size of the list. Holes will be filled with `nil`. See the examples below.

If used as an lvalue, the default value is **not** allowed.

Examples:

```
{
  list l = [1,2,3];

  // change the second element
  l[1] = 25;                // l = [1,25,3] now !

  // change the "c" element
  map m = $["a":1, "b":2, "c":3];
  m["c"] = 42;                 // m = $["a":1, "b":2, "c":42] now

  // extend the list to 7 elements (0-6)
  l[6] = 6;                // l = [1,25,3,nil,nil,nil,6] now !

  // add a new element to m

  m["zz"] = 13;

  return (mm);                 // $["a":1, "b":2, "c":42, "zz":13]
}
```

3.7. Data Type *locale*

A string that is localized by YaST2 via the *gettext* mechanism (see gettext and ngettext in libc for more informations). Basically a string to be translated via gettext must be enclosed in `_(...)` which causes gettext to look up the translated string. It is even possible to distinguish singular and plural verbalizations depending on a parameter that denotes the actual number of something.

For a simple number-independent string you write `_(some_string_constant)` which causes its translation.

If the string to be translated needs to be different depending on the multiplicity of something then you write `_(singular_string_constant1, plural_string_constant2, actual_number)`. If `actual_number` equals 1 then the translation of the first string is used. Otherwise the translation of the second string is used.

Note 1: There are languages that distinguish more than the two cases singular and plural. Principally gettext can handle even those cases as it allows more than two strings for selection, but that is beyond the scope of this document (see the gettext documentation).

Note 2: It is not possible to put something other than string constants between the brackets.

Example 3.13. Locale constants

_("Everybody likes Linux!")
_("An error has occured.", "Some errors have occured", error_count)

3.8. YCP Program Structure

Now that we have learned how data can be stored and evaluated in `YCP`, we will take a look at the surrounding code structure that can be realized. Code structure is created by means of *blocks* and *statements*.

3.8.1. Comments

Despite not being “really” *statements*, comments do (and should) belong to the overall structure of a

YCP program. There are two kinds of comments:

- *Single-line comments*

  Single-line comments may start at any position on the line and reach up to the end of this line. They are introduced with “//”.

- *Multi-line comments*

  Multi-line comments may also start at any position on the line but they may end on another line below the starting line. Consequently there must be a start tag (“/*”) and an end tag (“*/”) as is shown below.

Example 3.14. Comments

```
{
   // A single-line comment ends at the end of the line.

   /*
     Multi-line comments
     may span several lines.
   */

   y2milestone("This program runs without error");
}
```

3.8.2. Variable Declaration

Synopsis: **data_type** *variable_name* = *initial_value***;**

Variable declarations in YCP are simliar to C. Before you can use a variable, you must declare it. With the declaration you appoint the new variable to be of a certain *data_type* which means you can assign only values of that specific type. To avoid any errors caused by unintialized variables, a declaration *must* imply a suitable value assignment.

Note: A variable declaration may occur at several points in the code which determines its validity (accessability) in certain program regions (see Section 3.8.15, “Variable Scopes and blocks”).

Example 3.15. Variable Declaration

```
{
  integer int_num     = 42;
  float   float_num   = 42.0;
  string  TipOfTheDay = "Linux is best!";
  integer sum         = 4 * (num + 8);
}
```

3.8.3. Variable Assignment

Synopsis: *variable_name* = *value***;**

An assignment statement is almost the same as a declaration statement. Just leave out the declaration. It is an error to assign a value to a variable that has not already been declared or to a variable of different data type.

Example 3.16. Variable Assignment

```
{
  integer number = 0;

  number = number + 1;
  number = 2 * number;
  number = "Don't assign me to integers!";     // This will cause an error!!!
}
```

3.8.4. Conditional Branch

Synopsis: **if (***condition***)** *then_part* [**else** *else_part*]

Depending on *condition* only one of the code branches *then_part* and *else_part* is executed. The *else_part* is optional and may be omitted. Both *then_part* and *else_part* may either be single statements or a sequence of statements enclosed in curly brackets, i.e. a block. The *then_part* is executed if and only if *condition* evaluates to `true`, the *else_part* otherwise. It is an error if *condition* evaluates to something other than `true` or `false`.

Example 3.17. Conditional branch

```
{
   integer a = 10;

   if ( a > 10 )
      y2milestone("a is greater than 10");
   else
   {
      // Multiple statements require a block...

      y2milestone("a is less than or equal to 10");
      a = a * 10;
  }
}
```

3.8.5. *while()* Loop

Synopsis: **while (***condition***)** *loop_body*

The **while()** loop executes the attached *loop_body* again and again as long as *condition* evaluates to `true`. The *loop_body* may be either a single statement or a block of statements.

Because *condition* is checked at the top of *loop_body*, it may not be executed at all if *condition* is `false` right from start.

Example 3.18. *while()* Loop

```
{
   integer a = 0;

   while (a < 10) a = a + 1;

   while (a >= 0)
   {
      y2milestone("Current a: %1", a);
      a = a - 1;
   }
}
```

3.8.6. *do..while()* Loop

Synopsis: **do** *loop_body* **while (***condition***);**

The **do...while()** loop executes the attached *loop_body* again and again as long as *condition* evaluates to `true`. The *loop_body* may be either a single statement or a block of statements.

Because *condition* is checked at the bottom of *loop_body*, it is executed at least once, even if *condition* is `false` right from start.

Example 3.19. *do...while()* Loop

```
{
   integer a = 0;

   do
   {
      y2milestone("Current a: %1", a);
      a = a + 1;
   } while (a <= 10);
}
```

3.8.7. *repeat..until()* Loop

Synopsis: **repeat** *loop_body* **until (***condition***);**

The **repeat...until()** loop executes the attached *loop_body* again and again as long as *condition* evaluates to `false`. The *loop_body* may be either a single statement or a block of statements.

Because *condition* is checked at the bottom of *loop_body*, it is executed at least once, even if *condition* is `true` right from start.

repeat...until() is similar to **do...while()** except that *condition* is logically inverted. The example below has been converted from the **do...while()** example above.

Example 3.20. *repeat...until()* Loop

```
{
   integer a = 0;

   repeat
   {
      y2milestone("Current a: %1", a);
      a = a + 1;
   } until (a > 10);
}
```

3.8.8. *break* Statement

Synopsis: **break;**

The **break** statement is used within loops to exit immediately. The execution is continued at the first statement after the loop.

Example 3.21. *break* statement

```
{
   integer a = 0;

   repeat
   {
      y2milestone("Current a: %1", a);
      a = a + 1;
      if (a == 7) break;              // Exit the loop here, if a equals 7.
   } until (a > 10);                  // Value 10 will never be reached.
   y2milestone("Final a: %1", a);     // This prints 7.
}
```

3.8.9. *continue* Statement

Synopsis: **continue;**

The **continue** statement is used within loops to abandon the current loop cycle immediately. In contrast to **break** it doesnt exit the loop but jumps to the conditional clause that controls the loop. So for a **while()** loop, it jumps to the beginning of the loop and checks the condition. For a **do...while()** loop or **repeat...until()** loop, it jumps to the end of the loop end checks the condition.

Example 3.22. *continue* statement

```
{
   integer a = 0;

   while (a < 10)
   {
      a = a + 1;
      if (a % 2 == 1) continue;     // % is the modulo operator.
      y2milestone("This is an even number: %1", a);
   }
}
```

3.8.10. *return* Statement

Synopsis: **return** [*return_value*]**;**

The **return** statement immediately leaves the current function or a current toplevel block (that contains it) and optionally assigns a *return_value* to this block. If blocks are *nested*, i.e. if the current block is contained in another block, the return statement leaves all nested blocks and defines the value of the outermost block.

However, if a block is used in an expression other than a block, and that expression is contained in an outer block, the **return** statement of the inner block won't leave the outer block but define the value of the inner block. This behaviour is a as one would expect. For example in the iteration builtins in Section 3.8.16, “Applying Expressions To Lists And Maps”,

Example 3.23. *return* statement

```
{
   // This block evaluates to 42.

   return 42;
   y2milestone("This command will never be executed");
}

{
   // This block evaluates to 18

   while (true)
   {
      return 18;
   }
}

{
   // This program evaluates to 3:

   integer a = 1 + { return 2; };
   return a;
}
```

3.8.11. Function definition

Synopsis: *data_type function_name* **(** [*typed_parameters*] **)** *function_body*

A function definition creates a new function in the current namespace named *function_name* with a parameter list *typed_parameters* that has *function_body* attached for evaluation. The *function_body* must return a value of type *data_type* and the arguments passed upon function call must match the type definitions in *typed_parameters*.

Example 3.24. Function definition

```
{
   void nothing()
    {
      y2milestone("doing nothing, returning nothing");
   }

   integer half( integer value )
    {
      return value / 2;
   }

   // This renders: ...nothing: nil  -  half: 21...

   y2milestone("nothing: %1  -  half: %2", nothing(), half(42) );
}
```

3.8.12. Function declaration

Synopsis: *data_type function_name* **(** [*typed_parameters*] **);**

A function declaration allows you to declare only a header of a function without its body. It's main purpose is for indirect recursion etc. You have to provide a function definition with exactly the same arguments later in the same file. A new function will be declared in the current namespace named *function_name* with a parameter list *typed_parameters*.

Example 3.25. Function declaration

```
{
   void nothing();

   integer half( integer value )
    {
      return value / 2;
   }

   // This renders: ...nothing: nil  -  half: 21...

   y2milestone("nothing: %1  -  half: %2", nothing(), half(42) );

   void nothing()
   {
      y2milestone("doing nothing, returning nothing");
   }

}
```

3.8.13. *include* Statement

Synopsis: **include "** *included file***";**

The include statement allows you to insert contents of a file at the given place in the current file. If the current file is a module, the contents of the included file will become a part of the module.

This is useful for dividing a large file into number of pieces. However, if a file is included more than once in a single block, the 2nd, 3rd etc. include statements are ignored.

The *included file* can be a relative or an absolute file name. Relative names are looked up with `/usr/share/YaST2/include` as a base.

Example 3.26. Include a file

```
// this will include /usr/share/YaST2/include/program/definitions.ycp
#include "program/definitions.ycp";
```

3.8.14. *import* Statement

Synopsis: **import "** *name space***";**

Not written yet...

3.8.15. Variable Scopes and blocks

In contrast to many other programming languages, `YCP` variables can be defined at (almost) any point in the code, namely between other statements. Given that, there must be some rules regarding the creation, destruction and validity of variables. Generally variables are valid (accessible) within the block they are declared in. This also covers nested blocks that may exist in this current block. The valid program region for a variable is called a *scope*.

Example 3.27. Variable scopes and blocks

```
{
   // Declared in the outer block
   integer outer = 42;

   {
      // Declared in the inner block
      integer inner = 84;

      // This is OK.
      // Log: ...IN: inner: 84 - outer: 42
      //
      y2milestone("IN: inner: %1 - outer: %2", inner, outer);
   }

   // This yields an error because "inner" is not defined any more.
   //
   y2milestone("OUT: inner: %1 - outer: %2", inner, outer);
}
```

3.8.16. Applying Expressions To Lists And Maps

Additionally to the structural language elements described so far, there are special commands that apply to `lists` and `maps`. What is special about these commands is that they apply an expression to the single elements of a list or map. This is done in a *functional* manner, i.e. the expression to be applied is passed as a parameter. Generally this executes faster than a *procedural* loop because the internal functionality is realized in a very effective way.

Furthermore some of these commands create lists from maps, maps from maps, maps from lists etc., so that they can be used to avoid the cumbersome assembling of these compound data types in a *procedural* loop.

3.8.16.1. *foreach()* Statement

Synopsis (list): `any` **foreach(** *type variable*, *list<type>*, *(expression)* **);**

Synopsis (map): `any` **foreach(** *type_key variable_key, type_value variable_value, map<type_key, type_value>, (expression)* **);**

This statement is a means to process the content of *list* or *map* in a sequential manner. It establishes an implicit loop over all entries of the list or map thereby executing the given *expression* with the respective entries. With lists the *variable* is a placeholder for the current entry. With maps, *variable_key* and *variable_value* are substituted for the respective key-value-pair.

Note 1: FIXME: Typing

Note 2: The return value of the last execution of *expression* determines the value of the whole **foreach()** statement.

Example 3.28. *foreach()* Loop

```
{
   // This yields 3
   foreach(integer value, [1,2,3], { return value; });

   // This yields 9
   foreach(integer key, integer value, $[1:1,2:4,3:9],
           { y2milestone("value: %1", value); return value; });
}
```

3.8.16.2. *listmap()* Statement

Synopsis: `map<type1, type2>` **listmap(** *type3 variable, list<type3>, (expression returning map<type1, type2>)* **);**

This statement is a means to process the content of *list* in a sequential manner. It establishes an implicit loop over all entries in *list* thereby executing the given *expression* with the respective entry. During execution *variable* is a placeholder for the current entry. For each element in *list* the expression is evaluated in a new context. The result of each evaluation MUST be a map with a single pair of key-value. All the returned key-value-pairs are assembled to form the new map that is returned.

Note: FIXME: Typing, break, continue

Example 3.29. *listmap()* statement

```
{
   // This results in $[1:"xy", 2:"xy", 3:"xy"]
   //
   map<integer, string> m1 = listmap(integer s, [1,2,3], ( $[s: "xy"]));

   // This results in $[11:2, 12:4, 13:6]
   //
   map<integer, integer> m2 = listmap(integer s, [1,2,3],
                    { integer a = s+10;
                        integer b = s*2;
                        list ret = [a,b];
                        return(ret); });

   y2milestone("map 1: %1  - map 2: %2", m1, m2);
}
```

3.8.16.3. *maplist()* Statement

Synopsis (map): `list<type1>` **maplist(** *type2 key, type3 value, map<type2, type3>, (expression returning type1)* **);**

Synopsis (list): `list<type1>` **maplist(** *type2 variable, list<type2>, (expression returning type1)* **);**

This statement is a means to process the content of *map* or *list* in a sequential manner. It establishes an implicit loop over all entries in *map* or *list* thereby executing the given *expression* with the respective entries. With lists the *variable* is a placeholder for the current entry. With maps, *key* and *value* are substituted for the respective key-value-pair. For each element the expression is evaluated in a new context. All return values are assembled to form the new list that is returned.

Note: FIXME: Typing, break, continue

Example 3.30. *maplist()* statement

```
{
   // This results in [2, 4, 6]
   list<integer> l1 = maplist(integer s, [1,2,3], (s*2));

   // This results in [2, 6, 12]
   list<integer> l2 = maplist(integer k, integer v, $[1:2, 2:3, 3:4], (k*v));

   y2milestone("list 1: %1  - list 2: %2", l1, l2);
}
```

3.8.16.4. *mapmap()* Statement

Synopsis: `map<type1, type2>` **mapmap(** *type3 key, type4 value, map<type3, type4>, (expression returning map<type1, type2>)* **);**

This statement is a means to process the content of *map* in a sequential manner. It establishes an implicit loop over all entries in *map* thereby executing the given *expression* with *key* and *value* substituted for the respective key-value-pair. For each map element the expression is evaluated in a new context. The result of each evaluation MUST be a map with a single pair of key-value. All the returned key-value-pairs are assembled to form the new map that is returned.

Example 3.31. *mapmap()* statement

```
{
   // This results in $[11:"ax", 12:"bx"]
   //
   map<integer,string> m = mapmap(inetger k, string v, $[1:"a", 2:"b"], ([k+10, v+"x"]));

   y2milestone("map: %1", m);
}
```

3.9. Controlling The User Interface

In the previous sections we have already seen some `YCP` code that dealt with the creation and handling of onscreen dialogs. These examples were rather simple to show the *basic* strategy of creating dialogs. Of course designing “real” dialogs that do useful things is is bit more complicated and requires a rather good knowledge of the instuments provided by the UI.

Because the UI has been designed to be most flexible, the possibilities for creating and managing dialogs are quite versatile. Consequently the instruments for doing this are rather diverse. In fact the UI extends the basic `YCP` language to a large extent, thereby providing the means to create and manage onscreen dialogs.

For more information about the User Interface, handling events see the Layout HOWTO.

3.10. The YaST Wizard

In the previous section the basic mechanisms suitable for managing onscreen dialogs were presented. However, creating dialogs for each and every application from scratch would be cumbersome and moreover is unnecessary. For example the well-known layout that is presented with nearly every `YaST` dialog during installation was not *programmed* anew for each dialog. Rather it is kind of imported from a special `YCP` module, the *Wizard* that provides all the functionality necessary to create uniform dialogs.

Furthermore, as time went by, during the development of the `YaST` installer, the developers encounterd situations where the same (or similar) tasks ofttimes had to be accomplished at different locations in the overall program flow. For example opening a popup to ask the user a question with the predefined buttons “Yes” and “No” is a procedure used very often.

This led to the development of predefined dialog elements that can (and should) be included in the current `YCP` source. Displaying the Yes-No-popup from the example above is then reduced to calling a function with respective parameters. Aside from avoiding the need to redevelop such things again and again, another benefit is the ever same visual appearance that adds up to the well-known `YaST` look-and-feel. Meanwhile there are also many functions that are not UI-related but nonetheless very useful.

The omnium gatherum of all these elements has been collected to form the so-called “`YaST` Wizard”. In short the `YaST` Wizard consists of one `YCP` module that provides the layout framework used in the installation dialogs and some additional `YCP` modules that provide access to several common dialog elements needed rather often. Many “generic” functions are at hand as well.

The following two sections cover these topics mostly by means of references to the `YaST` developers documentation.

Chapter 4. Running y2base Stand-Alone

In the previous section you got to know how to do a YaST program. Well, normally YCP-scripts are executed involving the whole `YaST`-machinery, e.g. during installation, which requires correct embedding of the script into the surrounding `YCP` environment. Fortunately there is a way to let run `YCP`-scripts isolated, i.e. stand-alone.

To do so we make use of the architectural separation of components featured by `YaST`. The "command line version" of `YaST` is called **y2base** and can usually be found in `/usr/lib/YaST2/bin`. You could set the `PATH` to include this location to avoid typing in the full path every time.

```
$> y2base -h

Usage: y2base [LogOpts] Client [ClientOpts] Server [Generic ServerOpts] [Specific ServerOpts]
LogOptions are:
    -l | --logfile LogFile    : Set logfile
ClientOptions are:
    -s                        : Get options as one YCPList from stdin
    -f FileName               : Get YCPValue(s) from file
    '(any YCPValue)'          : Parameter _IS_ a YCPValue
Generic ServerOptions are:
    -p FileName               : Evaluate YCPValue(s) from file (preload)
    '(any YCPValue)'          : Parameter _IS_ a YCPValue to be evaluated
Specific ServerOptions are any options passed on unevaluated.

Examples:
y2base installation qt
    Start binary y2base with intallation.ycp as client and qt as server
y2base installation '("test")' qt
    Provide YCPValue '"test"' as parameter for client installation
y2base installation qt -geometry 800x600
    Provide geometry information as specific server options
```

This help page, showing the possible options in a call of **y2base**, is rather self-explaining. For the moment the interesting parameters are *Client* and *Server*. In Section 2.2.4, "External Programs" we learned that `YaST` consists of several modules, some of them being *client-components* and some others being *server-components*. By invoking `YaST` in the way displayed above we can connect any client-component with any server-component.

Because a `YCP`-program (also called `YCP`-module) can act as a client-component, it is possible to connect it with a server-component suitable of executing it. Since our "Hello, World!"-program displays something on screen, we need to use the UI as server-component in this case. As already said the UI is able to use a text-based console environment as well as a graphical X11 environment which leads to the following two methods of running a `YCP`-script.

- **y2base file.ycp qt**

  This will excute `file.ycp` in the graphical Qt-UI.

- **y2base file.ycp ncurses**

  This will excute `file.ycp` in the text-based NCurses-UI.

Chapter 5. SCR Details

In the intoductional chapter we have already heard something about accessing the system with SCR. Because manipulating the system at the lowest layer is all `YaST` is about, we now want to take a closer look at this topic.

Basically the SCR creates a consistent view of the system hardware and its configuration files. There are many dependencies between the different entities among those data and these dependencies have to be taken into consideration when manipulating them. For this being possible in a convenient way for the higher-level modules there must be an easy and consistent accessing method. This method is provided by the SCR as it presents kind of an abstraction of the various types of data to be handled.

Now the data “landscape” that must be covered here is rather heterogeneous. Hardware data and configuration data, both of most multifaceted type can hardly be handled by one single monolithic program. Therefore the SCR consists of a “head” that is accompanied by various helper programs, the so-called agents, each of them being specialized on a specific task.

5.1. SCR Agents

For each category of system data there is a corresponding SCR-agent. Their job is to map the real system data to `YCP`-data structures so that `YCP`-modules can access them in a convenient way. In fact the SCR-agents *provide* the `YCP` data structures. They come into existence with the presence of an SCR-agent that provides them. Otherwise they wouldn't be there.

For example there is an agent that reads and writes the `/etc/sysconfig` files. The `YCP`-representation of a `sysconfig`-variable is a single `YCP`-string. When reading, the agent reads the variable in the corresponding file and creates a `YCP`-string from it. When writing, the agent gets the new value as `YCP`-string and changes the variable in the corresponding file accordingly.

It must be said here, that the set of agents may change over time. New agents may be created in the future and other ones might be abandoned if their functionality is obsolete or taken over by another agent. Generally this is no problem because for module development it is not (and should not be) necessary to know exactly which agent does what. As already said, the SCR provides an abstraction of the data to be handled and this abstraction comes into being in form of a tree, the SCR-tree.

5.2. SCR Tree

As a computer's hardware and software configuration is quite complex, the SCR organizes all data in form of a tree. It resembles very much a filesystem with its folders, sub-folders and files whereby the tree structure reflects the thematical separation of the various configuration categories.

The SCR-tree consists of two different kinds of nodes:

Table 5.1. SCR Node Types

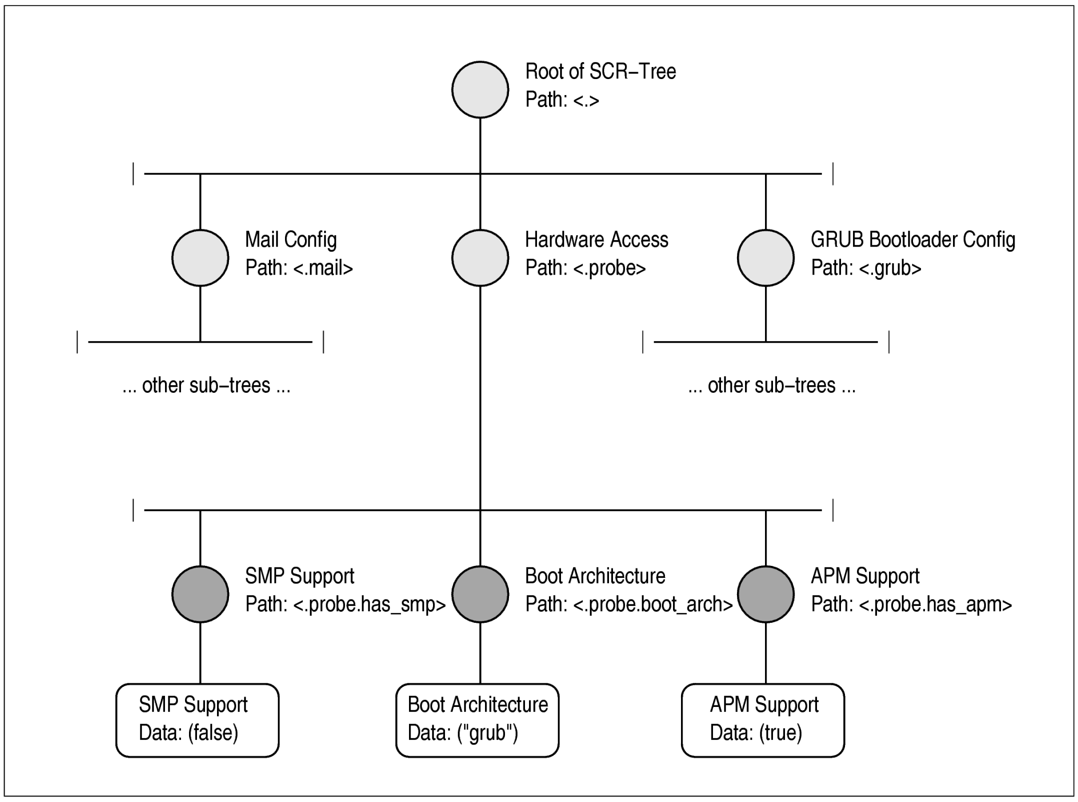
Data nodes	Data nodes represent single pieces of data, for example a `sysconfig`-entry or a mountpoint of a filesystem in `/etc/fstab`. They are the *leaves* of the tree and stand for actual data to be handled.
Map nodes	Map nodes allow for navigation to the leaves just like the path components in the directory structure of a file system. This way map nodes are used to structure the data in a suitable manner.

The names of the nodes in the SCR-tree can be concatenated resulting in the creation of an *SCR-path*. An SCR-path is a description were to find a node in the SCR-tree. It is a sequence of path components each of them being a string. As we have seen in the section Data type path `YCP`-paths are prepended by dots (.) which act as separators in compound paths. So `.foo.bar` is a valid `YCP`-

path. If `bar` is an SCR data node, then `SCR::Read(.foo.bar)` would render some data. If `bar` is a map node, then `SCR::Dir(.foo.bar)` would reveal the immediate sub-nodes in the SCR-tree, e.g. `["big", "brown", "fox"]`. The single dot (.) is also valid and denotes the root of the whole SCR-tree. Consequently `SCR::Dir(.)` will return a list of all the top-nodes in the SCR-tree.

In the figure below we see a (very small) cut-out of the SCR-tree that is related to hardware-specific information.

Figure 5.1. SCR Hierarchy Tree



The light grey nodes are SCR-map-nodes denoting the path to the data. They can be used with `SCR::Dir(...)` to find out what is below. So in the figure above `SCR::Dir(.probe)` would return a list as `[..."has_smp", "boot_arch", "has_apm"...]`.

The dark grey nodes are SCR-data-nodes that stand for the actual data. What can be done with them depends on the actual node (reading, writing, executing), but usually `SCR::Read(...)` is possible. As is shown above `SCR::Read(.probe.boot_arch)` would return `"grub"`.

5.3. Accessing SCR

Now that we know how the SCR-landscape can be navigated we will take a look at how the data that is dug in there can be accessed. The accessing methods already implied above shall now be defined more precisely.

There are four accessing methods.

- *Reading*

  We speak of *reading*, when the agent reads some configuration file or scans the system hardware and produces a YCP data structure representing this information.

- *Writing*

  We speak of *writing*, when the agent gets some YCP data structure and creates or modifies some system config file according to these data.

- *Executing*

  We speak of *executing*, when the agent gets some YCP data that can be interpreted as instruction and executes it. Usually this is being done by means of another program, e.g. the `bash`.

- *Dir*

  Compareared to the other accessing methods listed above, the *Dir*-command is somewhat special. It takes as argument an SCR-path that points to a specific node in the SCR-tree. It returns a list of all the sub-paths that are immediately below this node. This way it works just like a *dir*-command in a file system. For example if you apply this to the root of the SCR-tree (.), the answer would be a list with all the top nodes known by the SCR , e.g. `["audio", ... , "yast2"]`. [1]

As a rule all SCR-agents implement some of the four accessing methods listed above. However depending on the task the agent was made for, not *all* of them may be provided.

For convenient use from `YCP` the accessing methods are realized by means of an *API*, i.e. a defined set of `YCP`-functions that are understood. You can call these functions from `YCP` if you prepend the commands with the name space identifier `SCR::` which causes redirection to the SCR.

Table 5.2. The SCR-commands

Function	What it does
`Read(path p) -> any`	Reads the data represented by the node at path `p`. The value returned can be any `YCP` data type but it is always one single value.
`Write(path p, any v) -> boolean`	Writes the value `v` to the node at path `p`. The boolean return value is `true` on success. On error the return value is `false` and a log entry is generated in the log file. Reasons for errors can be a mistyped value `v` or some problem with the periphery that lies behind the data-node.
`Execute(path p) -> boolean`	This command is mostly used with the `system-agent` (see Section 5.6, “Useful SCR Agents”). Usually the return value indicates success or failure of the executed command.
`Dir(path p) -> list(string)`	Returns a list of all subtree nodes immediately below the node `p`. For each such node the list contains a string denoting its name. If `p` does not point to a map node, i.e. the last path component is a leaf, the command will return an empty list or `nil`.

5.4. Using SCR From Within YCP

FIXME: To be done... (Examples?)

[1] Unfortunately not all SCR-agents do support this command properly. There may be agents that wrongly return an empty list or even `nil` when queried this way.

5.5. Using SCR From The Command Line

In the last section we saw some examples of how the SCR can be used from `YCP`. However if you only want to test or explore different SCR-paths, writing a `YCP`-script for every access can be cumbersome. Fortunately the SCR-component can be run individually on the command line of a terminal using a method very similar to the one we saw in Chapter 4, *Running **y2base** Stand-Alone*.

In contrast to the method demonstrated there, this time we don't feed a `YCP`-script into `YaST`. Instead we make another use of the architectural separation of components featured by `YaST` in that we connect the so-called *stdio-component* with the *SCR-component*. By doing so we can feed everything we type on the command line into the SCR.

However, because of the “raw” nature of the `YaST`-internal communication paths, this method is not very comfortable. You can't correct typos with **Backspace** or **Del** here (the SCR is not *meant* to be operated in this way). By doing so we kind of “simulate” `YaST`-internal communication which normally forecloses any misspelling.

Furthermore, if you play around with the SCR in this manner you will be able to initiate privileged actions only if you are running the commands under the *root-account*.

Caution

If you run “manual” SCR-commands under the root-account, the SCR will “gracefully” fulfill all your wishes. So be careful with *Write* and *Execute*!!!

Now operating the SCR this way can be shown best with some examples.

Example 5.1. Operating the SCR from the command line

```
$> /usr/lib/YaST2/bin/y2base stdio scr
([])
Read(.probe.boot_arch)
("grub")
Read(.probe.version)
("Oct  7 2002, 15:05:08")
Read(.probe.has_smp)
(false)
Read(.probe.has_apm)
(true)
SCR::Read(.probe.boot_arch)
(nil)
```

As is shown above, the command **y2base stdio scr** starts `YaST` in a specific way. It connects the `YaST` client-component *stdio* with the server-component *scr*. After that the SCR is running and awaits any input on stdio which in this case is the console. To explore the content of the SCR-tree you can now enter any SCR-commands just as you would do in `YCP`. The only difference is the abscence of the `SCR::` name space identifier which must not be given here as can be seen in the last line.

5.6. Useful SCR Agents

The SCR-world knows many agents for all sorts of tasks. Unfortunately this matter is subject to a rather high change service and not (yet) well documented. Therefore it is not easily possible to explain the details in a manner of “Which agent provides which paths for what reason?”. As a result only the most helpful agents are mentioned here along with references to the respective developers documentation.

Please note that even the developers documentation might be outdated to some extent. Consequently the most reliable source of information are the “real” files below `/usr/share/YaST2/`.

- System Agent

  This agent realizes access to the target system during installation.

  /usr/share/doc/packages/yast2-core/agent-system/ag_system-builtins.html [instdoc_subset/yast2-core/agent-system/ag_system-builtins.html] / usr/share/doc/packages/yast2-core/agent-system/ag_system-builtins.html

- Background Agent

  This agent runs shell commands in the background.

  /usr/share/doc/packages/yast2-core/agents-perl/ag_background.html [instdoc_subset/yast2-core/agents-perl/ag_background.html] / usr/share/doc/packages/yast2-core/agents-perl/ag_background.html

- Hardware Probe Agent

  The agent being responsible for hardware probing.

  /usr/share/doc/packages/yast2-core/agent-probe/hwprobe.html [instdoc_subset/yast2-core/agent-probe/hwprobe.html] / usr/share/doc/packages/yast2-core/agent-probe/hwprobe.html

- Any-Agent

  This agent handles the access to configuration files of (almost) arbitrary syntax. The syntax to be understood must be specified in a configuration file.

  /usr/share/doc/packages/yast2-core/agent-any/anyagent.html [instdoc_subset/yast2-core/agent-any/anyagent.html] / usr/share/doc/packages/yast2-core/agent-any/anyagent.html

- Ini-Agent

  The Ini-agent is suitable for accessing configuration files with the well-known ini-file syntax.

  /usr/share/doc/packages/yast2-core/agent-ini/ini.html [instdoc_subset/yast2-core/agent-ini/ini.html] / usr/share/doc/packages/yast2-core/agent-ini/ini.html

- Modules Agent

  This agent is the interface to the `/etc/modules.conf` file.

  /usr/share/doc/packages/yast2-core/agent-modules/modules.html [instdoc_subset/yast2-core/agent-modules/modules.html] / usr/share/doc/packages/yast2-core/agent-modules/modules.html

- Perl Agent

  This agent is a means to call Perl scripts from within `YCP`.

  /usr/share/doc/packages/yast2-core/agents-perl/ycp-pm.html [instdoc_subset/yast2-core/agents-perl/ycp-pm.html] / usr/share/doc/packages/yast2-core/agents-perl/ycp-pm.html

Chapter 6. YaST Modules

Creating modules for YaST means extending its functionality. For this being possible it is necessary to follow the infrastructural and functional particularities of YaST as well as some guidelines regarding the interaction of the module with the user and the rest of the system. In the following we'll have a closer look at these topics .

6.1. YCP Modules Overview

Throughout this document the term “YCP module” was mentioned repeatedly without providing a sharp definition. In fact the term “module” is used quite loosely in the YaST world, because there are several *kinds* of modules involved in different contexts. The following text shall lighten this topic.

Different kinds of YCP modules

- Generic YCP modules

  In principle every YCP file that provides a distinct functionality can be seen as a module. Typical representatives of this kind of module are the `inst_xxx.ycp` files that are part of the YaST installer. Modules of this kind mostly represent rather self-contained functionality, e.g like `inst_keyboard.ycp` that provides the user dialog for selecting a keyboard during installation. These modules are usually called via `CallFunction()`.

- Library modules

  This kind of module can be seen as what is called a library in other programming languages. Usually these modules are a collection of functions that must be *included* to be used. As with other programming languages, *including* in YCP means merely text insertion that takes place each and every time an `include` is stated. This is often adverse with respect to speed and memory consumption.

- True YCP modules

  This kind of module is the most interesting one. *True modules* represent an “object oriented” approach to module design. Because the mechanisms associated with them deserve some special mention, the next section will cover this topic in more detail.

6.2. True YCP Modules

True modules are rather new in the YaST world and it is planned that they will replace the old method of *including* modules completely (with exception of some rare cases perhaps). The following sections will outline the differences between these concepts.

6.2.1. Included Modules

YCP, originally planned as a functional language, always did dynamic (i.e. runtime) binding of variables. Although useful in many cases, it's quite puzzling for someone used to “imperative” languages. So you could well program the following block and get an unexpected result.

```
{
   integer x = 42;
   define f() ``{ return x; }
   ... // lots of lines
   x = 55;
   return f();  // will return 55 because of runtime binding of x!
}
```

Another widely misused feature is to include global definitions. While there was no alternative as long as `include` was the only referencing instrument, this is certainly not a good programming practice in view of speed and memory considerations.

6.2.2. True Modules (Imported Modules)

In contrast to included modules, true modules have some distinct properties that are shown in the list below.

- Definition-time bindings

  Definitions are evaluated in the sequence of the program flow.

- One-time inclusion

  In contrast to `include` the `import` statement includes the module only once even if there are more than one `import` statement in the program flow. Later imports are silently ignored.

- Proprietary global namespace

  The module definition implies a module declaration that determines the namespace of the module's global variable scope.

- Local environment

  Aside from the data located in the module's global namespace all other data defined in the module is purely *local*, i.e. is invisible from the outside.

- Module constructor function

  Each true module may have a constructor function that is automatically executed upon first import.

The following listing is a brief sample of a true module.

```
{
   // This is a module called "Sample".
   // Therefore the file name MUST be Sample.ycp
   // The "module" statement makes the module accessible for 'import'.
   //
   module "Sample";

   // This is a local declaration.
   // It can only be 'seen' inside the module.
   //
   integer local_var = 42;

   // This is a global declaration.
   // It can be accessed from outside with the name space identifier 'Sample::'.
   //
   global integer global_var = 27;

   // This is a global function.
   // It has access to global_var *and* local_var.
   //
   global define sample_f () ``{ return local_var + global_var; }
}
```

The module above can be used with the `import` statement. The syntax for file inclusion with `import` is similar to `include`. The interpreter automatically appends “.ycp” to the filename and searches below `/usr/lib/YaST2/modules`. If the filename starts with “./”, the file is loaded from the local directory. The global declarations of the module can then be accessed with the name space identifier `Sample::`.

Note

The file name *must* match the module declaration! Inside modules, only variable or function declarations are allowed. Stand-alone blocks or any kind of evaluation statements are forbidden.

```
{
    // This imports the 'Sample'-module.
    //
    import "Sample";

    // The global function is called with the respective name space identifier.
    //
    integer i = Sample::sample_f();     // == 69

    // No access to local module variables.
    //
    i = Sample::local_var;              // ERROR, no access possible !

    // No problem with global variables.
    //
    i = Sample::global_var;             // == 27

    Sample::global_var = 0;             // This variable is writable !!

    return Sample::sample_f();          // == 42, since global_var is 0
}
```

Note

The first encounter of the statement `import "Sample";` triggers the loading of “Sample.ycp”. Subsequent import statements are ignored, because “Sample” is already defined. Consequently you can't replace a module during runtime !

6.2.3. True Modules And Constructors

If a global function with the same name as the module is defined, it is treated as a constructor. The constructor is called after the module has been loaded *and evaluated* for the first time. Because of this the constructor could (and should) be defined at the beginning of the module. Despite being located “on top” it can make use of the functions declared later in the file.

Module constructors are used mostly for initialization purposes, e.g. for setting local variables to proper values. However, the actions within a constructor can be arbitrarily complex.

Note

Constructors can't have any arguments. The result of calling a constructor from the outside is ignored.

```
{
   // This is a module called "Class" with a constructor function.
   //
   module "Class";

   // A globally accessible variable.
   //
   global integer class_var = 42;

   // This is the constructor (same name as the module).
   //
   global define Class() ``{ class_var = 12345; }
}


{
    // The usage of the "Class"-module.
    //
    import "Class";

    return Class::class_var;            // will be 12345 !
}
```

6.3. Some Rules

Most often when a YaST module shall be created, this module will have some interaction with the user. This usually implies the creation of dialogs to be displayed on screen. As you might have noticed the dialogs that come ready-made with YaST follow a distinct “look and feel” which is due to the fact that the YaST developers follow some rules regarding the visual appearance as well as the functional behaviour of a dialog. The keywords here are usability and GUI-consistency.

6.3.1. Usability

When it comes to user-interaction one concept that is stressed very often is *usability* or - more speaking - user-friendliness. If you have ever heard s.th. about ergonomics you may also know the term *Human Computer Interaction* (HCI). For us regular folks *usability* is probably the best notation because it best summarizes what's it all about. It means that the program in question is good “usable” by the user. In general that means that operating a screen dialog should enjoin as low a burden as possible on the user.

In order to have a good usability a system should satisfy the following criteria:

- Users must be able to accomplish their goal with minimal effort and maximum results.
- The system must not treat the user in a hostile fashion or treat the user as if they do not matter.
- The system can not crash or produce any unexpected results at any point in the process.
- There must be constraints on the users actions.
- Users should not suffer from information overload.
- The system must be consistent at every point in the process.
- The system must always provide feedback to the user so that they know and understand what is happening at every point in the process.

Important

If you want to create an interactive YaST module you should try to heed those rules to ease the users life and to assure your module fits smoothly into the surroundig YaST environment which (hopefully) follows them too.

All that said above in essence is an outline from a very good article by Todd Burgess. If you are interested in a more elaborate discussion of usability you may have a look at http://www.osOpinion.com/Opinions/ToddBurgess/ToddBurgess1.html [http://www.osOpinion.com/Opinions/ToddBurgess/ToddBurgess1.html]

6.4. Module Layout

FIXME: To be done...

6.4.1. Module Skeleton

FIXME: To be done...

6.4.2. Module Example

FIXME: To be done...

Chapter 7. YaST2 UI Layout and Events

7.1. YaST2 Layout

7.1.1. Summary: What's This All About?

This is both a tutorial and a reference on how to lay out YaST2 dialogs.

Since experience shows that most people begin this with only a vague perception of YaST2's concepts, it also includes some basics that might be covered in other YaST2 documentation as well - just enough to get started.

7.1.2. Basics and Terms

If you are in a hurry or - like most developers - you don't like to read docs, you can skip this section and move right on to the next section. That is, if you think you know what the next few headlines mean. You can always come back here later.

Just don't ask anything that is explained here on the *yast2-hackers* mailing list - you'll very likely just get a plain *RTFM* answer shot right back into your face. And *this is the FM,* so read it if you need explanations. ;-)

7.1.2.1. The UI

The UI (user interface) is that part of YaST2 that displays dialogs. It is a separate process which uses a separate interpreter. Always think of it as something running on a different machine: There is the machine you want to install with YaST2 (i.e. the machine where the disks will be formatted etc.) and there is the machine that displays the dialogs - the UI machine. In most cases, this will actually be the same machine. But it doesn't need to be. Both parts of YaST2 might as well run on different machines connected via a serial line, a network or by some other means of communication (telepathy? ;-)).

The logical consequence of this is that the UI uses its own separate set of function definitions and variables. You need to be real careful not to mix that up. Always keep in mind what part of YaST2 needs to do what and what variables need to be stored where. You can easily tell by the `UI` prefix within the YCP code what parts are getting executed by the UI.

7.1.2.2. Widgets

A *widget* is the most basic building block of the UI. In short, each single dialog item like a PushButton, a SelectionBox or a TextEntry field is a widget. But there are more: Most static texts in dialogs are widgets, too. And there are a lot of widgets you can't see: Layout boxes like *HBox* or *VBox* and many more that don't actually display something but arrange other widgets in some way.

See the widget reference for details and a list of all available widgets.

7.1.2.3. UI Independence and the *libyui*

There are several different UIs for YaST2. There is the Qt [http://www.trolltech.com/products/index.html] based UI (*y2qt*) as a graphical frontend which requires the X Window System; this is what most people know as the "normal" YaST2 UI. But there is also a NCurses based UI (*y2ncurses*) for text terminals or consoles. A web UI (*y2web*) is being developed at the time of this writing.

That means, of course, that all YaST2 dialogs need to be written in a way that is compatible with each of those UIs. This is why *libyui* was introduced as an intermediate abstract layer between the

YCP application and the UI. You do not communicate directly with either *y2qt*, *y2ncurses* or *y2web* - you communicate with the libyui.

Thus, YaST2 dialogs need to be described logically rather than in terms of pixel sizes and positions: You specify some buttons to be arranged next to each other rather than at positions (200, 50), (200, 150), (200, 200) etc. - whatever exactly this "next to each other" means to the specific UI.

Add to that the fact that there are several dialog languages to choose from: User messages or button labels have different lengths in different languages. Just compare the length of English messages to those in German or French, and you'll discover another good reason not to hard-code coordinates.

In addition to that, always keep in mind that *the same dialog might require a different amount of space in a different UI*. Overcrowded dialogs don't look good in the Qt UI. In the NCurses UI, they will very likely break completely: There simply isn't as much space available (80x25 characters vs. 640x480 pixels).

7.1.2.4. The Nice Size

Each widget has a so-called *nice size* - this is the size the widget would like to have in order to look nice. E.g. for PushButtons that means the entire button label fits into the button. Likewise for labels.

Then there are widgets that don't have a natural nice size. For example, what size should a SelectionBox get? It can scroll anyway, so anything that makes at least one line of the list visible will satisfy the basic requirements. More space will make it look nicer; but how much is enough? The widget cannot tell that by itself.

Such widgets report a somewhat random size as their *nice size*. This is a number chosen for debugging purposes rather than for aesthetics. You almost always need to specify the size from the outside for that very reason. Always supply a weight for such widgets or surround them with spacings.

7.1.2.5. Initial Dialog Sizes

By default, all dialogs will be as large as they need to be - up to full screen size (which is UI dependent - 640x480 pixels for Qt, 80x25 characters for NCurses): The outermost widget is asked what size it would like to have, i.e. its nice size. If that outermost widget has any children, for example because it is a layout box, it will ask all of its children and sum up the individual sizes. Those in turn may have to ask their children and so on. The resulting size will be the dialog's initial size - unless, of course, this would exceed the screen size (UI dependent, see above).

7.1.2.6. Full Screen Dialogs: `opt(`defaultsize)

You can force full screen size for any dialog by setting the `` `defaultsize `` option when opening it:

```
OpenDialog(
            `opt(`defaultsize ),
            `VBox(...)
           );
```

This will create a dialog of 640x480 pixels (*y2qt*) or 80x25 characters (*y2ncurses*) - regardless of its contents.

Use this for main windows or for popup dialogs with very much the same semantics - e.g. many of the YaST2 installation wizard's "expert" dialogs. Even though they are technically popup dialogs and they return to the main thread of dialog sequence they have main window semantics to the user.

Use your common sense when considering whether or not to use this feature for a particular dialog.

Note: Not every UI may be capable of this feature. This is only a hint to the UI; you cannot blindly rely on it being honored.

7.1.3. Layout Building Blocks

This section covers the widgets used for creating dialog layouts - the kind of widgets that are less obvious to the user. If you are interested in the "real" widgets, i.e. the kind you can actually see, please refer to the widget reference.

7.1.3.1. Layout Boxes: HBox and VBox

This is the most basic and also the most natural layout widget. The HBox widget arranges two or more widgets horizontally, i.e. left to right. The VBox arranges two or more widgets vertically, i.e. top to bottom.

The strategy used for doing this is the same, just the dimensions (horizontal / vertical) are different. Each child widget will be positioned logically next to its neighbor. You don't have to care about exact sizes and positions; the layout box will do that for you.

See the description of the layout algorithm for details.

For creating more complex layouts, nest HBox and VBox widgets into each other. Usually you will have a structure very much like this:

```
`VBox(
      `HBox(...),
      `HBox(...),
      ...
      )
```

i.e. a VBox that has several HBoxes inside. Those in turn can have VBoxes inside etc. - nest as deep as you like.

Almost every kind of layout can be broken down into such columns (i.e. VBoxes) or rows (i.e. HBoxes). If you feel you can't do that with your special layout, try using weights.

7.1.3.2. Specifying Proportions: HWeight and VWeight

By default, each widget in a layout box (i.e. in a HBox or a VBox) will get its nice size, no more and no less. If for any reason you don't want that, you can exactly specify the proportions of each widget in the layout box. You do that by supplying the widgets with a *weight* (to be more exact: by making it the child of a weight widget, a HWeight or a VWeight).

You can specify percentages for weights, or you can choose random numbers. The layout engine will add the weights of all children of a layout box and calculate percentages for each widget automatically. Specify a *HWeight* for *HBox* children and a *VWeight* for *VBox* children.

Example 7.1. Specifying Proportions 1

```
`HBox(
      `HWeight( 20, `PushButton( "OK"     ) ),
      `HWeight( 50, `PushButton( "Cancel" ) ),
      `HWeight( 30, `PushButton( "Help"   ) )
     )
```

In this example, the "OK" button will get 20%, the "Cancel" button 50% and the "Help" button 30% of the available space. In this example, the weights add up to 100, but they don't need to.

Note: *This dialog looks extremely ugly - don't try this at home, kids* ;-)

The weight ratios will be maintained at all times, even if that means violating nice size restrictions (i.e. a widget gets less space than it needs). You are the boss; if you specify weights, the layout engine assumes you know what you are doing.

Example 7.2. Specifying Proportions 2

(See also creating widgets of equal size in the common layout techniques section)

```
`HBox(
      `HWeight( 1, `PushButton( "OK"     ) ),
      `HWeight( 1, `PushButton( "Cancel" ) ),
      `HWeight( 1, `PushButton( "Help"   ) )
     )
```

Note: *This is a very common technique*.

In this example all buttons will get an equal size. The button with the largest label will determine the overall size and thus the size of each individual button.

Please note how the weights do not add up to 100 here. The value "1" is absolutely random; we might as well have specified "42" for each button to achieve that effect.

Example 7.3. Specifying Proportions 3

The YaST2 wizard layout reserves 30% of horizontal space for the help text (a RichText widget) and the remaining 70% for the rest of the dialog. The important part of that code (simplified for demonstration purposes) looks like that:

```
`HBox(
      `HWeight( 30, `RichText( "Help text") ),
      `HWeight( 70, `VBox(
                          ...   // the dialog contents
                          `HBox(
                                `PushButton( "Back"),
                                `HCenter( `PushButton( "Abort Installation") ),
                                `PushButton( "Next")
                               )
                          )
              )
      )
```

Specifying the size of the help text like that is important for most kinds of widgets that can scroll - like the RichText widget used here, for example. The RichText widget can take any amount of space available; it will wrap lines by itself as long as possible and provide scroll bars as necessary. Thus, it cannot supply any reasonable default size on its own - you must supply one. We chose 30% of the screen space - which of course is absolutely random but suits well for the purposes of YaST2.

Use this technique for widgets like the SelectionBox, the Table widget, the Tree widget, the Rich-Text, but also for less obvious ones like the TextEntry.

Note: This list may be incomplete. Use your common sense.

7.1.3.3. Rubber Bands: HStretch and VStretch

When you don't want parts of a dialog to be resized just because some neighboring widget needs more space, you can insert *stretch* widgets to take any excess space. Insert a HStretch in a HBox or a VStretch in a VBox. Those will act as "rubber bands" and leave the other widgets in the corres-ponding layout box untouched.

You can also insert several stretches in one layout box; excess space will be evenly distributed among them.

If there is no excess space, stretch widgets will be invisible. They don't consume any space unless there is too much of it (or unless you explicitly told them to - e.g. by using weights).

7.1.3.4. Making Common Widgets Stretchable: `opt(`hstretch) and `opt(`vstretch)

Some widgets that are not stretchable by default can be made stretchable by setting the

`` `hstretch `` option. PushButtons are typical candidates for this: They normally consume only as much space as they really need, i.e. their nice size. With the `` `hstretch `` option, however, they can grow and take any extra space - very much like stretch widgets.

Please note, however, that all widgets for with a weight are implicitly stretchable anyway, so specifying `` `opt(`hstretch) `` or `` `opt(`vstretch) `` for them as well is absolutely redundant.

7.1.3.5. Spacings: HSpacing and VSpacing

Use HSpacing or VSpacing to create some empty space within a layout. This is normally used for aesthetical reasons only - to make dialogs appear less cramped.

The size of a spacing is specified as a float number, measured in units roughly equivalent to the size of a character in the respective UI (1/80 of the full screen width horizontally , 1/25 of the full screen width vertically). Fractional numbers can be used here, but text based UIs may choose to round the number as appropriate - even if this means simply ignoring a spacing when its size becomes zero.

You can combine the effects of a spacing and a stretch if you specify a hstretch or a vstretch option for it: You will have a rubber band that will take at least the specified amount of space. Use this to create nicely spaced dialogs with a reasonable resize behaviour.

Example 7.4. Spacings

```
`HBox(
      `PushButton( "OK" ),
      `HSpacing( `opt(`hstretch), 0.5),
      `PushButton( "Cancel" )
     )
```

This will create two buttons with a spacing between them. When the dialog is resized, the spacing will grow.

7.1.3.6. Alignments: Left, Right, HCenter, Top, Bottom, VCenter, HVCenter

Alignments are widgets that align their single child widget in some way.

HCenter centers horizontally, VCenter centers vertically, HVCenter centers both horizontally and vertically. The others align their child as the name implies.

More often than not, you could achieve the same effect with a clever combination of spacings, but sometimes this might require an additional HBox within a HBox or a VBox within a VBox, i.e. more overhead.

7.1.3.7. Compressing Excess Space: HSquash, VSquash, HVSquash

Sometimes you wish to squeeze any extra space from a part of a dialog. This might be necessary if you want to draw a frame around a RadioBox in a defaultsize dialog: You want the frame drawn as close as possible to the RadioButtons, not next to the window frame with lots of empty space between the frame and the RadioButtons. Use a squash widget for that purpose:

```
`HVCenter(
          `HVSquash(
                    `Frame( "Select Software categories",
                          `VBox(
                                ...
                               )
                          )
                    )
         )
```

7.1.3.8. Optical Grouping: Frame

This is not exactly a layout-only widget - you can see it. It is being mentioned here more because like layout widgets it can have children.

Use a Frame to visually group widgets that logically belong together - such as the RadioButtons of a RadioBox or a group of CheckBoxes that have a meaning in common (e.g. individual file permissions, software categories to install, ...).

Note: Do not overuse frames. They have a nice visual effect, but only if used sparingly.

You may need to put a squash widget around the frame in order to avoid excessive empty space between the frame and its inner widgets.

7.1.3.9. Grouping RadioButtons: RadioButtonGroup

The RadioButtonGroup is a widget go logically group individual RadioButton widgets. It does not have a visual effect or an effect on the layout. All it does is to manage the one-out-of-many logic of a RadioBox: When one RadioButton is selected, all the others in the same RadioBox (i.e. in the same RadioButtonGroup) must be unselected.

Please notice that this might not be as trivial as it seems to be at first glance: There might be some outer RadioBox that switches between several general settings, enabling or disabling the others as necessary. Any of those general settings might contain another RadioBox - which of course is independent of the outer one. This is why you really need to specify the RadioButtonGroup.

You usually just surround the VBox containing the RadioButtons with a The RadioButtonGroup.

Don't forget to include your RadioBox within a frame! RadioButtonGroup, Frame and HVSquash usually all come together.

Example 7.5. Grouping RadioButtons

```
`HVCenter(
          `HVSquash(
                    `Frame( "Select Installation Type",
                           `RadioButtonGroup(
                                            `VBox(
                                                  `RadioButton(...),
                                                  `RadioButton(...),
                                                  `RadioButton(...)
                                                 )
                                           )
                          )
                   )
         )
```

7.1.3.10. The Esoterics: ReplacePoint

A ReplacePoint is a "marker" within the widget hierarchy of a layout. You can later refer to it with `ReplaceWidget()`. Use this to cut out a part of the widget hierarchy and paste some other subhierarchy to this point.

The YaST2 wizard dialogs use this a lot: The main window stays the same, just some parts are replaced as needed - usually the large part to the right of the help text, between the title bar and the "previous" and "next" buttons.

A ReplacePoint has no other visual or layout effect.

7.1.3.11. Obsolete: Split

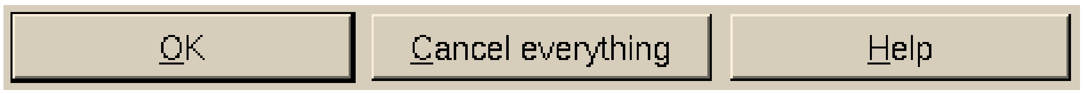
This is not used any more. If you know anything about it, forget it. If you don't, don't bother. It's old and obsolete and nobody used it anyway.

7.1.4. Common Layout Techniques

Use the case studies in this section as building blocks for your own dialogs.

Remember that even though most of the examples use a horizontal layout (a HBox), the same rules and techniques apply in the vertical dimension as well - just replace HBox with VBox, HWeight with VWeight etc.

7.1.4.1. Creating Widgets of Equal Size

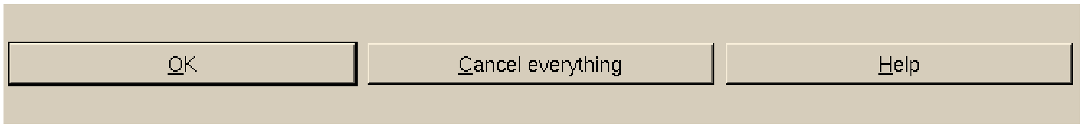


Screen shot of the Layout-Buttons-Equal-Growing.ycp [examples/Layout-Buttons-Equal-Growing.ycp] example

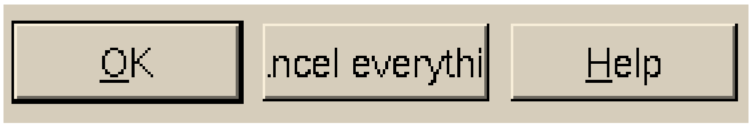
You can easily make several widgets the same size - like in this example. Just specify equal weights for all widgets:

```
`HBox(
      `HWeight(1, `PushButton( "OK"               ) ),
      `HWeight(1, `PushButton( "Cancel everything" ) ),
      `HWeight(1, `PushButton( "Help"             ) )
     )
```

The widgets will grow or shrink when resized. They will always retain equal sizes:

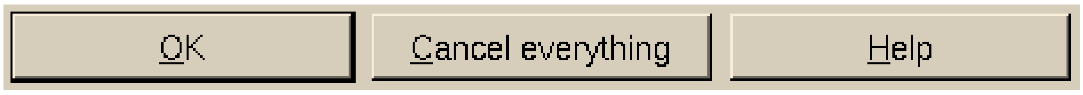


The same example, resized larger.



The same example, resized smaller.

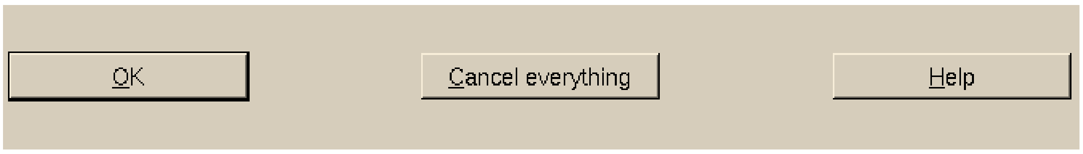
7.1.4.2. Creating Widgets of Equal Size that don't Grow



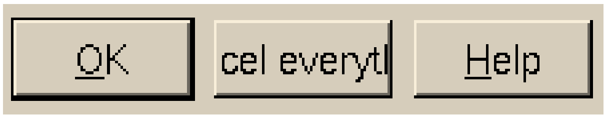
Screen shot of the Layout-Buttons-Equal-Even-Spaced1.ycp [examples/Layout-Buttons-Equal-Even-Spaced1.ycp] example

Widgets with a weight (such as these buttons) are implicitly stretchable. If you don't want the widgets to grow, insert stretches without any weight between them. They will take all excess space - but *only if there is no weight specified* (otherwise, the stretches would always maintain a size according to the specified weight - not what is desired here).

```
`HBox(
      `HWeight(1, `PushButton( "OK"               ) ),
      `HStretch(),
      `HWeight(1, `PushButton( "Cancel everything" ) ),
      `HStretch(),
      `HWeight(1, `PushButton( "Help"             ) )
     )
```

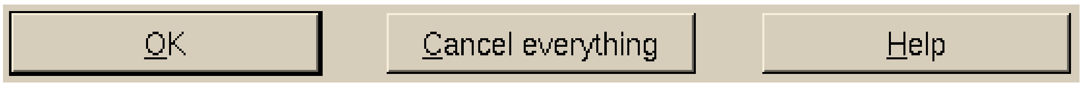


The same example, resized larger. Notice how the stretches take the excess space.



The same example, resized smaller. The stretches don't need any space if there is not enough space anyway.

7.1.4.3. Creating Widgets of Equal Size that don't Grow - with Spacings in between

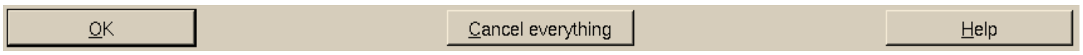


Screen shot of the Layout-Buttons-Equal-Even-Spaced2.ycp [examples/Layout-Buttons-Equal-Even-Spaced2.ycp] example. Notice the spacing between the buttons.

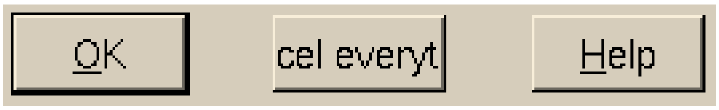
If you want some space between the individual widgets, insert a spacing. You could use both a spacing and a stretch, but specifying the stretchable option for the spacing will do the trick as well - and save some unnecessary widgets:

```
`HBox(
      `HWeight(1, `PushButton( "OK"                ) ),
      `HSpacing(`opt(`hstretch), 3),
      `HWeight(1, `PushButton( "Cancel everything" ) ),
      `HSpacing(`opt(`hstretch), 3),
      `HWeight(1, `PushButton( "Help"              ) )
     )
```

The value "3" used here for the spacing is absolutely random, chosen just for aesthetics. Use your own as appropriate.



The same example, resized larger. Notice how the spacings take the excess space.



The same example, resized smaller.

As you can see, the spacings have one disadvantage here: They need the space you specified even if that means that there is not enough space for the other widgets.

7.1.4.4. Specifying the Size of Scrollable Widgets

As mentioned before, most kinds of widgets that can scroll don't have a natural nice size. If the overall size of your layout is fixed by some other means (e.g. because it is a full screen dialog), you can have it take the remaining space or specify proportions with weights.

If this is not the case, create such "other means" yourself: Surround the scrollable widget with widgets of a well-defined size, e.g. with spacings.

Prevent the spacings from actually using precious screen space themselves by putting a VSpacing in a HBox or a HSpacing in a VBox - it will resize the corresponding layout box in its secondary dimension. It will take no space in its primary dimension.

Example 7.6. Specifying the Size of Scrollable Widgets

```
`VBox(
      `HSpacing(40),            // make the scrollable widget at least 40 units wide
      `HBox(
            `VSpacing(10),      // make the scrollable widget at least 10 units high
            `Table(...)         // or any other scrollable widget
           )
     )
```

See also the Table2.ycp [examples/Table2.ycp], Table3.ycp [examples/Table3.ycp], Table4.ycp [examples/Table4.ycp] and Table5.ycp [examples/Table5.ycp] examples.

As a general rule of thumb, use this technique whenever you place a scrollable widget in a non-defaultsize dialog. Don't leave the size of such widgets to pure coincidence - always explicitly specify their sizes.

7.1.5. Hints and Tips

7.1.5.1. Debugging Aids: The Log File

printf() is your best friend when debugging - every seasoned programmer knows that. YaST2 has something very much like that: *y2log()*, available both in YCP and in the C++ sources. It is being used a lot, and you can add your own in your YCP code. Thus, if something strange happens, check the log file - either in your home directory (`~/.y2log`) or the system wide log file (`/var/log/y2log`).

You can increase the level of verbosity by setting the `Y2DEBUG` environment variable to 1 - both in your shell and at the boot prompt (for debugging during an installation) - boot with something like

```
linux Y2DEBUG=1
```

Log files will be wrapped when they reach a certain size - i.e. the current log file is renamed to `~/.y2log-1`, `~/.y2log-2` etc. or `/var/log/y2log-1`, `/var/log/y2log-2` etc., and a new log file is begun.

7.1.5.2. Keep it Simple - Do not Overcrowd Dialogs!

If the layout engine complains about widgets not getting their *nice size* and tell you to *check the layout*, please do that before you write a bug report. More often than not that just means that your dialog is overcrowded. That doesn't only raise technical problems: In that case your dialog most likely is too complex and not likely to be understood by novice users. In short, you very likely have a problem with your logical design, not with the layout engine. Consider making it easier or splitting it up into several dialogs - e.g. an easy-to-understand novice level base dialog and an advanced "expert" dialog. Use YaST2's partitioning, software selection and LILO configuration dialogs as examples for how to do this.

You might also consider replacing some widgets with others that don't use as much screen space - e.g. use a ComboBox rather than a SelectionBox, or a ComboBox rather than a RadioBox. But always keep in mind that this just reduces screen space usage, not complexity. Plus, widgets like the ComboBox frequently are harder to operate from a user's point of view because they require more mouse clicks or keys presses to get anything done. Use with caution.

7.1.5.3. Always Keep Other UIs in Mind - What does it Look Like with NCurses?

When you created a new dialog or substantially changed an existing one always remember to check it with the other UIs, too. If it looks good with the Qt UI that doesn't mean it looks good with the NCurses UI as well - it might even break completely. There might be too many widgets or parts of widgets may be invisible because of insufficient screen space.

If you don't like that idea always remember some day *you* might be that poor guy who can't run YaST2 with Qt - maybe because of a brand new graphics card the X server doesn't support yet or maybe because you have to install a server system that just has a serial console.

The text based version may not need to look as good (but it would sure be nice if it did), but it needs to work. That means all widgets must be there and be visible. If they are not, you really need to re-arrange or even redesign your dialog. Possibly before somebody from the support department finds it out the hard way - because a user complained badly about it.

7.1.5.4. Do not Neglect Mouseless Users - Always Provide Keyboard Shortcuts!

Very much the same like the previous issue: Consider somebody who wants or needs to operate your dialog without a mouse. Maybe he doesn't have one or maybe it doesn't work - or maybe he uses the NCurses UI. There are even a lot of users who can work a whole lot quicker if they can use keyboard shortcuts for common tasks - e.g. activating buttons or jumping to text input fields. You can and should provide keyboard shortcuts for each of those kinds of widgets.

Of course this needs to be double-checked with each of the translated versions: Keyboard shortcuts not only are language dependent (so users can memorize them), they are even contained within messages files. The translators need to include their own in the respective language, and that means chances are some of the sort cuts are double used - e.g. Alt-K may be used twice in the same dialog, which renders the second use ineffective. Always check that, too.

7.1.6. The Layout Algorithm - How the Layout Engine Works Internally

You don't need to know the internals of the YaST2 UI layout engine in order to be able to create YaST2 dialogs. But this kind of background knowledge certainly helps a lot when you need to debug a layout - i.e. when a dialog you programmed behaves "strange" and doesn't look at all like you expected.

7.1.6.1. Primary and Secondary Dimensions

A HBox lays out its children horizontally, a VBox vertically. How they do that is very much the same except for the dimensions: The HBox uses horizontal as its *primary* dimension, the VBox vertical. The other dimension is called the *secondary* dimension (vertical for the HBox, horizontal for the VBox).

7.1.6.2. Calculating the Nice Size

7.1.6.2.1. Secondary Nice Size

Calculating the nice size in the secondary dimension is easy: It is the maximum of the nice sizes of all children. Thus, for a HBox this is the nice height of the highest child, for a VBox this is the nice width of the widest child.

If any child is a layout box itself (or any other container widget), this process will become recursive for the children of that layout box etc. - this holds true for both the primary and the secondary dimension.

7.1.6.2.2. Primary Nice Size

In the primary dimension things are a bit more complicated: First, the nice sizes of all children without weights are summed up.

Then the sizes of all children with weights are added to that sum - in such a way that each of those gets at least its nice size, yet all weights are maintained with respect to each other. I.e. when a button is supposed to get 30% he must get it, but its label must still be completely visible.

Maybe some of the children with weights need to be resized larger because of those restrictions. Exactly how large is calculated based on the so-called *boss child*. This is the one widget that commands the overall size of all children with weights, the one with

```
max ( nice size / weight )
```

The boss child's nice size and its weight determine the accumulated nice size of all children with weights. The other children with weights will be resized larger to get their share of that accumulated size according to their individual weights.

By the way this is why all children with weights are implicitly stretchable - most of them will be resized larger so the weights can be maintained at all times.

7.1.6.3. Setting the Size of a Layout - SetSize()

Each widget has a SetSize() method. This will be called recursively for all widgets from top (i.e. the outer dialog) to bottom. When a dialog is opened, the UI determines how large a dialog should become. The UI tries to use the dialog's nice size, if possible - unless the defaultsize option is set or the nice size exceeds the screen size, in which case the screen size is used.

After the dialog is opened, the SetSize() method will be called again when:

- The user resizes a dialog.
- A significant portion of the dialog changes - e.g. because of ReplaceWidget().

All of those cases will cause a re-layout of the entire dialog.

For layout boxes, the SetSize() method works like this:

If none of the children of a layout box has a weight, any extra space (i.e. space in excess of the nice size) is evenly distributed among the stretchable children. All non-stretchable children get their nice size, no more.

If there are not any stretchable children, there will be empty space at the end of the layout (i.e. to the right for a HBox and at the bottom of a VBox). If any child has a weight, all children without weights will get no more than their nice sizes - no matter whether or not they are stretchable.

The rest of the space will be distributed among the children with weights according to the individual weights.

There is one exception to that rule, however: If there is more space than the weighted childrens' nice size and there are any stretches or stretchable spacings without weights, the excess space will be evenly distributed among them.

This may sound like a *very* pathological case, but in fact only this gives the application programmer a chance to create equal sized widgets that don't grow, maybe with a little extra space between them. Simple popup dialogs with some buttons are typical examples for this, and this is quite common.

7.1.6.3.1. Running out of Space - the Pathological Cases

There should be enough space for any layout box: By default, the overall size of a dialog is calculated based on its nice size. But this might exceed the full screen size, or the user might manually have resized the dialog (some UIs are capable of that) - both of which cases will cause a dialog to get less than its nice size.

If there is not enough space, the layout engine will complain about that fact in the log file, asking you to "check the layout". Please do that if this message *always* appears when a certain dialog is opened - you may have to rearrange your dialog so all widgets properly fit into it.

Anyway, if it happens, some widgets will get less than their nice size and probably will not look good; some might even be completely invisible.

Even then, as long as there is enough space for all children without weights, those will get their nice sizes. Only the remaining space will be distributed among the children with weights.

If the space isn't even enough for the children without weights, each of them will have to spend some of its space to make up for the loss. The layout engine tries to treat each of them equally bad, i.e. each of them has to give some space.

7.1.6.3.2. Centering in the Secondary Dimension

stretchable

This behaviour may be somewhat unexpected, but not only is this compatible with older versions of the YaST2 UI, it also comes very handy for simple layout tasks like this (taken from the Label1.ycp [examples/Label1.ycp] example):

```
`VBox(
      `Label( "Hello, world" ),
      `PushButton( "OK" )
     )
```

This button will be centered horizontally - without the need for a HCenter around it.

7.2. UI Events

7.2.1. Introduction

7.2.1.1. The YaST2 Event Model

7.2.1.1.1. Classic GUI Event Loops

Classic graphical user interface (GUI) programming is almost always event-driven: The application initializes, creates its dialog(s) and then spends most of its time in one central event loop.

When the user clicks on a button or enters text in an input field, he generates *events*. The underlying GUI toolkit provides mechanisms so the application can react to those events - perform an action upon button click, store the characters the user typed etc.; all this is done from *callback* functions of one kind or the other (whatever they may be called in the respective GUI toolkit).

In any case, it all comes down to one single event loop in the application from where small functions (let's call them *callbacks* for the sake of simplicity) are called when events occur. Those callbacks each contain a small amount of the application's GUI logic to do whatever is to be done when the respective event occurs. The overall application logic is scattered among them all.

This approach is called *event-driven*. Most GUI toolkits have adopted it.

Depending on the primary goal of a GUI application, this event-driven approach may or may not be appropriate. It is perfectly suitable for example for word processor applications, for web browsers or for most other GUI applications that have one central main window the user works with most of his time: The user is the driving force behind those kinds of applications; only he knows what he next wishes to do. The application has no workflow in itself.

Thus the event-driven application model fits perfectly here: The callbacks can easily be self-contained; there is little context information, and there are limited application-wide data.

7.2.1.1.2. The YaST2 Approach

Applications like YaST2 with all its installation and configuration workflows, however, are radically different. The driving force here is the application workflow, the sequence of dialogs the user is presented with.

Of course this can be modeled with a traditional event loop, but doing that considerably adds to the complexity of the application: Either the application needs a lot more callbacks, or the callbacks need to keep track of a lot of status information (workflow step etc.) - or both.

For the YaST2 UI, a different approach was chosen: Rather than having one central event loop and lots of callbacks, the flow control remains in the interpreted YCP code. User input is requested on demand - very much like in simplistic programming languages like the first versions of BASIC.

This of course means that there is no single one central "waiting point" in the program (like the event loop in the event-driven model), but rather lots of such waiting points spread all over the YCP code within each UserInput() or WaitForEvent() statement.

Side note: Of course a graphical UI like the YaST2 Qt UI still has to be prepared to perform screen redraws whenever the underlying window system requires that - i.e. whenever X11 sends an *Expose* (or similar) event. For this purpose the Qt UI is multi-threaded: One thread takes care of X event handling, one thread is the actual YCP UI interpreter. This instant screen redraw is what you lose when you invoke *y2base* with the "--nothreads" command line option.

YCP was meant to be an easy-to-understand programming language for developers who specialize in a particular aspect of system configuration or installation, not in GUI programming.

Practical experience with all the YaST2 modules developed so far has shown that application developers tend to adopt this concept of UserInput() very easily. On the other hand it is a widely known fact that event-driven GUI programming means a steep learning curve because (as mentioned before) it requires splitting up the application logic into tiny pieces for all the callbacks.

Thus, this design decision of YaST2 seems to have proven right much more often than there are problems with its downsides (which of course also exist).

7.2.1.1.3. Simplicity vs. Features

The basic idea of YaST2 UI programming is to create a dialog asking the user for some data and then continue with the next such dialog - meaning that most of those dialogs are basically forms to be filled in with an "OK" (or "Next") and a "Cancel" (or "Back") button. The YCP application is usually interested only in those button presses, not in each individual keystroke the user performs.

This is why by default UserInput() and related functions react to little more than button presses - i.e. they ignore all other events, in particular low-level events the widgets handle all by themselves like keystrokes (this is the input fields' job) or selecting items in selection boxes, tables or similar. Most YCP applications simply don't need or even want to know anything about that.

This makes YCP UI programming pretty simple. The basic principle looks like this:

```
{
    UI::OpenDialog(
            `VBox(
                ... // Some input fields etc.
                `HBox(
                    `PushButton(`id(`back ), "Back" ),
                    `PushButton(`id(`next ), "Next" )
                    )
                )
            );

    symbol button_id = UI::UserInput();

    if ( button_id == `next )
    {
        // Handle "Next" button
    }
    else if ( button_id == `back )
    {
        // Handle "Back" button
    }

    UI::CloseDialog();
}
```

Strictly spoken, you don't even require a loop around that - even though this is very useful and thus strongly advised.

All that can make UserInput() return in this example are the two buttons. Other widgets like input fields (TextEntry), selection boxes etc. by do not do anything that makes UserInput() return - unless explicitly requested.

7.2.1.1.4. The *notify* Option

If a YCP application is interested in events that occur in a widget other than a button, the *notify* widget option can be used when creating it with `UI::OpenDialog()`.

Example 7.7. The notify option

```
UI::OpenDialog(...
  `SelectionBox(`id(`pizza ), `opt(`notify ), ... ),
   ...
  `Table(`id(`toppings), `opt(`notify, `immediate ), ... ),
   ...
)
```

In general, the *notify* options makes UserInput() return when something "important" happens to that widget. The *immediate* option (always in combination with *notify*!) makes the widget even more "verbose".

Note: UserInput() always returns the ID of the widget that caused an event. You cannot tell the difference when many different types of event could have occured. This is why there are different levels of verbosity with *`opt(`notifyÂ)* or *`opt(`notify,Â `immediateÂ)* and the new WaitForEvent() UI builtin function which returns more detailed information. A Table widget for example can generate both Activated and SelectionChanged WidgetEvents.

Exactly what makes UserInput() return for each widget class is described in full detail in the YaST2 event reference.

7.2.1.1.5. Downsides and Discussions

The YaST2 event handling model has been (and will probably always remain) a subject of neverending discussions. Each and every new team member and everybody who casually writes a YaST2 module (to configure the subsystem that is his real responsibility) feels compelled to restart this discussion.

The idea of having a function called *UserInput()* seems to conjure up ghastly memories of horrible times that we hoped to have overcome: The days of home-computer era BASIC programming or university Pascal lectures (remember Pascal's *readln()*?) or even low-tech primitive C programs (*gets()* or *scanf()* are not better, either).

But it's not quite like that. Even though the function name is similar, the concept is radically different: It is not just one single value that is being read, it is a whole dialog full of whatever widgets you see fit to put there. All the widgets take care of themselves; they all handle their values automatically. You just have to ask them (UI::QueryWidget()) for the values when you need them (leave them alone as long as you don't).

The similarity with computing stone age remains, however, in that you have to explicitly call UserInput() or related when you need user input. If you don't, you open your dialog, and a moment later when you continue in your code it closes again - with little chance for the user to enter anything.

Thus, the YaST2 approach has its intrinsic formalisms in that sequence:

```
OpenDialog(...);

UserInput();
QueryWidget(...);
QueryWidget(...);
QueryWidget(...);
...

CloseDialog();
```

This is the price to pay for this level of simplicity.

7.2.1.1.6. Design Alternatives

In the course of those discussions some design alternatives began to emerge:

1. Use the single-event-loop and callback model like most other toolkits.
2. Keep multiple event loops (like UserInput()), but add callbacks to individual widget events when needed so the YCP application can do some more fine-grained control of individual events.
3. Keep multiple event loops, but return more information than this simplistic UserInput() that can return no more than one single ID.

Having just a single event loop would not really solve any problem, but create a lot of new ones: A sequence of *wizard* style dialogs would be really hard to program. Switching back and forth between individual wizard dialogs would have to be moved into some callbacks, and a lot of status data for them all to share (which dialog, widget status etc.) would have to be made global.

What a mess. We certainly don't want that.

All the callback-driven models have one thing in common: Most of the application logic would have to be split up and moved into the callbacks. The sequence of operations would be pretty much invisible to the application developer, thus the logical workflow would be pretty much lost.

Most who discussed that agreed that we don't want that, too.

Add to that the formalisms that would be required for having callbacks: Either add a piece of callback code (at least a function name) to *UI::OpenDialog()* for each widget that should get callbacks or provide a new UI builtin function like, say, *UI::SetCallback()* or *UI::AddCallback()* that gets a YCP map that specifies at least the widget to add the callback to, the event to react to and the code (or at least a function name) to execute and some transparent *client data* where the application can pass arbitrary data to the callback to keep the amount of required global data down.

UI::RemoveCallback()

It might look about like this:

```
define void selectionChanged( any widgetID, map event, any clientData ) {
   ...
   // Handle SelectionChanged event
   ...
};

define void activated( any widgetID, map event, any clientData ) {
   ...
   // Handle Activated event
   ...
};

...
UI::OpenDialog(
   ...
   `Table(`id(`devices ), ... ),
   ...
);
...
UI::AddCallback(`id(`devices ), `SelectionChanged, nil );
UI::AddCallback(`id(`devices ), `Activated, nil );
```

If you think "oh, that doesn't look all too bad", think twice. This example is trivial, yet there are already three separate places that address similar things:

- The callback definitions. Agreed, you'll need some kind of code that actually does the application's business somewhere anyway. But chances are that the callbacks are no more than mere wrappers that call the functions that actually do the application's operations. You don't want to mix up all the back engine code with the UI related stuff.
- Widget creation with *UI::OpenDialog()*
- Adding callbacks with *UI::AddCallback()*

A lot of GUI toolkits do it very much this way - most Xt based toolkits for example (OSF/Motif, Athena widgets, ...). But this used to be a source of constant trouble: Change a few things here and be sure that revenge will come upon you shortly. It simply adds to the overall complexity of something that is already complex enough - way enough.

Bottom line: Having callbacks is not really an improvement.

What remains is to stick to the general model of YaST2 but return more information - of course while remaining compatible with existing YCP code. We don't want (neither can we economically afford to) break all existing YCP code. So the existing UI builtin functions like UserInput() or PollInput() have to remain exactly the same. But of course we can easily add a completely new UI builtin function that does return more information.

This is what we did. This is how WaitForEvent() came into existence. It behaves like UserInput(), but it returns more information about what really happened - in the form of an event *map* rather than just a single ID. That map contains that ID (of course) plus additional data depending on the event that occured.

One charming advantage of just adding another UI builtin is that existing code does not need to be touched at all. Only if you want to take advantage of the additional information returned by WaitForEvent() you need to do anything at all.

So let's all hope with this approach we found a compromise we all can live with. While that probably will not prevent those discussions by new team members, maybe it will calm down the current team members' discussion a bit. ;-)

7.2.1.2. Event Delivery

7.2.1.2.1. Event Queues vs. One Single Pending Event

Since the YaST2 UI doesn't have a single event loop where the program spends most of its time, an indefinite period of time may pass between causing an event (e.g., the user clicks on a widget) and event *delivery* - the time where the (YCP) application actually receives the event and begins processing it. That time gap depends on exactly when the YCP code executes the next UserInput() etc. statement.

This of course means that events that occured in the mean time need to be stored somewhere for the YCP code to pick them up with UserInput() etc.

The first approach that automatically comes to mind is "use a queue and deliver them first-in, first-out". But this brings along its own problems:

Events are only useful in the context of the dialog they belong to. When an event's dialog is closed or when a new dialog is opened on top of that event's dialog (a popup for example) it doesn't make any more sense to handle that event. Even worse, it will usually lead to utter confusion, maybe even damage.

Imagine this situation: The user opens a YaST2 partitioning module just to have a look at his current partitioning scheme.

Side note: This scenario is fictious. The real YaST2 partitioning module is not like that. Any similarities with present or past partitioning modules or present or past YaST2 hackers or users is pure coincidence and not intended. Ah yes, and no animals were harmed in the process of making that scenario. ;-)

- The main dialog with an "OK" button (with, say, ID `ok) opens.
- It takes some time to initialize data in the background.
- The user clicks "OK".
- The background initialization takes some more time.
- The user becomes impatient and clicks "OK" again.
- The background initialization still is not done.
- The user clicks "OK" again.
- The initialization is done. Usually, the YCP code would now reach UserInput() and ueued events would be delivered (remember, this is only a fictious scenario - the UI does not really do that). The first "OK" click from the queue is delivered - i.e. UserInput() returns `ok.

  But this doesn't happen this time: The initialization code found out that something might be wrong with the partitioning or file systems. It might make sense to convert, say, the mounted `/usr` file system from *oldLameFs-3.0* to *newCoolFs-0.95Beta* - which usually works out allright, but of course you never know what disaster lies ahead when doing such things with file systems (and, even worse, with an experimental beta version).
- The initialization code opens a popup dialog with some text to informs the user about that. The user can now click "OK" to do trigger the file system conversion or "Cancel" to keep everything as it is.
- The handler for that popup dialog calls UserInput() - which happily takes the next event from the queue - the `ok button click that doesn't really belong to that dialog, but UserInput() cannot tell that. Neither can the caller. It simply gets `ok as if the user had clicked the "OK" button in the popup.
- The program has to assume the user confirmed the request to convert the file system. The conversion starts.
- The experimental beta code in *newCoolFs-0.95Beta* cannot handle the existing data in that partition as it should. It asks if it is allright to delete all data on that partition. Another popup dialog opens with that question.
- The handler for that confirmation popup takes the next event from the queue which is the third `ok click that should have gone to the main window. But the handler doesn't know that and takes that `ok as the confirmation it asked for.
- `/usr` is completely emptied. Half of the system is gone (along with most of YaST2's files). The disaster is complete - the system is wrecked beyond repair.

Argh. What a mess.

Yes, this example is contrived. But it shows the general problem: Events belong to one specific dialog. It never makes any sense to deliver events to other dialogs.

But this isn't all. Even if the internal UI engine (the *libyui*) could make sure that events are only delivered to the dialog they belong to (maybe with a separate queue for each dialog), events may never blindly be taken from any queue. If the user typed (or clicked) a lot ahead, disaster scenarios similar to the one described above might occur just as well.

Events are context specific. The dialog they belong to is not their only context; they also depend on the application logic (i.e. on YCP code). This is another byproduct of the YaST2 event handling approach.

It has been suggested to use (per-dialog) event queues, but to flush their contents when the dialog context changes:

- When a new dialog is opened (*OpenDialog()*)
- When the current dialog is closed (*CloseDialog()*)
- When parts of the dialog are replaced (*ReplaceWidget()*)
- Upon the YCP application's specific request (new UI builtin *FlushEvents()*)

Exactly when and how this should happen is unclear. Every imaginable way has its downsides or some pathologic scenarios. You just can't do this right. And YCP application developers would have to know when and how this happens - which is clearly nothing they should be troubled with.

This is why *all current YaST2 UIs have onle one single* pending event *and not a queue of events.* When a new event occurs, it usually overwrites any event that may still be pending - i.e. events get lost if there are too many of them (more than the YCP application can and wants to handle).

7.2.1.2.2. Event Reliability

While it may sound critical to have only one single *pending event*, on this works out just as everybody expects:

- When the YCP application is busy and the user clicks wildly around in the dialog, only the last of his clicks is acted upon. This is what all impatient users want anyway: "do this, no, do that, no, do that, no, cancel that all". The "Cancel" is what he will get, not everything in the sequence he clicked.
- The YCP application does not get bogged down by a near-endless sequence of events from the event queues. If things are so sluggish that there are more events than the application can handle in the first place, getting even more to handle will not help any.
- YaST2 dialogs are designed like fill-in forms with a few (not too many) buttons. The input field widgets etc. are self-sufficient; they do their own event handling (so no typed text will get lost). No more than one button click in each dialog makes sense anyway. After that the user has to wait for the next dialog to answer more questions. It does not make any sense to queue events here; the context in the next dialog is different anyway.

As described above, events can and do get lost if there are too many of them. This is not a problem for button clicks (the most common type of event), and it should not be a problem for any other events if the YCP application is written defensively.

7.2.1.2.3. Defensive Programming

Don't take anything for granted. Never rely on any specific event to always occur to make the application work allright.

In particular, never rely on individual SelectionChanged WidgetEvents to keep several widgets in sync with each other. If the user clicks faster than the application can handle, don't simply count those events to find out what to do. Always treat that as a hint to find out what exactly happened: Ask the widgets about their current status. They know best. They are what the user sees on the screen. Don't surprise the user with other values than what he can see on-screen.

In the past, some widgets that accepted initially selected items upon creation had sometimes triggered events for that initial selection, sometimes not. Even though it is a performance optimization goal of the UI to suppress such program-generated events, it cannot be taken for granted if they occur or not. But it's easy not to rely on that. Instead of writing code like this:

```
{
    // Example how NOT to do things

    UI::OpenDialog(
                    ...
                    `SelectionBox(`id(`colors ),
                        [
                            `item(`id("FF0000"), "Red" ),
                            `item(`id("00FF00"), "Blue",true),  // Initially selected
                            `item(`id("0000FF"), "Green" )
                        ]
                    )
    );

    // Intentionally NOT setting the initial color:
    //
    // Selecting an item in the SelectionBox upon creation will trigger a
    // SelectionChanged event right upon entering the event loop.
    // The SelectionChanged handler code will take care of setting the initial color.
    // THIS IS A STUPID IDEA!

    map event = $[];

    repeat
    {
          event = UI::WaitForEvent();

          if ( event["ID"]:nil == `colors )
          {
              if ( event["EventReason"]:nil == "SelectionChanged" )
              {
                  // Handle color change
                  setColor( UI::QueryWidget(`id(`colors ), `SelectedItem ) );
              }
          }
          ...
    } until ( event["ID"]:nil == `close );
}
```

```
{
    // Fixed the broken logic in the example above

    UI::OpenDialog(
                    ...
                    `SelectionBox(`id(`colors ),
                    [
                    `item(`id("FF0000"), "Red" ),
                    `item(`id("00FF00"), "Blue",true),  // Initially selected
                    `item(`id("0000FF"), "Green" )
                    ] ),
                    );

    // Set initial color
    setColor( UI::QueryWidget(`id(`colors ), `SelectedItem ) );

    map event = $[];

    repeat
    {
        event = UI::WaitForEvent();

        if ( event["ID"]:nil == `colors )
        {
            if ( event["EventReason"]:nil == "SelectionChanged" )
            {
                // Handle color change
                setColor( UI::QueryWidget(`id(`colors ), `SelectedItem ) );
            }
        }
        ...
    } until ( event["ID"]:nil == `close );
}
```

It's that easy. This small change can make code reliable or subject to failure on minor outside changes - like a version of the Qt lib that handles things differently and sends another *Selection-Changed* Qt signal that might be mapped to a SelectionChanged WidgetEvents - or does not send that signal any more like previous versions might have done.

Being sceptical and not believing anything, much less taking anything for granted is an attitude that most programmers adopt as they gain more an more programming experience.

Keep it that way. It's a healthy attitude. It helps to avoid a lot of problems in the first place that might become hard-to-find bugs after a while.

7.2.2. Event-related UI Builtin Functions

This section describes only those builtin functions of the YaST2 user interface that are relevant for event handling. The YaST2 UI has many more builtin functions that are not mentioned here. Refer to the UI builtin reference for details.

The Event-related UI Builtin are available in the reference

7.2.3. Event Reference

7.2.3.1. Event Maps in General

Use WaitForEvent() to get full information about a YaST2 UI event. UserInput() only returns a small part of that information, the ID field of the event map.

The event map returned by WaitForEvent() always contains at least the following elements:

Map Key	Value Type	Valid Values	Description
EventType	string	• WidgetEvent • MenuEvent • TimeoutEvent • CancelEvent • KeyEvent • DebugEvent	• The type of this event. • Use this for general event classification.
ID	any		The ID (a widget ID for WidgetEvents) that caused the event. This is what UserInput() returns.
EventSerialNo	integer	>= 0	The serial number of this event. Intended for debugging.

7.2.3.2. Event Types

7.2.3.2.1. WidgetEvent

All WidgetEvents have these map fields in common:

Map Key	Value Type	Valid Values	Description
EventType	string	WidgetEvent	(constant)
EventReason	string	• Activated • ValueChanged • SelectionChanged	The reason for this event. This is something like an event sub-type. Use this to find out what the user really did with the widget.
ID	any		The ID of the widget that caused the event. This is what UserInput() returns.
WidgetID	any		The ID of the widget that caused the event. This is nothing but an alias for "ID", but with this alias you can easily find out if this is a widget event at the same time as you retrieve the widget ID: No other events than WidgetEvent have this field.

Map Key	Value Type	Valid Values	Description
WidgetClass	string	PushButton SelectionBox Table CheckBox ...	The class (type) of the widget that caused the event.
WidgetDebugLabel	string		The label (more general: the widget's *shortcut property*) of the widget that caused the event - in human readable form without any shortcut markers ("&"), maybe abbreviated to a reasonable length. This label is translated to the current locale (the current user's language). This is intended for debugging so you can easily dump something into the log file when you get an event. Wigets that don't have a label don't add this field to the event map, so make sure you use a reasonable default when using a map lookup for this field: Don't use *nil*, use "" (the emtpy string) instead.

7.2.3.2.2. Activated WidgetEvent

Map Key	Value Type	Valid Values	Description
EventReason	string	Activated	(constant)

An *Activated* WidgetEvent is sent when the user explicitly wishes to activate an action.

Traditionally, this means clicking on a PushButton or activating it with some other means like pressing its shortcut key combination, moving the keyboard focus to it and pressing *space*.

Some other widgets (Table, SelectionBox, Tree) can also trigger this kind of event if they have the notify option set.

User interface style hint: YCP applications should use this to do the "typical" operation of that item - like editing an entry if the dialog has an "Edit" button. Use this *Activated* WidgetEvent only as a redundant way (for "power users") of invoking an action. Always keep that "Edit" (or similar) button around for novice users; double-clicks are by no way obvious. The user shouldn't need to experiment how to get things done.

Widget Type	Widget Options	Action to Trigger the Event
PushButton	*(none)*	• Single click on the button (Qt).

Widget Type	Widget Options	Action to Trigger the Event
		• Press *space* on the button. • Press *return* anywhere in the dialog. This activates the dialog's *default button* if it has any and if the respective UI can handle default buttons.
Table	`opt(`notify)	• Double click on an item (Qt). • Press *space* on an item.
SelectionBox	`opt(`notify)	• Double click on an item (Qt). • Press *space* on an item.
Tree	`opt(`notify)	• Double click on an item (Qt). *Note:* This will also open or close items that have children! • Press *space* on an item.

Note that MenuButton and RichText don't ever send WidgetEvents. They send MenuEvents instead.

7.2.3.2.3. ValueChanged WidgetEvent

Map Key	Value Type	Valid Values	Description
EventReason	string	ValueChanged	(constant)

A *ValueChanged* WidgetEvent is sent by most interactive widgets that have a value that can be changed by the user. They all require the notify option to be set to send this event.

Widgets that have the concept of a "selected item" like SelectionBox, Table, or Tree don't send this event - they send a SelectionChanged WidgetEvent instead. One exception to this rule is the MultiSelectionBox which can send both events, depending on what the user did.

Widget Type	Widget Options	Action to Trigger the Event
MultiSelectionBox	`opt(`notify)	Toggle an item's on/off state: • Click on an item's checkbox (Qt). • Press *space* on an item.
CheckBox	`opt(`notify)	Toggle the on/off state: • Single click the widget (Qt). • Press *space* on the widget.
RadioButton	`opt(`notify)	Set this RadioButton to *on*:

Widget Type	Widget Options	Action to Trigger the Event
		• Single click the widget (Qt). • Press *space* on the widget. No event is sent when the button's status is set to *off* because another RadioButton of the same RadioButtonGroup is set to *on* to avoid generating a lot of useless events: Only the *on* case is relevant for most YCP applications.
• TextEntry • MultiLineEdit	`opt(`notify)	Enter text.
ComboBox	`opt(`notify)	• Select another value from the drop-down list: • Open the drop-down list and click on one of its items (Qt). • Open the drop-down list, use the cursor keys to move the selection and press *space* or *return* to actually accept that item. Simply opening the drop-down list and moving the cursor around in it (i.e. changing its selection) does *not* trigger this event. • Enter text (with `opt(`editable)).
IntField	`opt(`notify)	Change the numeric value: • Enter a number. • Click on the *up* button (Qt). • Click on the *down* button (Qt). • Press *cursor up* in the widget (NCurses). • Press *cursor down* in the widget (NCurses).
Slider	`opt(`notify)	• Move the slider. • Enter a number in the embedded IntField. • Use one of the embedded IntField's *up* / *down* button.
PartitionSplitter	`opt(`notify)	• Move the slider. • Enter a number in one of the embedded IntFields.

Widget Type	Widget Options	Action to Trigger the Event
		• Use one of the embedded IntFields' *up* / *down* button.

7.2.3.2.4. SelectionChanged WidgetEvent

Map Key	Value Type	Valid Values	Description
EventReason	string	SelectionChanged	(constant)

A *SelectionChanged* WidgetEvent is sent by most widgets that have the concept of a "selected item" like SelectionBox, Table, or Tree when the selected item changes.

Note that the MultiSelectionBox widget can send a *SelectionChanged* event, but also a ValueChanged WidgetEvent depending on what the user did. This is one reason to keep *Selection-Changed* and ValueChanged two distinct events: Widgets can have both concepts which may be equally important, depending on the YCP application.

The ComboBox never sends a *SelectionChanged* event. It only sends ValueChanged WidgetEvents.

The rationale behind this is that merely opening the drop-down list without actually accepting one of its items is just a temporary operation in a separate pop-up window (the drop-down list) that should not affect the YCP application or other widgets in the same dialog until the user actually accepts a value - upon which event a ValueChanged WidgetEvent is sent.

Widget Type	Widget Options	Action to Trigger the Event
SelectionBox	Qt: `opt(`notify) NCurses: ,`i m me dia `opt(`notifyte)	Select another item: • Click on an item (Qt). • Press *cursor up* in the wid-get. • Press *cursor down* in the widget.
Qt:	`opt(`notify)	
NCurses:	`opt(`notify,`immediate)	
Table	`opt(`notify,`immediate)	Select another item: • Click on an item (Qt). • Press *cursor up* in the wid-get. • Press *cursor down* in the widget.
Tree	`opt(`notify)	Select another item: • Click on an item (Qt). • Press *cursor up* in the wid-get. • Press *cursor down* in the widget.
MultiSelectionBox	`opt(`notify)	Select another item: • Click on an item's text (not on the checkbox) (Qt).

Widget Type	Widget Options	Action to Trigger the Event
		• Press *cursor up* in the widget. • Press *cursor down* in the widget.

7.2.3.2.5. MenuEvent

Map Key	Value Type	Valid Values	Description
EventType	string	MenuEvent	(constant)
ID	any		The ID of the menu item the user selected or the *href* target (as string) for hyperlinks in RichText widgets. *Notice:*This is not the widget ID, it is a menu item or hyperlink ID *inside* that MenuButton or RichText widget!

A MenuEvent is sent when the user activates a menu entry in a MenuButton or a hyperlink in a RichText widget.

Since the ID of the MenuButton or RichText widget is irrelevant in either case, this is not another subclass of WidgetEvent; the ID field has different semantics - and remember, the ID field is the only thing what UserInput() returns so this is particularly important.

For most YCP applications this difference is purely academic. Simply use the ID and treat it like it were just another button's ID.

No notify option is necessary for getting this event. Both MenuButton and RichText deliver MenuEvents right away.

7.2.3.2.6. TimeoutEvent

Map Key	Value Type	Valid Values	Description
EventType	string	TimeoutEvent	(constant)
ID	symbol	`timeout	(constant)

A TimeoutEvent is sent when the timeout specified at WaitForEvent() or TimeoutUserInput() is expired and there is no other event pending (i.e. there is no other user input).

PollInput() never returns a TimeoutEvent; it simply returns *nil* if there is no input.

7.2.3.2.7. CancelEvent

Map Key	Value Type	Valid Values	Description
EventType	string	CancelEvent	(constant)
ID	symbol	`cancel	(constant)

A CancelEvent is an event that is sent when the user performs a general "cancel" action that is usually not part of the YCP application.

For the *Qt UI*, this means he used the window manager close button or a special key combination like Alt-F4 to close the active dialog's window. For the *NCurses UI*, this means he hit the ESC key.

User interface style hint: It is usually a good idea for each dialog to provide some kind of "safe exit" anyway. Most popup dialogs (at least those that have more than just a simple "OK" button) should provide a "Cancel" button. If you use the widget ID `cancel for that button, CancelEvents integrate seamlessly into your YCP application.

"Main window" type dialogs should have an "Abort" button or something similar. If you don't use the widget ID `cancel for that button, don't forget to handle `cancel or "CancelEvent" like that "Abort" button. The user should always have a safe way out of a dialog - preferably one that doesn't change anything. Don't forget to add a confirmation popup before you really exit if there are unsaved data that might get lost!

7.2.3.2.8. KeyEvent

KeyEvents are specific to the NCurses UI. They are not intended for general usage. The idea is to use them where the default keyboard focus handling is insufficient - for example, when the logical layout of a dialog is known and the keyboard focus should be moved to the logically right widget upon pressing the *cursor right* key.

Widgets deliver KeyEvents if they have `` `opt( keyEvent ) `` set. This is independent of the notify option.

It is completely up to the UI what key presses are delivered as key events. Never rely on each and every key press to be delivered.

Map Key	**Value Type**	**Valid Values**	**Description**
EventType	string	KeyEvent	(constant)
ID	string	CursorRight CursorDown F1 a A ...	The key symbol of this event in human readable form. This is what UserInput() returns.
KeySymbol	string	CursorRight CursorDown F1 a A ...	The key symbol of this event in human readable form. This is nothing but an alias for "ID", but with this alias you can easily find out if this is a key event at the same time as you retrieve the key symbol: No other events than KeyEvent have this field.
FocusWidgetID	any		The ID of the widget that currently has the keyboard focus. Unlike a WidgetEvent, this is *not* the same as "ID".
FocusWidgetClass	string	TextEntry SelectionBox ...	The class (type) of the widget that has the key-

Map Key	Value Type	Valid Values	Description
			board focus.
FocusWidgetDebugLabel	string		The label (more general: the widget's *shortcut property*) of the focus widget - in human readable form without any shortcut markers ("&"), maybe abbreviated to a reasonable length. This label is translated to the current locale (the current user's language). This is intended for debugging so you can easily dump something into the log file when you get an event. Wigets that don't have a label don't add this field to the event map, so make sure you use a reasonable default when using a map lookup for this field: Don't use *nil*, use "" (the emtpy string) instead.

Even though at first glance the KeyEvent map looks very much like the WidgetEvent's map, it is different in how the "ID" field is used: A KeyEvent uses it to return the key symbol, while a WidgetEvent returns the widget ID. This is intended to integrate more seamlessly with common usage of UserInput(): A YCP application can simply use UserInput() and check for a return value "CursorRight" etc. - which should not cause any trouble unless somebody uses this as a (badly chosen) widget ID.

7.2.3.2.9. DebugEvent

Map Key	Value Type	Valid Values	Description
EventType	string	DebugEvent	(constant)
ID	symbol	`debugHotkey	(constant)

A DebugEvent is an event type especially intended for debugging YCP code. It is sent when the user presses a special key combination.

For the *Qt UI*, this event is sent upon pressing Alt-Ctrl-Shift-D. There is currently no such key combination in the *NCurses UI*.

Use DebugEvents event to dump additional data to the log file or to open special debugging popup dialogs - but *never* do anything with it that might turn out to be a security hazard. Remember, even though the key combination is really awkward, sooner or later some users will get to know it, and they will experiment.

Chapter 8. Installation Frameworks and Installation Process

8.1. Product Installation Control

8.1.1. Functionality

The product control enables customization of the installation makes it possible to enable and disable features during installation in the final installed product. It controls the workflow and what is really shown to the user during installation.

Beside workflow configuration, other system variables are configurable and can be predefined by the system administrator,to name a fre the software selection, environment settings such as language, time zone, keyboard can be configured and would override default variables provided with shipped products.

The idea of having a pre-defined installation workflow and pre-defined system settings is the middle ground between manual installation and automated installation.

The product configuration file is provided in text on the installation media and defines various settings needed during installation. The following is a list of supported configuration options:

- Workflow

  Replaces the static workflow list with a configurable list using the product configuration file. Entire sections of the workflow can be skipped.

  For example, it will be possible to set the language variable in the configuration file and if the installation language is to be forced for some reason, i.e. IT department wants to force French installations in Quebec, Canada, then the entire dialogue is skipped. If the IT department is to recommend some settings but still give the user the choice to change the default settings, the language dialogue will be shown with Frenchy preselected.

  If non of the above options is used, the default dialogue settings is shown.

- Proposals

  As with the workflow, proposals are also be configurable. For example, certain products would skip some proposals. In the proposal screen the pre-configured settings can be shown with the possibility to change them or with inactive links if the configuration is to be forced.

- System Variables

  Lets the user define system variables like language, keyboard, time zone, window manager, display manager etc. The defined variables will be used as defaults in the respective dialogues.

- Package Selections and additional individual packages

  Define what base package selection and add-on selections should be used for the installation. Additionally provide the possibility to define a list of additional packages. All packages and selections can be selected depending on the architecture using a special architecture attribute in the configuration file.

- Partitioning

  Integrates flexible partitioning into configuration file, instead of the separate file currently used.

- Scripting and Hooks

To customize installation further more, hooks and special slots can be defined where the user can execute scripts. For example, scripts can be executed at the very beginning of the installation (After processing the configuration file), in the installation system before initial boot, in the chroot-ed environment and after initial boot and before/after every step in the workflow. Scripting languages supported during installation are currently Shell, Perl.

8.1.2. Implementation

The control file is implemented in simple structured XML syntax which so far has been used for automated installation . The XML structure used can be mapped easily to YaST data structures and all data types available in YaST are supported for easy data access and manipulation.

The primary use of the control file is to configure the workflow of the installation and it offers the possibility to predefine a certain setup, but it also defines product installation features and other product related variables.

Note

Note that the control file is not an optional tool to help customize installation, it is required during installation and without the file, installation may fail or lead to unexpected results. YaST provides a default and general control file which is always available in the system. The general and product independent control files is installed by the package *yast2-installation* in `/usr/share/YaST2/control/control.xml`.

During installation, *linuxrc* searches for the a file named `control.xml` on the installation medium (CD, NFS, FTP..) and copies the file into the installation system and makes the file available to YaST. YaST then starts and looks for the control file in 3 location before it starts with the installation workflow:

- `/control.xml`

  Usually the file is in top directory after it has been copied by linuxrc and during initial installation phase.

- `/var/lib/YaST2/control.xml`

  After reading the file, and before second installation phase, the control file is copies from the top directory to `/var/lib/YaST2/control.xml`

- `/usr/share/YaST2/control/control.xml`

  This is the location where *yast2-installation* installs the file in all products. The file is the same on all products.

One of the main reasons for using the control is to provide non YaST developers to change the installation behavior and customize various settings without the need to change YaST packages.

The control file for the various product shall therefore be maintained out of the YaST development trees and in separate SUSE internal and product specific packages.

8.1.3. Configuration

8.1.3.1. workflows

Using the control file, multiple workflows can be defined for different modes and installation stages. Thus, the element *workflows* in the control file evaluates to a list of workflows.

Beside defining what YaST clients should be executed during installation, the workflow configuration also let you specify the wizard steps and how they should appear during graphical installation.

A workflow list element is a map with the following elements:

- label

  The label of the workflow as it appears on the left side of the wizard. For example *Base Installation*

- defaults

  The default arguments to the clients. This is a map element.

- stage

  This options defines the stage or phase of installation.. Possible values are *initial* for the initial stage and *continue* for the workflow of the installation after reboot

- mode

  Defines installation mode. Several modes are available, most important modes are:

  - installation
  - update
  - autoinst

- modules

  This is the actual workflow and is a list of elements describing the order in which the installation should proceed.

  A module element is a map with the following configuration options:

  - name: The name of the module. All installation clients and modules have a unified prefix (inst_) which can be ommited here. For example, if the YaST file for the module is called *inst_test*, then the name in the control file is *test*

  - label: The label of the module in the step dialog. This is an optional element. If it is not set, the label of the previous module is used.

  - arguments: The arguments for the module is a comma separated list which can accept booleans and symbols.

The following listing shows a typical installation workflow:

```
                <workflows config:type="list">
        <workflow>
            <!-- 'label' is what the user will see -->
            <label>Base Installation</label>
            <!-- default settings for all modules -->
            <defaults>
                <!-- arguments for the clients -->
                <arguments>false,false</arguments>
                <!-- allowed architectures "all", "i386", "i386,ia64,x86_64"  -->
                <archs>all</archs>
            </defaults>
            <stage>initial</stage>
            <mode>installation,update</mode>
            <modules  config:type="list">
                <module>
                    <name>info</name>
                    <arguments>false,true</arguments>
                </module>
                <module>
                    <name>proposal</name>
                    <arguments>true,true,`ini</arguments>
                    <label>Installation Settings</label>
                </module>
                <module>
                    <name>do_resize</name>
                    <update config:type="boolean">false</update>
                    <archs>i386,x86_64,ia64</archs>
                    <label>Perform Installation</label>
                </module>
                <module>
                    <name>prepdisk</name>
                    <!-- Multiple modules with the same 'label' will be
                         collapsed to one single user-visible step.
                         The step is considered finished when the last module
                         with the same 'label' is finished.  -->
                    <label>Perform Installation</label>
                </module>
                <module>
                    <name>kickoff</name>
                    <label>Perform Installation</label>
                </module>
                <module>
                    <name>rpmcopy</name>
                    <label>Perform Installation</label>
                </module>
                <module>
                    <name>finish</name>
                    <label>Perform Installation</label>
                </module>
            </modules>
        </workflow>
```

8.1.3.2. Proposals

Part of the installation workflows are proposal screens, which consists of group of related configuration settings. For example *Network*, *Hardware* and the initial *Installation* proposal.

If you with for some reason to add or modify a proposal, which is discourged because of configuration dependencies, then this would be possible using the control file.

```
        <proposal>
            <type>network</type>
            <stage>continue,normal</stage>
            <proposal_modules config:type="list">
                <proposal_module>lan</proposal_module>
                <proposal_module>dsl</proposal_module>
                <proposal_module>isdn</proposal_module>
                <proposal_module>modem</proposal_module>
                <proposal_module>proxy</proposal_module>
        <proposal_module>remote</proposal_module>
            </proposal_modules>
        </proposal>
```

The proposal in the above listing is displayed in the so called *continue* mode which is the second phase of the installation. The proposal consists of different configuration options which are controled using a special API.

Currently, proposals names and captions as fixed and cant be changed. It is not possible to create a special proposal screen, instead those available should be used: *network*, *hardware*, *service*.

In the workflow, the proposals are called as any workflow step with an additional argument identifying the proposal screen to be started. (*`net* for network, *`hw* for hardware and *`service* for service proposals. The following examples shows how the network proposal is called as a workflow step:

```
<module>
    <label>Network</label>
    <name>proposal</name>
    <arguments>true,true,`net</arguments>
</module>
```

8.1.3.3. Installation and Product Variables

It is possible to define some installation variables (language, timezone, keyboard,..) and force them in the proposal. User will still be able to change them however.

The following variables can be set:

- Timzeone
- Language
- Keyboard
- Auto Login (not recommended for multi-user environments and server installations)
- IO Scheduler

  Default is *as*.
- Desktop Scheduler

the following example shows all options above

```
<globals>
    <enable_autologin config:type="boolean">true</enable_autologin>
    <language>de_DE</language>
    <timezone>Canada/Eastern</timezone>
    <use_desktop_scheduler config:type="boolean">true</use_desktop_scheduler>
    <io_scheduler>as</io_scheduler>
</globals>
```

8.1.3.4. Software

Using this section in the control file you can change the software proposal during installation.

By default, the software proposal is generated depending on the available space in the system. If enough space is available, a fairly large selection of packages and add-on package groups is automatically selected. This behavior is controlled by the *selection_type* element in the control file.

The default value for the above element is *auto*. To force a selection which can not be changed by the user, the value *fixed* has to be used.

If a fixed software selection is desired, then the fixed selection name has to be specifed in the control file. This selection must be a base selection. To specify the name of the base selectiont to be forced, use the *base_selection* element.

The list of base selections can be found on the first CD of the product in the directory `suse/setup/descr` or by using the following command (The example shows the list of selections for *SLES 9*):

To switch from *desktop* based proposals to the normal behavior of software proposals in YaST , the *software_proposal* element has been introduced. Setting the value to *desktop* will make YaST show a dialog with Desktops to select from, i.e. KDE or *Gnome*. If this element is not defined, default behavior is assumed.

```
for i in `grep -l baseconf SUSE-SLES-Version-9/CD1/suse/setup/descr/* `; do
  basename $i .sel;
done
```

The above will have the following output:

```
Full-Installation
Minimal+X11
Minimal
default
```

which matches the base selections in *SLES 9*.

Additionally, you can configure how updating of packages should be performed. The following options are available:

- delete_old_packages

  Do not delete old RPMs when updating.

- only_update_selected

  One can update (only update packages already installed) or upgrade (also install new packages with new functionality). For example, SLES should do "update", not "upgrade" by default

8.1.3.5. Partitioning

If present, the partition proposal will be based on the data provided in the control file.

8.1.3.5.1. Algorithm for space allocation

Space allocation on a disk happens in the following order. First all partition get the size allocated that is determined by the size parameter of the partition description. If a disk cannot hold the sum of these sizes this disk is not considered for installation. If all demands by the size parameter are fulfilled and there is still space available on the disk, the partitions which have a parameter "percent" specified are increased until the size demanded by by "percent" is fulfilled. If there is still available space on the disk (this normally only can happen if the sum of all percent values are below 100), all partitions that are specified with a size of zero are enlarged as far as possible. If a "maxsize" is specified for a partition, all enlargement are only done up to the specified maxsize.

If more than one of the available disks is eligible to hold a certain partition set, the disk is selected as follows. If there is a partition allocated on that disk that has its size specified by keywords "percent" or by "size=0" and does not have a "maxsize" value set then the desired size for this partition is considered to be unlimited. If a partition group contains a partition which an unlimited desired size, the disk that maximizes the partition size for the unlimited partitions is selected. If all partitions in a partition group are limited in size then the smallest disk that can hold the desired sizes of all partitions is selected for that partition group.

If there are multiple partition groups the the partition group with the lowest number (means highest priority) get assigned its disk first. Afterward the partition group with the next priority gets assigned a the optimal disk from the so far unassigned disks.

8.1.3.5.2. Configuration Options

The following elements are global to all disks and partitions:

prefer_remove

Possible values	true|false

Default value	true
Description	If set to false the partition suggestion tries to use gaps on the disks or to re-use existing partitions. If set to true then the partition suggestion prefers removal of existing partitions.

remove_special_partitions

Possible values	true|false
Default value	false
Description	If set to false YaST2 will not remove some special partitions (e.g. 0x12 Compaq diagnostics, 0xde Dell Utility) if they exists on the disk even if prefer_remove is set to true. If set to true YaST2 will remove even those special partitions.

Caution

Caution: Since some machines are not even bootable any more when these partitions are removed one should really know what he does when setting this to true

keep_partition_fsys

Possible values	comma separated list of reiser, xfs, fat, vfat, ext2, ext3, jfs, ntfs, swap
Default value	Empty list
Description	Partitions that contain filesystems in that list are not deleted even if prefer_remove is set to true.

keep_partition_id

Possible values	comma separated list of possible partition ids
Default value	Empty list
Description	Partitions that have a partition id that is contained in the list are not deleted even if prefer_remove is set to true.

keep_partition_num

Possible values	comma separated list of possible partition numbers
Default value	Empty list
Description	Partitions that have a partition number that is contained in the list are not deleted even if prefer_remove is set to true.

To configure individual partitions and disks, a list element is used with its items describing how should the partitions be created and configured

The attributes of such a partition are determined by several elements. These elements are described in more detail later.

General remarks to all option values

If there is a blank or a equal sign (=) contained in an option value, the values has to be surrounded by double quotes ("). Values that describe sizes can be followed by the letters K, M, G. (K means Kilobytes, M Megabytes and G Gigabytes).

mount

Example	<mount>swap</mount>
Description	This entry describes the mount point of the partition. For a swap partition the special value "swap" has to be used.

fsys

Example	<fsys>reiser</fsys>
Description	This entry describes the filesystem type created on this partition. Possible Filesystem types are: reiser, ext2, ext3, xfs, vfat, jfs, swap If no filesystem type is given for a partition, reiserfs is used.

formatopt

Example	<formatopt>reiser<formatopt>
Description	This entry describes the options given to the format command. Multiple options have to be separated by blanks. There must not be a blank between option letter and option value. This entry is optional.

fstopt

Example	<fstopt>acl,user_xattr<fstopt>
Description	This entry describes the options written to `/etc/fstab`. Multiple options have to be separated by comma. This entry is optional.

label

Example	<label>emil<label>
Description	If the filesystem can have a label, the value of the label is set to this value.

id

Example	<id>0x8E<id>
Description	This keyword makes it possible to create partitions with partition ide other than 0x83 (for normal filesystem partitions) or 0x82 (for swap partitions). This make it possible to create LVM or MD partitions on a disk.

size

Example	<size>2G<size>
Description	This keyword determines the size that is at least needed for a partition. A size value of zero means that YaST2 should try to make the partition as large as possible after all other demands regarding partition size are fulfilled. The special value of "auto" can be given for the `/boot` and swap partition. If auto is set for a /boot or swap partition YaST2 computes a suitable partition size by itself.

percent

Example	<percent>30<percent>
Description	This keyword determines that a partition should be allocated a certain percentage of the available space for installation on a disk.

maxsize

Example	<maxsize>4G<maxsize>
Description	This keyword limits the maximal amount of space that is allocated to a certain partition. This keyword is only useful in conjunction with a size specification by keyword "percent" or by an entry of "size=0".

increasable

Example	<increasable config:type="boolean">true<increasable>
Default	false
Description	After determining the optimal disk usage the partition may be increased if there is unallocated space in the same gap available. If this keyword is set, the partition may grow larger than specified by the maxsize and percent parameter. This keyword is intended to avoid having unallocated space on a disk after partitioning if possible.

disk

Example	<disk>2<disk>
Description	This keyword specifies which partitions should be placed on which disks if multiple disks are present in the system. All partitions with the same disk value will be placed on the same disk. The value after the keyword determines the priority of the partition group. Lower numbers mean higher priority. If there are not enough disks in the system a partition group with lower priority is assigned a separate disks before a partition group with higher priority. A partition without disk keyword is implicitly assigned the highest priority 0.

Example 8.1. Flexible Partitioning

If in the example below the machine has three disks then each of the partition groups gets on a separate disk. So one disk will hold `/var`, another disk will hold /home and another disk will hold `/`, `/usr` and `/opt`. If in the above example the machine has only two disks then `/home` will still be on a separate disk (since it has lower priority than the other partition groups) and `/`, `/usr`, `/opt` and `/var` will share the other disk.

If there is only one disk in the system of course all partitions will be on that disk.

```
        <partitions config:type="list">
            <partition>
                <disk config:type="integer">3</disk>
                <mount>/var</mount>
                <percent config:type="integer">100</percent>
            </partition>
            <partition>
                <disk config:type="integer">2</disk>
                <mount>/</mount>
                <size>1G</size>
            </partition>
            <partition>
                <disk config:type="integer">2</disk>
                <mount>/usr</mount>
                <size>2G</size>
            </partition>
            <partition>
                <disk config:type="integer">2</disk>
                <mount>/opt</mount>
                <size>2G</size>
            </partition>
           <partition>
                <disk config:type="integer">1</disk>
                <mount>/home</mount>
                <percent config:type="integer">100</percent>
            </partition>
        </partitions>
```

A more complete example with other options is shown below:

```
    <partitioning>
        <partitions config:type="list">
            <partition>
                <disk config:type="integer">2</disk>
                <mount>swap</mount>
                <size>auto</size>
            </partition>
            <partition>
                <disk config:type="integer">1</disk>
                <fstopt>defaults</fstopt>
                <fsys>reiser</fsys>
                <increasable config:type="boolean">true</increasable>
                <mount>/</mount>
                <size>2gb</size>
            </partition>
            <partition>
                <disk config:type="integer">2</disk>
                <fstopt>defaults,data=writeback,noatime</fstopt>
                <fsys>reiser</fsys>
                <increasable config:type="boolean">true</increasable>
                <mount>/var</mount>
                <percent config:type="integer">100</percent>
                <size>2gb</size>
            </partition>
        </partitions>
        <prefer_remove config:type="boolean">true</prefer_remove>
        <remove_special_partitions config:type="boolean">false</remove_special_partitions>
    </partitioning>
```

8.1.3.6. Hooks

It is possible to add hooks before and after any workflow step for further customization of the installed system and to to perform non-standard tasks during installation.

Two additional elements define custom script hooks:

- prescript: Executed before the module is called.
- postscript: Executed after the module is called.

Both script types accept two elements, the interpreter used (shell or perl) and the source of the scripts which is embedded in the XML file using CDATA sections to avoid confusion with the XML syntax. The following example shows how scripts can be embedded in the control file:

```
                <module>
                    <name>info</name>
                    <arguments>false,true</arguments>
                    <prescript>
                        <interpreter>shell</interpreter>
                        <source>
<![CDATA[#!/bin/sh
touch /tmp/anas
echo anas > /tmp/anas
]]>
                        </source>
                    </prescript>
                </module>
```

8.2. Firstboot Configuration

The YaST firstboot utility (YaST Initial System Configuration), which runs after the installation is completed, lets you configure the Novell Linux Desktop system before creation of the install image so that on the first boot after configuration, users are guided through a series of steps that allow for easier configuration of their desktops. YaST firstboot does not run by default and has to be configured to run by the user or the system administrator. It is useful for image deployments where the system in the image is completely configured. However, some final steps such as root password and user logins have to be created to personalize the system.

The default workflow for the interface is as follows:

1. The Welcome screen
2. The License Agreement
3. Date & Time
4. Network
5. Root Password
6. User Account
7. Hardware
8. Finish

During firstboot, two additional dialogs are shown for writing the data and running SuSEconfig which require no user interaction.

8.2.1. Enabling Firstboot

Firstboot is disabled by default. The yast2-firstboot package is not part of any software selection and has to be installed individually. During the Installation, click Software in the Installation Settings screen, then select the yast2-firstboot package in the Rest selection list.

1. Install the product on a master box, making sure that you install the firstboot package.
2. Create the empty file `/etc/reconfig_system`. This file will be deleted when firstboot configuration is completed. This can be done by issuing the commnd: **touch /etc/reconfig_system**
3. Enable the firstboot service using the YaST runlevel editor, or directly on the command line using the following command: **chkconfig firstboot on**

When the system comes up after a shutdown, the firstboot configuration utility will be started and the user will be presented with the configuration screens.

There are different ways the firstboot configuration utility can be used, one of them for creating installation images. The following step by step description shows how an image can be created after firstboot has been enabled.

1. Boot the master box using the rescue boot option.
2. Configure network in the rescue system.
3. Mount an NFS exported directory to /mnt.
4. Run **dd if=/dev/hda of=/mnt/image.bin count=4000000** to store the master box's hard disk image onto the NFS server.

And to install the image you have just created:

1. Boot a user's machine using the rescue boot option.
2. Configure network in the rescue system.
3. Mount the NFS exported directory to /mnt.
4. Run **dd if=/mnt/image.bin of=/dev/hda count=4000000.**
5. Remove the boot media and boot the user's machine.
6. After firstboot comes up, configure the date and time, root password, and user account and any other additional settings.

The Post Configuration Utility (firstboot) expects the X server to be configured. If no X server is configured, it will automatically start in text mode.

8.2.2. Customizing YaST Firstboot

8.2.2.1. Customizing Messages

The utility has standard and translated texts in the default setup. If you want to change those texts, use the firstboot configuration file,`/etc/sysconfig/firstboot`.

This file lets you change the text of the following dialogs:

- Welcome screen
- License Agreement
- Finish dialog

To do this, change the values of the respective variables (FIRSTBOOT_WELCOME_FILE,FIRSTBOOT_LICENSE_FILE , and FIRSTBOOT_FINISH_FILE) to the full path of a plain or rich text formatted text file. This will give you dialogs with customized text. If the references files are in plain text, they will be shown as such automatically. If they contain any markup language, they will be formatted as rich text.

The default license text shown is taken from the file `/var/lib/YaST2/info.txt` which is the EULA of the product being installed.

8.2.2.2. License Action

The variable LICENSE_REFUSAL_ACTION sets the action to be executed if the user does not accept the license. The following options are available:

- halt: system is halted (shut down)
- continue: continue with configuration
- abort: Abort firstboot configuration utility and continue with the boot process.

8.2.2.3. Release Notes

Use the configuration option FIRSTBOOT_RELEASE_NOTES_PATH to show release notes in the target language. The value of the option should be a path to a directory with files using language dependent naming (`RELEASE-NOTES.{language}.rtf`). For english, the following file will be needed: RELEASE-NOTES.{language}.rtf.

The original release notes for the installed product availabe in `/usr/share/doc/release-notes` can be used as an example.

8.2.2.4. Customizing Workflow Components

The default firstboot workflow can be controled using one single file which is a subset of the control.xml file used to control the complete installation. The firstboot control file consists of workflow and proposal configurations and can be used to add or remove configuration screens depending on the end configuration of the system. The file firstboot.xml is installed with the yast2-firstboot package and can be found at the following location: `/usr/share/YaST2/control/firstboot.xml`.

This file can be modified to match the post installation requirements of the product in question. In addition to the default and pre-installed components, custom screens can be added to enable maximal flexiblity during post installation. For more information about the syntax of the control file, see the document titled “Product Installation Control”.

8.2.3. Scripting

You can add scripts to be executed at the end of the firstboot configuration to customize the system depending on user input or the environment of the machine. Scripts should be placed in `/`

`usr/share/firstboot/scripts` or in a custom location that can be set using the `/etc/sysconfig/firstboot` configuration file.

8.3. API for YaST2 installation proposal

8.3.1. Motivation

After five releases, YaST2 is now smart enough to make reasonable proposals for (near) every installation setting, thus it is no longer necessary to ask the user that many questions during installation: Most users simply hit the [next] button anyway.

Hence, YaST2 now collects all the individual proposals from its submodules and presents them for confirmation right away. The user can change each individual setting, but he is no longer required to go through all the steps just to change some simple things. The only that (currently) really has to be queried is the installation language - this cannot reasonably be guessed (yet?).

The new YaST2 installation includes the following steps:

- (Minimal) hardware probing - no user interaction required
- Language selection - user picks installation language
- Installation proposal - very much like the old installation summary just before the real installation started, only this time the user CAN change settings by clicking into the summary (or via a separate menu as a fallback).
- Create / format partitions according to proposal / user selection - no user interaction required
- Install software packages from CD / DVD / other installation media

After this, all that is remained left is basic system configuration like:

- X11
- Network interface(s)
- Network services
- Additional hardware (printer, sound card, scanner, ...)

8.3.2. Overview

YaST2 installation modules should cooperate with the main program in a consistent API. General usage:

- inst_proposal (main program) creates empty dialog with RichText widget
- inst_proposal calls each sub-module in turn to make proposal
- user may choose to change individual settings (i.e., clicks on a hyperlink)
- inst_proposal starts that module's sub-workflow which runs independently. After this, inst_proposal tells all subsequent (all?) modules to check their states and return whether a change of their proposal is necessary after the user interaction.
- main program calls each sub-module to write the settings to the system

8.3.3. The Dispatcher Interface

Each submodule provides a function dispatcher that can be called with 'CallFunction()'. The function to be called is passed as a parameter to this dispatcher. Parameters to the function are passed as another parameter in a map. The result of each call is a map, the contents of which depend on the function called.

The reason for this additional overhead is to provide extensibility and reusability for the installation workflow: A list of submodules to be called is read from file. This requires, however, that no explicit 'mod::func()' calls are used in 'inst_proposal.ycp'. Rather, the list contains the name of the submodule. Since each submodule is required to provide an identical API, this is sufficient.

Example 8.2. Proposal Example

Proposal is to call

- input_devices (keyboard, mouse)
- partitioning
- software_selection
- boot_loader
- timezone

inst_proposal calls

```
        map result = CallFunction (input_devices( "MakeProposal", $[ "force_reset"     : false,
        "language_changed": false ] ) );
        map result = CallFunction (partitioning ( "MakeProposal", $[ "force_reset"   : false,
        "language_changed": false ] ) );
        ...
```

If the user clicks on the hyperlink on "input_devices" in the proposal display, inst_proposal calls:

```
        map result = CallFunction (input_devices( "AskUser", $[ "has_next": true ] ) );
```

8.3.4. API functions

Note

If any parameter is marked as "optional", it should only be specified if it contains a meaningful value. Don't add it with a 'nil' value.

The dispatcher provides the following functions:

- MakeProposal
- AskUser
- Description
- Write

8.3.5. Dummy Proposal

```
     /**
 * Module:          proposal_dummy.ycp
 *
 * $Id: dummy_proposal.ycp,v 1.1 2004/02/27 02:37:39 nashif Exp $
 *
 * Author:          Stefan Hundhammer <sh@suse.de>
 *
 * Purpose:          Proposal function dispatcher - dummy version.
 *                  Use this as a template for other proposal dispatchers.
 *                  Don't forget to replace all fixed values with real values!
 *
 *                  See also file proposal-API.txt for details.
 */
{
    textdomain "installation";

    string func  = (string) WFM::Args(0);
    map    param = (map) WFM::Args(1);
    map    ret   = $[];

    if ( func == "MakeProposal" )
    {
        boolean force_reset      = param["force_reset"     ]:false;
        boolean language_changed = param["language_changed"]:false;

        // call some function that makes a proposal here:
        //
        // DummyMod::MakeProposal( force_reset );

        // Fill return map

        ret =
            $[
              "raw_proposal" :  [
                                 "proposal item #1",
                                 "proposal item #2",
                                 "proposal item #3"
                                ],
              "warning"         : "This is just a dummy proposal!",
              "warning_level" : `blocker
            ];
    }
    else if ( func == "AskUser" )
    {
        boolean has_next = param["has_next"]:false;

        // call some function that displays a user dialog
        // or a sequence of dialogs here:
        //
        // sequence = DummyMod::AskUser( has_next );


        // Fill return map

        ret =
            $[
              "workflow_sequence": `next
            ];
    }
    else if ( func == "Description" )
    {
        // Fill return map.
        //
        // Static values do just nicely here, no need to call a function.

        ret =
            $[
              // this is a heading
              "rich_text_title" : _( "Dummy"  ),
              // this is a menu entry
              "menu_title" : _( "&Dummy" ),
              "id"         : "dummy_stuff"
            ];
    }
    else if ( func == "Write" )
    {
        // Fill return map.
        //

        ret =
            $[
              "success" : true
            ];
    }

    return ret;
}
```

Chapter 9. YaST Development And Tools

9.1. YaST2 Development Tools

This document is a user's guide to the YaST2 *devtools* (short for "development tools"), a utility collection to make developing YaST2 code easier - C++ as well as YCP.

9.1.1. Quick Start

- Install the `yast2-devtools` RPM or check out the `devtools` module from the YaST2 CVS and build and install it:

```
cd yast2                # your YaST2 CVS working directory
cvs co devtools
cd devtools
make -f Makefile.cvs
make
sudo make install
```

- See the Migration how to change your C++ or YCP module.
- Build and use your package as before.
- If `make package` complains, fix the complaints. For a tempoarary package to check whether or not `build` works with your changes, use "`make package-local`" - but *never check in a package to `/work/src/done` that you created this way!*

9.1.2. What is it?

The YaST2 devtools are an add-on to the classic automake / autoconf environment YaST2 used to use.

Since the toplevel `Makefile.am` is pretty much the same throughout all YaST2 C++ or YCP modules yet contains more and more specialized `make` targets, this toplevel `Makefile.am` is now automatically generated.

The only thing that is (or, rather, "should be") different in all those toplevel `Makefile.am` files is the "`SUBDIRS =`" line. This line is moved to a `SUBDIRS` in the package's toplevel directory, much like `RPMNAME`, `VERSION`, `MAINTAINER` etc. - the rest of `Makefile.am` is copied from a common path `/usr/share/YaST2/data/devtools/admin`. Thus, changes that should affect all of YaST2's toplevel `Makefile.am` files are much easier to do and all YaST2 modules can benefit from them without the need to change (i.e. `cvs up`, edit, `cvs ci`) all of over 85 individual files.

This implies, of course, that the toplevel `Makefile.am` is no longer stored in the CVS repository since it is now automatically generated.

On the downside, this of course implies that the files and scripts required for this new automagic are available at build time - i.e. on each YaST2 developer's development machine as well as in the *build* environment. Thus, you will need to either install the appropriate RPM or build the devtools manually - see the Quick Start section for details.

9.1.3. Migration

If you haven't done that yet, install the devtools - see the Quick Start section for details.

If you are not sure, check `/usr/share/YaST2/data/devtools` - if you don't have that directory, the devtools are not installed.

You can simply use the `devtools-migration` script that comes with the devtools package:

```
cd yast2/modules/mypackage
y2tool devtools-migration
cvs ci
```

This script performs the following steps:

- Go to your package's toplevel directory:

  ```
  cd yast2/modules/mypackage
  ```

- Create a `SUBDIRS` file from your existing `Makefile.am`.

  Note

  You can do without that `SUBDIRS` file if you want to include all subdirectories that have a `Makefile.am` in alphabetical order anyway):

  ```
  grep 'SUBDIRS' Makefile.am | sed -e 's/SUBDIRS *= *//' >SUBDIRS
  ```

  Getting rid of the "`SUBDIRS = `" prefix is not exactly mandatory (the devtools are forgiving enough to handle that), but recommended.

- Add that new `SUBDIRS` file to the CVS repository:

  ```
  cvs add SUBDIRS
  ```

- Get rid of the old `Makefile.am` both locally and in the CVS repository - this file will be automatically generated from now on:

  ```
  cvs rm -f Makefile.am
  ```

- Get rid of the old copyright notices (`COPYING`, `COPYRIGHT.{english,german,french}`) both locally and in the CVS repository:

  ```
  cvs rm -f COPYING COPYRIGHT.{english,french,german}
  ```

  Those files will automatically be added to the tarball upon `make package`, `make dist` and related commands.

- Add `Makefile.am` to the `.cvsignore` file since it will be automatically generated from now on (otherwise "cvs up" will keep complaining about it):

  ```
      echo "Makefile.am" >>.cvsignore
  ```

- Edit your `.spec.in` file. Locate the `neededforbuild` line and add `yast2-devtools` to it:

  ```
  vi *.spec.in
  ...
  (locate "neededforbuild")
  (add "yast2-devtools")
  (save + quit)
  ```

  OK, that was the wimp version. Here is the freak version:

  ```
  perl -p -i -e 's/neededforbuild/neededforbuild yast2-devtools/' *.spec.in
  ```

- Add the line that creates the toplevel `Makefile.am` to your `Makefile.cvs`:

  ```
  vi Makefile.cvs
  (locate "aclocal")
  (add a new line above this:)
  [tab] y2tool y2automake
  (save + quit)
  ```

  Again, a freak version for this:

  ```
  perl -p -i -e 'print "\ty2tool y2automake\n" if /aclocal/' Makefile.cvs
  ```

  The new `Makefile.cvs` should look like this:

  ```
  all:
          y2tool y2automake
          autoreconf --force --install
  ```

- Double-check what you just did and check it into the CVS when everything looks OK. "`cvs up`" should print something like this:

  ```
  M .cvsignore
  R COPYING
  R COPYRIGHT.english
  R COPYRIGHT.french
  R COPYRIGHT.german
  R Makefile.am
  M Makefile.cvs
  A SUBDIRS
  M myproject.spec.in
  ```

  "`cvs diff`"should print something like this:

  ```
  Index: .cvsignore
  ...
   config.log
   aclocal.m4
  +Makefile.am
  ...
  cvs server: Makefile.am was removed, no comparison available
  ...
  Index: Makefile.cvs
  ...
   all:
  +       y2tool y2automake
          autoreconf --force --install
  ...
  cvs server: SUBDIRS is a new entry, no comparison available
  ...
  Index: myproject.spec.in
  ...
  -# neededforbuild autoconf automake ...
  +# neededforbuild yast2-devtools autoconf automake ...
  ...
  ```

  Important: *Don't build yet*, otherwise `Makefile.am` will be regenerated and "cvs ci" will complain when trying to check all this in.

- Check your changes in:

  ```
  cvs ci
  ```

- Test-build your package locally:

  ```
  make -f Makefile.cvs && make && sudo make install
  ```

  You should now have a new `Makefile.am`.

9.1.4. Translation (po) Modules)

For YaST2 translation modules (`yast2-trans-...`), the `Makefile.am` in the `po` subdirectory is automatically generated as well. The strategy for that is slightly different, though: The resulting `Makefile.am` is combined from `Makefile.am.top`, `Makefile.am.center`, and `Makefile.am.bottom`. The top and bottom files are used from the current project, if there is such a file; otherwise, all files are taken from `/usr/share/YaST2/data/devtools/admin/po`. The center part is *always* taken from there.

Add custom `make` targets or variable definitions to the top or bottom part as required. This may only be necessary for the `yast2-trans-...` data modules (e.g., keyboard, mouse, printers).

The migration script takes care of that: It migrates the `po/` subdirectory, too, if there is one - and if there is a `Y2TEXTDOMAIN` file in the the project toplevel directory. For data modules, the migration script backs up the existing `Makefile.am` to `Makefile.am.bottom` (or to `Makefile.am.old`, if there already is a file named `Makefile.am.bottom`). *Make sure to edit this file* and get rid of duplicate parts before checking in.

9.1.5. create-spec: Automatic creation of the `.spec` file

make package-local handles the file *.spec.in. The file `package/*.spec` created in the time of "make -f Makefile.cvs" is overwitten in the time of "make package(-local)" with the package/*.spec created by **y2tool create-spec**. But it should have the same content.

With **create-spec** you can use more 'macros' in the *.spec.in:

@HEADER-COMMENT@	writes the SuSE .spec comment
@HEADER@	writes the usual header except BuildArch:, Requires:, Summary:
@PREP@	writes %prep with %setup
@BUILD-YCP@	writes %build with usual make
@INSTALL-YCP@	writes %install with usual YCP make install
@CLEAN@	writes %clean with removing RPM_BUILD_ROOT

So the new *.spec.in could look like:

```
@HEADER-COMMENT@
# neededforbuild  autoconf automake ycpdoc yast2-testsuite ...

@HEADER@
Requires:       yast2 yast2-trans-XXpkgXX yast2-lib-wizard yast2-lib-sequencer
BuildArchitectures:     noarch

Summary:        Configuration of XXpkgXX

%description
This package is a part of YaST2. It contains the necessary scripts to
configure XXpkgXX.

@PREP@

@BUILD-YCP@

@INSTALL-YCP@

@CLEAN@

%files
%dir @yncludedir@/XXpkgXX
/...
```

9.1.6. Overview of Paths

These paths are defined in your `configure.in` generated by `y2autoconf` and substituted by `create-spec`. That means they are accessible in all your Makefiles and can be uses in `spec.in` files.

yast2dir=${prefix}/share/YaST2	not for direct use

docdir=${prefix}/share/doc/packages/\$RPMNAME	for documentation
ybindir=${prefix}/lib/YaST2/bin	for all yast2 programs not be started by the user.
plugindir=${libdir}/YaST2/plugin	for loadable plugins
included-ir=${prefix}/include/YaST2	for c header files
localedir=${yast2dir}/locale	for translations files
clientdir=${yast2dir}/clients	for ycp clients
moduledir=${yast2dir}/modules	for ycp modules
schemadir=${yast2dir}/schema	for schema files (autoyast, control file)
yncludedir=${yast2dir}/include	for ycp includes
scrconfdir=${yast2dir}/scrconf	for scr files
desktop-dir=${prefix}/share/applications/YaST2/modules	for .desktop files (former *.y2cc)
execcompdir=${prefix}/lib/YaST2	for external programs that are yast2 components. here you have to append servers, servers_non_y2, clients or cli-ents_non_y2.
ydatadir=${yast2dir}/data	for general data
imagedir=${yast2dir}/images	for non theme-able images
themedir=${yast2dir}/theme	for theme-able images (every theme must provide the same list of images)

In Makefile.am you can simply say

```
ybin_PROGRAMS = y2base
```

when you what the program y2base to be installed in ybindir. No need to change bindir or even pre-fix.

In the files section of your spec.in file use something like this:

```
@scrconfdir@/*.scr
```

Remember that the asterisk is only save when using a BuildRoot (and please use a BuildRoot).

If you need a define in a C++ file you have to pass it to the compiler. Simply use

```
AM_CXXFLAGS = -DPLUGINDIR=\"${plugindir}\"
```

in your Makefile.am.

9.1.7. Toplevel `make` Targets in Detail

9.1.7.1. `make package-local`

Create a tarball from your module and put it into the `package/` directory. This also creates a spec file from the .spec.in file.

9.1.7.2. `make package`

This checks for cvs consistency (see `make check-tagversion`) and whether or not you correctly tagged that version (don't forget to increase the version number in `VERSION`!), then does everything `make package-local` did.

9.1.7.3. `make check-tagversion`

This is performed by `make package` prior to actually creating a tarball: It checks whether or not you correctly tagged the current version. Use "`y2tool tagversion`" to do that once you increased the version number in `VERSION`.

Note

You will very likely never call this manually.

9.1.7.4. `make check-cvs-up-to-date`

This is performed by `make package` prior to actually creating a tarball: It checks whether or not everyting in this directory tree is checked into CVS. Any modified, added or removed files make this check fail.

Note

You will very likely never call this manually.

9.1.7.5. `make checkin-stable`

This makes a package (i.e. it does everything "`make package`" does and checks it into the correct SuSE Linux distribution.

Note

This requires `/work/src/done` to be mounted via NFS.

9.1.7.6. `make stable`

Just an alias for "`make checkin-stable`".

9.2. YaST2 Logging

9.2.1. Introduction

During execution *YaST2* components create log messages. The purpose is to inform the user or the programmer about errors and other incidents.

The logging should be used instead of `fprintf(stderr,...)` to create logmessages of different types. It can be better controlled, what to log and what not, where to log and how to log.

9.2.2. Quick start

- Use `y2debug()` for debugging messages, `y2warning()` for warnings and `y2error()` for error messages, syntax is same as `printf(3)`.
- Set `"export Y2DEBUG=1"` in your `.profile` or run `"Y2DEBUG=1 yast2"`.
- If `root`, see `/var/log/YaST2/y2log`, otherwise `~/.y2log` for the output.
- In the `y2log`, entries are uniquely identified by the filename and line number.

9.2.3. Logging levels

There exist six different log levels denoting incidents of different importance:

0: DEBUG	Debug messages, which help the programmers.
1: MILESTONE	Normal log messages. Some important actions are logged. For example each time a *YaST2* module is started, a log entry is created.
2: WARNING	Some error has occured, but the execution could be continued.
3: ERROR	Some major error has occured. The execution may be continued, but probably more errors will occur.
4: SECURITY	Some security relevant incident has occured.
5: INTERNAL	Internal error. Please enter into Bugzilla or directly contact the programmers.

In the default setting the levels 1-5 are logged, level 0 (DEBUG) is switched off. See the Logging control and Environment control for more details how to control the logging and its levels.

9.2.4. Logging functions

According to the logging levels, use the following logging functions:

```
void y2debug(const char *format, ...);
void y2milestone(const char *format, ...);
void y2warning(const char *format, ...);
void y2error(const char *format, ...);
void y2security(const char *format, ...);
void y2internal(const char *format, ...);
```

The parameter `format` is the format string like the one for `printf(3)`

9.2.5. Additional functions

9.2.5.1. Setting the logfile name

```
void y2setLogfileName(const char *filename);
```

This function sets the logfile name. If the name cannot be open for writing (append), it use the default logfiles. If you want to output the debug log the `stderr`, use `"-"` as the argument for the y2setLogfileName:

```
y2setLogfileName("-");
```

9.2.5.2. Universal logging functions:

```
void y2logger(loglevel_t level, const char *format, ...);
void y2vlogger(loglevel_t level, const char *format, va_list ap);
```

These functions are provided probably only for those who don't want to use the regular logging functions. For example for setting the loglevel acording to some rule.

9.2.6. Components

As the filenames are not unique over the whole *YaST2* source, you can specify the component name. Then the pair of the component name and the filename will uniquely identify the message.

Note: I think that the filenames should be self explaining and thus unique overall the whole source. Then the component name can be removed, but as now the filename is not unique, you can optionally specify the component name.

As the component is a more general property then filename, it should be same in all messages in one file. So for one source file it is defined only once, at the beginning of the file. And because of implementation purposes (just) before the inclusion of `y2log.h`:

```
#define y2log_component "y2a_mods"
#include <ycp/y2log.h>
```

9.2.7. Logfiles

The *YaST2* log is written to a file. If you work as normal user, the default logfile is `~/.y2log`. If you work as root, the file is `/var/log/YaST2/y2log`. The logfile is created with the permissions 600, since it may contain secret data when the debug level is turned on.

If the logfile cannot be open, the `stderr` is use instead.

9.2.8. Log entries

Each log entry consist of these fields:

`date`	The date when the log entry has been made.
`time`	The time when the log entry has been made.
`level`	The log entry level. See Logging levels.
`hostname`	The hostname of host where the yast2 runs.
`pid`	The process ID of the yast2 process.
`component`	The name of the current component. Optional and probably obsolete.
`filename`	The name of the source file where the log entry has been made.
`function`	The name of the function where the log entry has been made.
`line`	The line number where the log entry has been made.
`message`	The text of the log message.

The output format:

```
date time <level> hostname(pid) [component] filename(function):line message...
date time <level> hostname(pid) filename(function):line message...
```

Example:

```
2000-10-13 15:35:36 <3> beholder(2971) [ag_modules] Modules.cc(quit):22 io=7
2000-10-13 15:35:37 <0> beholder(2971) ModulesAgent.cc(main):23 irq=7
```

9.2.9. Logging control

The log control uses a simple ini-like configuration file. It is looked for at /etc/YaST2/log.conf for root and at $HOME/.yast2/log.conf for regular users.

Example log.conf file could look like:

```
    [Log]
    file = true
    syslog = false
    debug = false

    [Debug]
    YCP = true
    agent-pam = true
    packagemanager = false
```

"syslog=true", which basically means remote-logging. The similar option "file=true" means use the usual log files for logging. You could also turn those off which means no logging would be done at all, but rather don't do that ;-)

The "debug=true" means basically the same as Y2DEBUG=1 (that envirnoment variable overrides the log.conf settings) and that is log by default all debug messages (if not said otherwise).

You can turn debuggin on ("agent-pam=true") for a particular component (even if "debug=false") and also turn debugging off (for the case that "debug=true").

To provide a useful example, normal developers would need something like this $HOME/.yast2/log.conf (and unset Y2DEBUG):

```
    [Debug]
    YCP = true
    agent-pam = true
```

It means turn YCP debug messages on and also turn on some particular agent. The other debug are in most uninteresting, so let them turned off.

During installation , define the variable "Loghost" on the command line with the log server ip address (Loghost=192.168.1.1) and all messages will be sent to this host. If you add y2debug, debugging will also be activated in log.conf.

On the server side, using syslog-ng, you can have logging per host using the following filters:

```
source network {
    tcp();
    udp();
};

destination netmessages { file("/var/log/messages.$HOST"); };
log { source(network); filter(f_messages); destination(netmessages); };
```

9.2.10. Environment control

Additionally to the usual logfile control you can control some logging feature by the environment variables.

Y2DEBUG	By setting this variable to an arbitrary value you turn on the debug log output. But only when entry control is not covered by the usual logfile control.
Y2DEBUGALL	By setting this variable to an arbitrary value you turn on the debug log output. Everything will be logged.
Y2DEBUGSHELL	By setting this variable to an arbitrary value you turn on the debug log output for the bash_background processes.
Y2MAXLOGSIZE	By this variable you can control the size of logfiles. See Logfiles for details.

M	By this variable you can control the number of logfiles. See Logfiles for details.

Example: call the module `password` with *QT* interface and debugging messages set to on:

```
bash$ Y2DEBUG=1 yast2 users
```

9.3. Coding In YCP

As with any other programming language, there are some rules of “DOs and DON'Ts” in `YCP` too. During the development of the `YaST` installer the developers realized that some particular ways of doing things are usually better than some others. Furthermore some “standards” have been worked out, that (if heeded) make the resulting code uniform to some extent which makes it much easier to understand code written by other people. The following links give some hints that should ease programmers life.

9.3.1. Coding Rules

Any fool can write code that a computer can understand. Good programmers write code that humans can understand.
—Martin Fowler in: Refactoring, improving the design of existing code

Having multiple developers working on the same source code needs a basic set of coding rules to adhere to. A proper code layout makes it *a lot* easier for others to read, understand, enhance, debug, and clean-up code.

Having a coding style is quite common, two of the more prominent examples are *The Linux kernel coding style* and the *GNU coding standard*

The document on hand describes how to lay our code written in YCP. How to name your variables and functions, how to place braces, and how to indent blocks.

Every programmer usually has her/his own style of writing code. The rules presented here might not match your personal preferences, but will help to work on the code as a team. Helping out and fixing bugs will be easier with a common coding style.

The following set of rules tries to be complete, but probably isn't.

These rules should apply to C, C++, and YCP code alike, even though the examples use YCP.

9.3.1.1. The file header

Every file must begin with a proper header. This header should be started within the first 10 lines of the file, so it is visible when loading the file in an editor.

The file header must include

- the file name
- the file purpose summary (in one line !)
- the authors name and email address
- the CVS $Id: coding-rules.xml,v 1.1 2004/10/03 16:54:44 nashif Exp $
- a few lines description about the contents

The '$Id: coding-rules.xml,v 1.1 2004/10/03 16:54:44 nashif Exp $' will be replaced automatically when the file is handled by CVS. Don't touch this line afterwards, it's controlled by CVS then.

Do	Don't

Do	Don't

/**
 * File:
 *   io.ycp
 *
 * Module:
 *   Security configuration
 *
 * Summary:
 *   Input and output functions.
 *
 * Authors:
 *   Michal Svec <msvec@suse.cz>
 *
 * $Id: coding-rules.xml,v 1.1 2004/10/03 16:54:44 nashif Exp $
 *
 * There are in this file all functions needed for
 * the input and output of security settings.
 */
	{
	  // a small example with no version and no hint
	  // about the author.
	  return 42;
	}

9.3.1.2. Indendation

Among developers, indentation of code is one of the most heated points of discussion. There are several 'good' ways to use whitespace when writing sourcecode, all are 'right' in some respect.

The only bad indentation is no indentation at all. To make code easy to read, a common way of using whitespaces is needed across a team of developers:

- indent by 4 spaces
- tabs are 8 spaces
- always indent

Only a few lines of a file are allowed to be not indented. These are the initial comment lines of the file header and the opening and closing braces around the code.

Do	Don't

/*
 ... initial header
 */
 	
{   // opening brace at start of code
 	
    // my first variable, 4 spaces indentation
 	
    integer first_int_variable = 42;
 	
    if (first_int_variable > 42)
    {
        // 8 spaces (== 1 tab character) indentation
        doSomething ();
    }
    else
    {
        somethingDifferent();
    }
 	
    // final return
 	
    return first_int_variable;
 	
}   // closing brace
	/* ... initial header */
	
	{ // opening brace at start of code
	
	// my first variable, bad indentation
	
	integer first_int_variable=42;
	
	if(first_int_variable>42) doSomething ();
	else somethingDifferent();
	return first_int_variable;}

9.3.1.3. Whitespace

Whitespaces (blank, tab, and newline characters) are allowed anywhere in the code. Proper use of whitespace does make code a lot easier to read and more pleasing to the eye.

Blanks are mandatory at the following places:

- before an open parantheses

  - at function calls
  - at if and while expressions
- after a comma
  - at parameter lists in function calls
  - at list and map elements
- before and after a binary operator. '=' is a binary operator

Newlines are mandatory at the following places:

- before and after every opening brace.
- before and after every closing brace.
- after the initial variable declarations, before the first statement.
- to separate functional groups. a functional group is this sense is a set of variable declarations before a group of computational statements. another example is grouping in pre-processing, computing, and post-processing often used in larger modules.

Feel free to use whitespaces at other places you find appropriate.

Some explanation to the above "Don't" example:

- there is no blank before the "(" in the if, while, and callFunction lines.
- there is no newline to properly separate the group of variable declarations from the computational statements.
- there are two variable declarations in the if () block. Without whitespace this isn't easily visible.

Do	**Don't**

``if (bool_flag) ...``
 	
``while (stay_in_loop) ...``
 	
``callFunction ( value1, value2 );``
 	
``list a_list = [ 1, 2, 3, 4 ];``
``map a_map = [ 1:`first, 2:`second ];``
``boolean test_flag = true;``
 	
``if (test_flag)``
``{``
``    integer one = 1;``
``    boolean two_flag = false;``
 	
``    callFunction (one, two_flag);``
``}``
	``if(bool_flag) ...``
	``while(stay_in_loop) ...``
	``callFunction(value1,value2);``
	``list a_list=[1,2,3,4];``
	``map a_map=[1:`first,2:`second];``
	
	``boolean test_flag=true;``
	``if (test_flag){``
	``    integer one=1;boolean two_flag=false;``
	``    callFunction(one,two_flag);}``

9.3.1.4. Naming of variables

This rule should be easy if you keep in mind that other developers want to read and understand your code.

General rule: Use speaking variable names. By reading the name of a variable it should be immediately clear

- what kind of value variable represents
- how the variable is used
- probably its scope

There is no restriction in the length of a variable, use this fact!

To make a clear destinction of variable names vs. function names, use '_' in variables and upper/lower case in function names.

Do	Don't
`boolean is_sparc = (architecture == "sparc")`	`boolean n = (architecture == "sparc");`
`list probed_modems = SCR::Read ( .probe.mode`	`list dev = SCR::Read(.probe.modem);`
`integer list_index = 0;`	`integer i = 0;`
`map a_modem = $[];`	`map m = $[];`
`while (list_index < size (probed_modems))`	`while (i<size(dev))`
`{`	`{`
`    a_modem = select (probed_modems, list_in`	`    m = select (dev, i);`
`    doSomething (a_modem, is_sparc);`	`    func (m, n);`
`    list_index = list_index + 1;`	`    i=i+1;`
`}`	`}`

9.3.1.5. Naming of functions

Like variables, function names should speak for themselves. So the above arguments for naming variables apply also to functions.

But instead of '_', use mixed upper and lower case to make the names 'speak'.

It is also helpful to distinguish between global and local functions. Local functions should start with a lower case letter, global functions with an upper case letter.

Do	Don't
`// this is a local function`	`f1 (a_string, file_name);`
`writeStringToFile (a_string, file_name);`	`gs = rgs();`
`// this is a global function`	
`global_settings = ReadGlobalSettings ();`	

9.3.1.6. Blocks and Braces

There are more ways to place braces around a block than there are computer languages which use '{' and '}'.

For YaST2, only two rules about braces are important:

- the opening and closing brace are on the same indentation level.
- the opening brace increases the indentation level by one (which equals 4 spaces).

9.3.1.7. if-then-else, while, etc.

This one is really easy, always use a block for if and else cases and while statements.

Do	Don't
`// start of file`	`{integer initial_index = 0;`
`// first brace doesn't have any indentation`	`while (initial_index < 10){`
`{`	`initial_index = initial_index + 1;}`
`    // 4 spaces indentation`	`return initial_index;}`
`    integer initial_index = 0;`	
`    while (initial_index < 10)`	
`    {`	
`        // incremented indentation level`	
`        initial_index = initial_index + 1;`	
`    }`	
`    return initial_index;`	
`}`	

9.3.1.8. Comments

Comment every function with a structured comment. The comments will be used for the documentation generation. The syntax is similar to ydoc (kdoc).

```
/**
 * Update the SCR from the map of all security settings
 * @param settings a map of all security settings
 * @return boolean success
 */

define SecurityWrite(map settings) ``{
    ...
}
```

9.3.1.9. Other habits

This is a small list of other things to consider when writing code.

- Superfluous whitespace in the source. This was discussed on the linux kernel mailing list, see Kernel Traffic #103 For 19 Jan [http://kt.linuxcare.com/kernel-traffic/kt20010119_103.epl#11] for more. For vi users, adding:

  ```
  syntax on
  let c_space_errors=1
  ```

  should help. In Emacs 21, set the variable `show-trailing-whitespace`, also see `whitespace.el`
- Replace blanks with tabs. Since a tab character equals 8 blanks, multiple blanks should be replaced by tabs.

9.3.2. Examples of bad code

Bad	Good

`any ret = boolean_function ();`
 	
`if (ret)`
`{`
`      ....`
	`boolean ret = boolean_function ();`
	
	`if (ret)`
	`{`
	`      ....`

Here the type for "`ret`" is known, since it is used as a boolean in the if expression. So don't use "`any`" as a type declaration is you know better.

Bad	Good

`term|any x = nil;`
 	
`if (boolean_value)`
`{`
`   x = `` `VBox (...); ``
`}`
`else`
`{`
`   x = `` `Empty (); ``
`}`
	`` term x = `Empty (); ``
	
	`if (boolean_value)`
	`{`
	` x = `` `VBox (...); ``
	`}`

Why invent an extra "`else`" case for a simple expression?

OK, for complex things where one can't say 'this will be used 90% of the time', an else-case is needed. But then it's still better to initialize the variable with a dummy value of the expected type (i.e. ``term x = `Dummy();``) instead of "`nil`".

Bad	Good

```
any ret = UserInput ();
...
return (ret == `ok);
```
	```
	symbol ret = UserInput ();
	...
	return (ret == `ok);
	```

Similar to the "`boolean`" example above, we know the type for "`ret`" in advance.

Bad	Good

```
any|list x = list_function ()
if ( x != nil && select (x, 1) == true)
{
      ....
```
	```
	list x = list_function ()
	if ( x != nil
	    && (size (x) > 1)
	    && (x[1]:false == true))
	{
	      ....
	```

This is a particulary bad example since the code was plain wrong initially. The list variable was just tested for "`nil`" but expected to have two elements.

9.4. Check YCP Syntax

9.4.1. Quick Start

Simply invoke `check_ycp` with the YCP file(s) you wish to check as arguments:

```
check_ycp myfile.ycp
```

```
check_ycp *.ycp
```

The error messages should be self explanatory. They stick to the GNU standards for error messages, so you can use your favourite editor's (e.g. Emacs) function to process them.

Most of the checks can individually be turned off. Type

```
check_ycp -h
```

for a complete list of command line options. Those options are intentionally not listed here since such a list would inevitably be outdated before too long.

9.4.2. Why this Document?

Even though using `check_ycp` is pretty straightforward, some background information is useful in order to understand what it does, its output and the limitations of this tool - in short, what you can expect it to do and why not to blindly rely on it.

`check_ycp` is far from fool proof. In fact, it is pretty dumb. It just tries to parse YCP code ("try" is the operative word here!) and applies various checks for obvious programming errors. Some errors it will catch and report, many it will not. But we (i.e. the YaST2 core development team) decided we'd rather have a tool with limited capabilities than none at all.

Another reason for writing this document is pointing out why we try to enforce certain things, many of which are because of ergonomics or mere conventions in developing as a team, not true requirements of YaST2 or the YCP language.

9.4.3. Header Comment Checks

The YaST2 team uses a standardized file header format for YCP modules. Standard fields are included there for various purposes - see the "why" sections of the individual checks.

Everything up to the first opening brace "`{`" outside a comment is considered part of the header. Nothing outside this portion of the file is checked. You may, however, open and close as many comments as you like up to this opening brace.

Leading asterisks "`*`" at the start of lines are silently discarded since they are often used to beautify multi line comments.

The comment markers themselves of course are also discarded for the checks:

- `/*`
- `*/`
- `//`

9.4.4. Filename Check

What

If present, the contents of a `Module` field is checked against the current file name.

Why

Much code gets written by copying existing code. There is nothing wrong with that (in fact, it saves a lot of work), but when you do, please change fields accordingly - fields like `Author`, `Maintainer`, `Purpose` etc. - and the file name in `Module`.

How

This file name is particularly easy to check (plus, it's the only one that can reliably be checked), so `check_ycp` checks it: It compares the base name (not the complete path) of the current file to what you specified in `Module:` in the header.

9.4.5. Author / Maintainer Entry Check

What

`check_ycp` checks the file header for presence of at least one of

- `Author:`
- `Authors:`
- `Maintainer:`
- `Maintainers:`

If found, each entry is checked for some contents, i.e. it may not be completely empty (but use whitespace as you like).

The contents must include something that looks like an e-mail address.

Why

There must be at least one person to contact when there are any problems or questions about the module. The full name is desired, but at least an e-mail address must be there to get in contact with the maintainer or the author.

How

The fields are checked for presence of something like `somebody@somewhere.domain` - in fact only for something before the at sign "@" and something with a period "." behind it.

9.4.6. CVS Id: Marker Check

What

Presence of a Id CVS / RCS identity marker is checked, e.g.

```
            Id myfile.ycp,v 1.2 2001/02/14 18:04:50 sh Exp
```

Why

This CVS / RCS ID is the only way of finding out exactly what CVS revision the file has and what change date. The file date (what `ls -l` shows) is absolutely unreliable and irrelevant: This may have changed just by copying the file around which didn't change anything.

This is important for bug tracking and for finding and fixing bugs - only when a developer knows what version of a file has been used he has a chance to reproduce a bug - or even make sure that a supposedly fixed bug didn't turn up again.

How

Presence of

```
Id:
```

is checked. There may be more characters before the closing dollar sign "`$`", but the exact contents is not checked.

Note

When creating a new file, it is absolutely sufficient to include the unexpanded string ("`Id:`") somewhere in the file. CVS or RCS will automatically expand this to the full ID string.

9.4.7. Translatable Messages Checks

9.4.7.1. `textdomain` Check

What

If there is any message that is marked for translation with `_("...")`, there must be a `textdomain` statement.

Why

The YaST2 translator module needs to know where to take the messages to be translated from. This is what the `textdomain` specification does.

Technically one `textdomain` statement somewhere in the YCP program would be sufficient, i.e.

include files or modules called with `CallFunction()` don't really require an additional `textdomain` specification.

However, it is highy recommended all YCP files with translatable messages include their own `textdomain` statement so each YCP file is self-sufficient in that regard, thus more easily reusable for other purposes. This policy is enforced with this check.

How

After being stripped of all comments, the entire YCP code is scanned for the translation marker sequence: An underscore immediately followed by an opening parenthesis: `_(`

If this sequence is found, presence of translatable messages is assumed. If no `textdomain` statment is found there will be an error.

On the other hand, if there is no text to translate, a `textdomain` statement is not necessary (but it can't hurt).

Note

Theoretically the "`_(`" sequence contained in a literal string (i.e. within double quotes `"..."`) could falsely trigger this error, too. But if you do that, you are very likely to run into trouble with other tools as well - most likely even the original `getext` tools regularly used to extract the messages for translation. Bottom line: Don't do that.

9.4.8. RichText / HTML Sanity Check

Literal strings in YCP code that contains HTML tags are usually help text that will be displayed in the YaST2 RichText widget. This HTML text is subjected to the sanity checks explained below.

Please notice that everything within double quotes `"` will be checked that contains anything surrounded by angle brackets <...> - i.e. anything that looks remotely like an HTML tag. Unknown tags will be silently ignored, but the entire text within the quotes will be checked.

Limitation: If a portion of help text lacks any HTML tag, it will not be checked since it will not be recognized by `check_ycp` as help text. Such completely wrong portions of help text will slip through undetected, thus unchecked.

9.4.8.1. Completeness of <p> / </p> Paragraph Tags

What

Each HTML text must start with a `<p>` tag and end with a `</p>` tag.

There must be a corresponding closing `</p>` tag for each opening `<p>` tag.

Why

This is a basic requirement of HTML. The underlying YaST2 widgets may or may not be forgiving enough to tolerate missing tags, but we'd rather not rely on that.

Besides, no other types of paragraphs other than plain text paragraphs <p> ... </p> are desired in YaST2 help texts - in particular, no large font boldface headings etc.

How

See the intro of this section.

9.4.8.2. Text Before, After, Between Paragraphs

What

For each portion of HTML text:

- No text before the first `<p>` tag is permitted.
- No text after the last `</p>` tag is permitted.
- No text between a closing `</p>` and the next opening `<p>` tag is permitted.

Why

Each of those cases is a simple yet common HTML syntax error.

How

See the intro of this section.

9.4.8.3. No More Than One Paragraph per Message

What

Each single portion of HTML text may contain exactly one paragraph, i.e. one <p> ... </p> pair.

Why

This is a convention to make life easier for the translators.

The tools used for extracting translatable texts from the sources (GNU `gettext`) detect differences between the last translated version of a message and the current message from the latest source. They mark such messages as *fuzzy*, i.e. the (human) translator is asked to have a good look at it and decide whether there has been a real change in the message (thus it needs to be retranslated) or just a cosmetic change (fixed typo, inserted whitespace, reformatted the paragraph etc.).

This is a tedious task and it gets more tedious the longer each individual portion of text becomes. Changes from the old to the new version are hard to find if the portions are very long.

Plus, if they are that long it is very likely that always somewhere something has changed, thus the entire text is marked as *fuzzy* and needs careful manual checking which is not really necessary for all the text.

Workaround

Split your help texts and use the YCP string addition operator to put them together.

Don't:

```
help_text = _("<p>
bla blurb bla ...
blurb bla blurb ...
bla blurb bla ...
</p>
<p>
bla blurb bla ...
blurb bla blurb ...
bla blurb bla ...
</p>");
```

Instead, do:

```
// Help text (HTML like)
help_text = _("<p>
bla blurb bla ...
blurb bla blurb ...
bla blurb bla ...
</p>");
```

```
// Help text (HTML like), continued
help_text = help_text + _("<p>
bla blurb bla ...
blurb bla blurb ...
bla blurb bla ...
</p>");
```

Please also notice the *comments* for the translators just above the text. The `gettext` tools will automatically extract them along with the text to translate and put them into the `.po` file. The translators can use them as additional hints what this text is all about.

How

See the intro of this section.

9.4.8.4. Excess Forced Line Breaks <br> after Paragraphs

What

Forced line break tags `<br>` are discouraged, especially after a paragraph end tag `</p>`.

Why

Such forced line breaks are plain superfluous. The HTML renderer will format the paragraph automatically - after each paragraph there will be a newline and some empty space to set each paragraph apart from the next.

There is no need nor is it desired to add extra empty space between paragraphs. This just looks plain ugly, even more so if this results in different spacings between several paragraphs of the same help text.

The most superfluous of those excess line breaks are those at the very end of a help text - after the last paragraph. Not only are they not good for anything, they sometimes even cause a vertical scroll bar to be displayed even though this would not be necessary otherwise.

Plus, there have been cases where erstwhile last help text paragraphs had been rearranged so they now are in the middle of the help text - but unfortunately the trailing `<br>` tag had been forgotten and moved along with the paragraph, thus causing different inter-paragraph spacings.

To make things even worse, fixing this breaks the translation for the affected paragraph: It will be marked as *fuzzy* just because of this even though it has not really changed.

We cannot entirely get rid of the `<br>` tags (but we would like to). Sometimes they are needed *within* paragraphs. But at least those at the end of paragraphs we can do without.

How

`<br>` after `</p>` (maybe with anything in between) is rejected. All other `<br>` tags are silently ignored.

9.4.9. Widget / UI Function Parameter Checks

Parameters to YaST2 UI widgets plus some commonly used functions (e.g. `Wizard::SetContents()`, `Popup::Message()` etc.) are checked where possible - if the parameters are simple string constants, maybe surrounded by translation markers ("`_("`...`")`").

Optional widget parameters like `` `opt(...) `` or `` `id(...) `` are ignored.

The following examples *will be checked:*

```
PushButton("OK");
PushButton( _("Cancel"));
```

```
PushButton(`id(`apply), _("Apply"));
```

```
PushButton(`opt(`default), _("OK"));
```

More complex parameters like variable contents or YCP terms cannot be checked.

The parser used in `check_ycp` for that is really dumb. In fact, it only scans for keywords like `PushButton` outside string constants, tries to find the corresponding matching pair of parentheses "`(`...`)`" and splits everything inside into comma-separated subexpressions.

Only the most basic of those subexpressions are checked - only simple string constants `"..."` or string constants marked for translation `_("...")`.

The following examples *will not be checked:*

```
CheckBox( "/dev/"+device );
```

```
CheckBox( sformat("/dev/%1"), device );
```

```
CheckBox( GetDevName() );
```

```
        string message = "OK";
        PushButton( message );
```

9.4.9.1. Keyboard Shortcut Check

What

Widgets that can have a keyboard shortcut (one character marked with an ampersand "&") are checked for presence of a keyboard shortcut.

Note

Consistency of the keyboard shortcuts is *not* checked, only presence. `check_ycp` cannot know which widgets will be on-screen at the same time, thus it cannot find out whether the same keyboard shortcut has been assigned twice to different widgets.

Why

This is for users whose mouse doesn't work (especially during installation time) as well as for experienced users who prefer working with the keyboard. Navigation from one widget to another is much easier when each widget that can get the keyboard focus can be reached with an *[Alt]* key sequence rather than repeatedly using the *[Tab]* key and/or the cursor keys.

There may be a lot more widgets that can have keyboard shortcuts than you expected. Basically, every widget that can somehow be operated with the keyboard (even if it is only scrolling) and that has an associated label (within, above or beside) can have a keyboard shortcut and should get one.

How

The widget parameter that acts as a label is checked for presence of exactly one ampersand "&".

See the widget checks of this section for more.

9.4.9.2. Translatable Messages Check

What

Widget parameters that are displayed literally as text are checked for translation markers ("`_("...")`").

Why

Every text message that ever gets to the end user is to be translated into the user's native language. This can only be made sure if the message is marked for translation.

How

See the intro of this section.

9.4.10. Standardized Lib Function Checks

9.4.10.1. Duplicate Definitions of Wizard Lib Functions

What

Presence of definitions of functions from the *wizard lib* (Package `yast2`) outside the *wizard library* itself is checked such as `Wizard::SetContents()` etc.

Why

At the start of YaST2 develompent there was no other way of sharing code other than simply copying it. Those days are gone; YCP now supports an `include` mechanism similar to C or C++.

Very general code like how to create the typical YaST2 wizard window layout has now been moved to the *wizard lib*, a collection of include files that provide such facilities. We want to get rid of duplicate code as soon as possible for obvious reasons (consistency, maintainability, efficiency).

How

If "`define`" followed by one of the known function names of the *wizard lib* is found outside the file where this is supposed to be, a warning is issued. Both function names and file names are hardwired within `check_ycp`.

9.4.10.2. Definitions and Usage of Obsolete Functions

What

Usage or presence of definitions of known obsolete functions is checked, e.g. `Popup::Message()`, `Popup::YesOrNo()` etc.; using an equivalent replacement function from the *wizard lib*'s `common_popups.ycp` include file is suggested.

Why

Those functions are now superseded by those from `common_popups.ycp`. The replacement functions usually require less parameters (thus are easier to use) and use a common and consistent widget layout.

How

The definitions are checked very much like the wizard function definitions above; function and file names are hardwired here as well.

Usage of the obsolete functions is checked simply by checking for occurrence of the function name followed by an opening parenthesis (maybe with whitespace in between) somewhere in the code.

9.4.10.3. Usage of Predefined Messages

What

Presence of predefined message strings is checked, e.g. "`&Next`", "`&Back`" etc.; using a corresponding function from the *wizard lib* (`Label` Module) is suggested, e.g. `Label::NextButton()`, `Label::BackButton()` etc.

Why

- Ease the burden on the translators - with the predefined messages they don't need to translate the same standard texts over and over again.
- Consistent messages for the same type of buttons etc. throughout all of YaST2.
- Consistent keyboard shortcuts for the same button throughout all of YaST2.
- If we ever need to change one of those standard messages, we can do that centralized.

How

The YCP code, stripped of comments, is checked for any one of the predefined messages (including any keyboard shortcuts that may be there), surrounded by translation markers ("`_("`...`")`").
Limitations

Differences in spelling or only in whitespace will not be caught. If there is no or another keyboard shortcut, the message will not be considered the same - so if anybody uses "`Ne&xt`" rather than "`&Next`", this will go undetected.

9.4.11. Alternative Variable Declarations

What

Alternative variable declarations are rejected, e.g.

```
string|void value = anything();
symbol|term result = UI::UserInput();
integer|string x = 42;
```

Why

Just about the only situation where this made sense was when a variable might sometimes be `nil` to indicate some error condition. All other variants of this are of purely academic nature or (more likely) poor programming style. Since all YCP types can be `nil` now, however, this feature becomes totally redundant. It will very likely be dropped in the near future.

How

The entire YCP code, stripped of comments, is checked for occurences of one of the primitive YCP types (`string`, `integer`, `boolean`, `map`, `list`, `any`, `void` etc.) followed by a pipe sign `|` (maybe with whitespace before or after it) and another primitive YCP type.

9.4.12. Checking YCP Examples

You probably don't want to perform all of the available checks for simple YCP examples. Those should be concise and written for legibility rather than for completeness. They will usually not contain a standard format file header with all bells and whistles, no translation markers etc. - you don't want to bloat `HelloWorld.ycp` with all that stuff.

`check_ycp` has a special *example mode* for just this purpose: It turns off all checks that don't make sense for simple examples, yet allows you to use `check_ycp` anyway. If you think *"well, what's left then?"* think about the future. `check_ycp` can and will be expanded to cover more and more checks, and even your examples can benefit from it.

For simple YCP examples (and only for them, please!) invoke `check_ycp` with the `-x` command line option:

```
check_ycp -x HelloWorld.ycp
```

This turns off all checks that don't make sense for examples.

9.4.13. check_ycp and Emacs

`check_ycp` and Emacs go together well:

- Load a YCP file into Emacs.
- Invoke the Emacs compile command:

  ```
  M-x compile
  ```

- Edit the compile command ("`make -k`" by default) in the minibuffer; change it to the `check_ycp` command you wish to invoke (you only need to do this once for each Emacs session):

  ```
  check_ycp *.ycp
  ```

- Hit *Return*
- Use the `next-error` function to go to the next error `check_ycp` has reported. The corresponding YCP file will automatically be loaded into Emacs if needed, and Emacs will jump to the corresponding line within that file.

If you haven't done so already, you might want to bind the `compile` and `next-error` functions to keys in your `~/.emacs` file, e.g.

```
(global-set-key "f42" 'compile)
(global-set-key "f43" 'next-error)
```

The real challenge here is to find a key that is not already in use for some other important function.

If you are a real hardcore YCP hacker, you can even go so far and change the default compile command to `check_ycp` in `~/.emacs`:

```
(setq compile-command "check_ycp *.ycp")
```

9.4.14. Extending check_ycp

9.4.14.1. Adding new Widgets / UI Functions

Everybody should be able to add checks for a new widget or a new function that uses keyboard shortcuts (unlikely) or translatable messages (very likely) - even without any knowledge of Perl:

1. Locate the `check_widget_params()` function.
2. Add your widget to the list (the large regexp) near the function beginning, where all the other widgets are. Be careful not to include any whitespace (blanks or tabs) inside the parentheses. *Wrong:*

   ```
   ( MyNewWidget ) |
   ```

   OK:

   ```
   (MyNewWidget) |
   ```

3. Add an `elsif()` branch to the large `if()`...`elsif()`...`elsif()` construction:

   ```
   elsif ( $widget =~ /MyWidget/ )
   {
       check_keyboard_shortcut ( $widget, $line_no, 1, @args );
       check_translation       ( $widget, $line_no, 1, @args );
   }
   ```

You might have to change the third parameter according to your widget or function: This is the number of the parameter to be checked (the first one is 1) after all `` `opt() `` and `` `id() `` parameters have been removed.

Of course you can omit the keyboard shortcut check (`check_keyboard_shortcut()`) if it doesn't make sense for your widget or function.

If there is more than one parameter to be checked for translatable messages, add a call to `check_translation()` for each.

9.4.14.2. Other Extensions

Like Linus Torvalds once said: *"Use the source, Luke!"* ;-)

`check_ycp`'s sources are extensively commented, even the many regular expressions used there. But changing those regexps really requires some in-depth knowledge of regexps in general and Perl regexps in particular. If you feel unsafe, better not touch them.

Other than that, use your creativity.

9.5. The YaST2 Macro Recorder

Target Audience:

- YaST2 power users
- Quality assurance (Testers)
- Technical writers

9.5.1. Introduction

The YaST2 UI (User Interface) features a macro recorder and player that records user interaction during installation or system configuration and plays those user actions at a later time in the same scenario.

The Qt (graphical) and NCurses (text mode) user interfaces both support the macro recorder. It is also independent of graphics mode, display resolution, widget theme, terminal type or other details of the desktop being used: Not low-level input device (mouse or keyboard) events are recorded but logical user actions such as "Accept button was activated", and widget status information is saved in terms as "the *user name* input field contains *tux*", not individual keystrokes that might include lots of cursor movement and hitting the "backspace" key.

Macros recorded in one environment using the Qt UI may be played in another using the NCurses UI - unless, of course, the dialogs in either situation have completely different contents because extended features were used the other UI is not capable of. This should occur only very rarely, however.

9.5.2. Quick Start

Since most readers are impatient and just want to know how to get it going, here is a quick start guide - but PLEASE read the other sections anyway to avoid disappointment or even severe data loss:

- Start a YaST2 module in Qt (grahpical) mode - from the YaST2 control center, from the KDE control center, from the desktop menu or via command line.
- To record a macro, press **Alt-Ctrl-Shift M** The sequence of first Alt, then Ctrl, then Shift is important! A file selection box opens prompting you to enter a file name for the macro. Make sure you have write permission to the directory you select.
- Work with the YaST2 module as usual.
- Press **Alt-Ctrl-Shift M** again to stop recording. When the module is finished of course recording stops automatically.
- Start the same module again.
- To play a macro, press **Alt-Ctrl-Shift P**.

  A file selection box opens. Select the macro file you recorded earlier.
- Watch the YaST2 module replay the same you did when recording the macro.

Alternatively, you can also supply a macro on the **"y2base"** command line:

```
          /usr/lib/YaST2/bin/y2base some_yast2_module qt --macro /wherever/macro.ycp
```

```
          /usr/lib/YaST2/bin/y2base some_yast2_module ncurses --macro /wherever/macro.ycp
```

This is currently the only way to play macros with the NCurses (text mode) UI - it doesn't provide special key combinations for recording or playing macros (yet).

9.5.3. Purpose

The general idea of this macro recorder is to provide an easy way of automating repetetive tasks on the user level - for automated testing and to easily produce lots of screen shots in recurring situations.

For example, the SuSE Linux installation manuals include lots of screen shots that of course look differently in each language, and it is very desirable to have the screen shot in the same language as the rest of the manual. For the documentation department, this means that all required screen shots have to be made in all languages we ship translated manuals for, so all relevant installation scenarios have to be restaged with only the language being different. To make matters worse, the responsible person might not even understand all those languages and thus has to rely on guessing what button to click on.

With the macro recorder, he can record a macro in a language he understands, do all screen shots there, then restart the process, select another language and play the macro - all screen shots he took will then automatically be made in that language. Of course, not a button text like "Accept" is being recorded, but an internal logical button name that doesn't change depending on language.

9.5.4. What it is not

The macro recorder is not intended as a poor man's substitute for AutoYaST, the automatic untattended installation - even though it can be (mis-) used that way to some degree.

If you have lots of machines to install in a similar way, use AutoYaST, not the macro recorder: The macro recorder is dumb. It will blindly repeat whatever you did while recording the macro. If however at some other machine the situation is only slightly different, this might easily not work any more: If there are hard disks of different size or with a different partitioning scheme and you didn't rely on YaST2's automatic modes but created partitions manually, this might fail on the other machine.

Then you will get an error dialog, at which point your macro will no longer match the situation, but of course the macro will not realize this and happily keep playing user actions that are completely out of sync with the dialogs on-screen. Usually this will just cause lots of more error dialogs, but chances are that it might keep working for a while and cause data loss on that machine.

Note: This might even happen at the same machine if the environment just changed a bit - if, say, you added a hard disk partition in the first run, this might fail in the second run (or in the third or fourth run) because there is no more free space on the disk. Then you will also get error dialogs.

AutoYaST on the other hand takes all this into account and reacts in a much more intelligent way.

9.5.5. Quirks and Limitations

Given the intentions and target user group of the macro recorder, there are some known limitations that will very likely remain for the forseeable future:

Some widgets in the Qt UI "eat" the special key combinations. If one of those has the keyboard focus, pressing Alt-Ctrl-Shift M (or P) will have no effect. But there is an easy workaround: Simply move the keyboard focus with the "Tab" key to another widget like a push button - this will not change the environment for macro recording or playing. It is otherwise irrelevant to the macro recorder which widget has the keyboard focus.

Qt selection boxes are particular notable for this - you have to hit "Tab" in the language selection in the first dialog of a YaST2 installation before you can start recording or playing a macro there.

The software package manager user interface does not support the macro recorder at all. This was the tradeoff for getting a user interface that powerful: All that dialog is one large widget written purely in C++ unlike almost all other YaST2 dialogs.

If you use the macro recorder, don't go into the detailed software selection.

The Qt version of the YaST2 control center also does not support the macro recorder at all. It is a relatively basic Qt/C++ program that acts as a program launcher for the YaST2 modules, but it is not connected to them very closely. This was also a tradeoff: It is optimized for pretty looks and fast startup time.

9.5.6. Anatomy of a Macro

Here is an example of a macro recorded during the first part of a YaST2 installation:

In the language dialog, "German" was selected. Notice how there are no German texts to be seen anywhere inside the macro - only symbolic names are used.

Then in a popup asking whether to update or to do a new installation "installation" was chosen.

From the installation settings proposal "software" was selected to change the amount of software to install from "default system" to "minimal+X11".

Then the installation was started.

Example 9.1. YaST2 UI macro file

```
// YaST2 UI macro file generated by UI macro recorder
//
//     Qt UI: Alt-Ctrl-Shift-M: start/stop Macro recorder
//            Alt-Ctrl-Shift-P: Play macro
//
// Each block will be executed just before the next UserInput().
// 'return' before the closing brace ( '}' ) of each block relinquishes control
// back to the YCP source.
// Inside each block arbitrary YCP code can be added manually.

{
    {
        UI::ChangeWidget( `id (`language), `CurrentItem, "de_DE" ); // YSelectionBox

        // UI::MakeScreenShot( "/tmp/yast2-0000" );
        UI::FakeUserInput( `language );

        return;
    }

    {
        UI::ChangeWidget( `id (`language), `CurrentItem, "de_DE" ); // YSelectionBox

        UI::MakeScreenShot( "/tmp/screen-shots/language-selection.png" );
        // UI::MakeScreenShot( "/tmp/yast2-0001" );
        UI::FakeUserInput( `accept );

        return;
    }

    {
        UI::ChangeWidget( `id (`install), `Value, true ); // YRadioButton

        // UI::MakeScreenShot( "/tmp/yast2-0002" );
        UI::FakeUserInput( `ok );

        return;
    }

    {
        // UI::MakeScreenShot( "/tmp/yast2-0003" );
        UI::FakeUserInput( "software" );

        return;
    }

    {
        UI::ChangeWidget( `id ("Minimal+X11"), `Value, true ); // YRadioButton

        UI::MakeScreenShot( "/tmp/screen-shots/sw-base-selection.png" );
        // UI::MakeScreenShot( "/tmp/yast2-0004" );
        UI::FakeUserInput( "Minimal+X11" );

        return;
    }

    {
        UI::ChangeWidget( `id ("Minimal+X11"), `Value, true ); // YRadioButton

        // UI::MakeScreenShot( "/tmp/yast2-0005" );
        UI::FakeUserInput( `accept );

        return;
    }

    {
        // UI::MakeScreenShot( "/tmp/yast2-0006" );
        UI::FakeUserInput( `accept );

        return;
    }

}
```

Each dialog that is opened gets its own block enclosed in curly braces ("{..}"). In each block, the status of each widget that holds status information is restored (UI::ChangeWidget()).

Then there is a chance to make a screen shot; the macro recorder automatically adds a UI::MakeScreenShot() statement at the appropriate place, assigning generic file names that end in numbers. This statement is commented out by default as indicated by leading double slashes ("//") - this makes it simple to enable it if desired.

If the user explicitly hits the [PrintScreen] key to make a screen shot, another UI::MakeScreenShot() call (this time not commented out) will be added with the exact file name the user entered at the file selection box. The idea is to give the user a chance to assign more descriptive names to the screen shots.

After that, UI::FakeUserInput() simulates the same input the user had done while recording the macro. Usually, this is activation of a push button (like `accept as seen so many times above), but it can also be changing a selection box like in the language selection at the start of the macro (for insiders: if the widget has `opt(`notify) set).

The last line is a "return" statement that returns control flow from the macro block to the application that is being executed.

When the next dialog is executed (for insiders: when UI::UserInput() or related are called), the next macro block is executed.

So not only is the mechanism very generic, the code that is produced is also plain YCP code that is readable and that can easily be adapted if necessary.

Appendix A. UI Richtext

The RichText widget in the Qt UI currently supports the tags listed below. Note that not all of them will make sense for use within YaST2 (Most notably all those which refer to files)

- <qt>...</qt> - A Qt rich text document. It understands the following attributes
- title - the caption of the document. This attribute is easily accessible with [25]QTextView::documentTitle()
- type - The type of the document. The default type is page . It indicates that the document is displayed in a page of its own. Another style is detail. It can be used to explain certain expressions more detailed in a few sentences. The QTextBrowser will then keep the current page and display the new document in a small popup similar to QWhatsThis. Note that links will not work in documents with <qt type="detail" >...</qt>
- bgcolor - The background color, for example bgcolor="yellow" or bgcolor="#0000FF"
- background - The background pixmap, for example background="granit.xpm". The pixmap name will be resolved by a [26]QMimeSourceFactory().
- text - The default text color, for example text="red"
- link - The link color, for example link="green"

- <a>...</a> - An anchor or link. The reference target is defined in the href attribute of the tag as in \c<a href="target.qml">...</a>. You can also specify an additional anchor within the specified target document, for example <a href="target.qml#123">...</a>. If a is meant to be an anchor, the reference source is given in the name attribute.
- <font>...</font> - Customizes the font size and text color. The tag understands two attributes:

- color - the text color, for example color="red" or color="#FF0000".
- size - the logical size of the font. Logical sizes 1 to 7 are supported. The value may either be absolute, for example size=3, or relative. In the latter case, the sizes are simply added.

- <em>...</em> - Emphasized. As default, this is the same as <i>...</i> (Italic)
- <strong>...</strong> - Strong. As default, this is the same as <bold>...</bold> (bold)
- <big>...</big> - A larger font size.
- <small>...</small> - A smaller font size.
- <code>...</code> - Indicates Code. As default, this is the same as <tt>...</tt> (typewriter)
- <pre>...</pre> - For larger junks of code. Whitespaces in the contents are preserved.
- <large>...</large> - Large font size.
- <b>...</b> - Bold font style.
- <h1>...</h1> - A top-level heading.
- <h2>...</h2> - A sub-level heading.
- <h3>...</h3> - A sub-sub-level heading.
- <p>...</p> - A paragraph.
- <center>...</center> - A centered paragraph.
- <blockquote>...</blockquote> - An indented paragraph, useful for quotes.
- <multicol cols=n >...</multicol> - Multicol display with n columns
- <twocolumn>...</twocolumn> - Two-column display.
- <ul>...</ul> - An un-ordered list. You can also pass a type argument to define the bullet style. The default is type=disc, other types are circle and square.
- <ol>...</ol> - An ordered list. You can also pass a type argument to define the enumeration label style. The default is type="1", other types are "a" and "A".
- <li>...</li> - A list item.
- <img> - An image. The image name for the mime source factory is given in the source attribute, for example <img source="qt.xpm"/> The image tag also understands the attributes width and height that determine the size of the image. If the pixmap does not fit to the specified size, it will be scaled automatically.
- <br> - A line break
- <hr> - A horizonal line

Part I. YCP Reference

Table of Contents

WFM Builtins

Name

SCROpen -- Create a new scr instance.

SCROpen

```
integer SCROpen (name, check_version);
string name ;
boolean check_version ;
```

Parameters

string *name*	a valid y2component name
boolean *check_version*	determines whether the SuSE Version should be checked

Return

integer	On error a negative value is returned.

Description

Create a new scr instance. The name must be a valid y2component name (e.g. "scr", "chroot=/mnt:scr"). The component is created immediately. The parameter check_version determines whether the SuSE Version should be checked. On error a negative value is returned.

Name

SCRClose -- Close a scr instance.

SCRClose

```
void SCRClose (handle);
integer handle ;
```

Parameters

integer *handle*	SCR handle

Return

void

Name

SCRGetName -- Get the name of a scr instance.

SCRGetName

```
string SCRGetName (handle);
integer handle ;
```

Parameters

integer *handle*	SCR handle

Return

string	Name

Name

SCRSetDefault -- Set's the default scr instance.

SCRSetDefault

```
void SCRSetDefault (handle);
integer handle ;
```

Parameters

integer *handle*	SCR handle

Return

void

Name

SCRGetDefault -- Get's the default scr instance.

SCRGetDefault

```
integer SCRGetDefault ();
```

Return

integer	Default SCR handle

Name

Args -- Returns the arguments with which the module was called.

Args

```
list Args ();
```

Return

list	List of arguments

Description

The result is a list whose arguments are the module's arguments. If the module was called with *`CallFunction("my_mod", [17,true])`*, *`Args()`* will return *`[ 17, true ]`*.

Name

GetLanguage -- Returns the current language code (without modifiers !)

GetLanguage

```
string GetLanguage ();
```

Return

string	Language

Name

GetEncoding -- Returns the current encoding code

GetEncoding

```
string GetEncoding ();
```

Return

string	Encoding

Name

GetEnvironmentEncoding -- Returns the encoding code of the environment where yast is started

GetEnvironmentEncoding

```
string GetEnvironmentEncoding ();
```

Return

string	encoding code of the environment

Name

SetLanguage -- Selects the language for translate()

SetLanguage

```
string SetLanguage (language, encoding);
string language ;
string encoding ;
```

Parameters

string *language*

Optional Arguments

string *encoding*

Return

string	proposed encoding have fun

Description

The "<proposed encoding>" is the output of 'nl_langinfo (CODESET)' and only given if SetLanguage() is called with a single argument.

Usage

```
SetLanguage("de_DE", "UTF-8") -> ""
SetLanguage("de_DE@euro") -> "ISO-8859-15"
```

Name

Read -- Special interface to the system agent. Not for general use.

Read

```
any Read (path, options);
path path ;
any options ;
```

Parameters

path *path*	Path

Optional Arguments

any *options*

Return

any

Name

Write -- Special interface to the system agent. Not for general use.

Write

```
boolean Write (path, options);
path path ;
any options ;
```

Parameters

path *path*	Path

Optional Arguments

any *options*

Return

boolean

Name

Execute -- Special interface to the system agent. Not for general use.

Execute

```
any Execute (path, options);
path path ;
any options ;
```

Parameters

path *path*	Path

Optional Arguments

any *options*

Return

any

Name

call -- Executes a YCP client or a Y2 client component.

call

```
any call (name, arguments);
string name ;
list arguments ;
```

Parameters

string *name*	client name
list *arguments*	list of arguments

Return

any

Description

This implies * that the called YCP code has full access to all module status in the currently running YaST.

The modulename is temporarily changed to the name of the called script or a component.

In the example, WFM looks for the file YAST2HOME/clients/inst_mouse.ycp and executes it. If the client is not found, a Y2 client component is tried to be created.

Usage

```
call ("inst_mouse", [true, false]) -> ....
```

YCP Byteblock Builtins

Name

tobyteblock -- Converts a value to a byteblock.

tobyteblock

```
byteblock tobyteblock (VALUE);
any VALUE ;
```

Parameters

any *VALUE*

Return

byteblock

Name

size -- Returns a size of a byteblock in bytes.

size

```
integer **size** (VALUE);
byteblock VALUE ;
```

Parameters

byteblock *VALUE*

Return

integer

YCP Float Builtins

Name

tostring -- Converts a floating point number to a string

tostring

```
string tostring (FLOAT, PRECISION);
float FLOAT ;
integer PRECISION ;
```

Parameters

float *FLOAT*
integer *PRECISION*

Return

string

Usage

```
tostring (0.12345, 4) -> 0.1235
```

Name

tofloat -- Converts a value to a floating point number.

tofloat

```
float **tofloat** (VALUE);
any VALUE ;
```

Parameters

any *VALUE*

Return

float

Usage

```
tofloat (4) -> 4.0
tofloat ("42") -> 42.0
tofloat ("3.14") -> 3.14
```

YCP Integer Builtins

Name

tointeger -- Converts a value to an integer.

tointeger

```
integer tointeger (VALUE);
any VALUE ;
```

Parameters

any *VALUE*

Return

integer

Usage

```
tointeger (4.03) -> 4
tointeger ("42") -> 42
tointeger ("0x42") -> 66
tointeger ("042") -> 34
```

YCP List Builtins

Name

find -- Search for a certain element in a list

find

```
any find (VAR, LIST, EXPR);
any VAR ;
list LIST ;
block EXPR ;
```

Parameters

any *VAR*
list *LIST*
block *EXPR*

Return

any	Returns nil, if nothing is found.

Description

Searches for a certain item in the list. It applies the expression *EXPR* to each element in the list and returns the first element the makes the expression evaluate to true, if *VAR* is bound to that element.

Usage

```
find (integer n, [3,5,6,4], ``(n >= 5)) -> 5
```

Name

prepend -- Prepend a list with a new element

prepend

```
list **prepend** (ELEMENT, LIST);
any ELEMENT ;
list LIST ;
```

Parameters

any *ELEMENT*	Element to prepend
list *LIST*	List

Return

list

Description

Creates a new list that is identical to the list *LIST* but has the value *ELEMENT* prepended as additional element.

Usage

```
prepend ([1, 4], 8) -> [8, 1, 4]
```

Name

contains -- Check if a list contains an element

contains

```
boolean contains (LIST, ELEMENT);
list LIST ;
any ELEMENT ;
```

Parameters

list *LIST*	List
any *ELEMENT*	Element

Return

boolean	True if element is in the list.

Description

Determines, if a certain value *ELEMENT* is contained in a list *LIST*.

Usage

```
contains ([1, 2, 5], 2) -> true
```

Name

setcontains -- Check if a sorted list contains an element

setcontains

```
boolean setcontains (LIST, ELEMENT);
list LIST ;
any ELEMENT ;
```

Parameters

list *LIST*	List
any *ELEMENT*	Element

Return

boolean	True if element is in the list.

Description

Determines, if a certain value *ELEMENT* is contained in a list *LIST*, but assumes that *LIST* is sorted. If *LIST* is not sorted, the result is undefined.

Usage

```
setcontains ([1, 2, 5], 2) -> true
```

Name

union -- Union of lists

union

```
list union (LIST1, LIST2);
list LIST1 ;
list LIST2 ;
```

Parameters

list *`LIST1`*	First List
list *`LIST2`*	Second List

Return

list

Description

Interprets two lists as sets and returns a new list that has all elements of the first list and all of the second list. Identical elements are dropped. The order of the elements in the new list is preserved. Elements of *`l1`* are prior to elements from *`l2`*.

WARNING: quadratic complexity so far

Usage

```
union ([1, 2], [3, 4]) -> [1, 2, 3, 4]
union ([1, 2, 3], [2, 3, 4]) -> [1, 2, 3, 4]
```

Name

merge -- Merge two lists into one

merge

```
list merge (LIST1, LIST2);
list LIST1 ;
list LIST2 ;
```

Parameters

list *LIST1*	First List
list *LIST2*	Second List

Return

list

Usage

```
merge ([1, 2], [3, 4]) -> [1, 2, 3, 4]
merge ([1, 2, 3], [2, 3, 4]) -> [1, 2, 3, 2, 3, 4]
```

Name

filter -- Filter a List

filter

```
list filter (VAR, LIST, EXPR);
any VAR ;
list LIST ;
block<boolean> EXPR ;
```

Parameters

any *VAR*	Variable
list *LIST*	List to be filtered
block<boolean> *EXPR*	Block

Return

list

Description

For each element of the list `LIST` the expression *`expr`* is executed in a new context, where the variable *`VAR`* is assigned to that value. If the expression evaluates to true under this circumstances, the value is appended to the result list.

Usage

```
filter (integer v, [1, 2, 3, 5], { return (v > 2); }) -> [3, 5]
```

Name

maplist -- Maps an operation onto all elements of a list and thus creates a new list.

maplist

```
list<any> maplist (VAR, LIST, EXPR);
any VAR ;
list<any> LIST ;
block EXPR ;
```

Parameters

any *VAR*
list<any> *LIST*
block *EXPR*

Return

list<any>

Description

For each element of the list *LIST* the expression *EXPR* is evaluated in a new context, where the variable *VAR* is assigned to that value. The result is the list of those evaluations.

Usage

```
maplist (integer v, [1, 2, 3, 5], { return (v + 1); }) -> [2, 3, 4, 6]
```

Name

listmap -- Maps an operation onto all elements of a list and thus creates a map.

listmap

```
list listmap (VAR, LIST, EXPR);
any VAR ;
list LIST ;
block EXPR ;
```

Parameters

any *VAR*
list *LIST*
block *EXPR*

Return

list

Description

For each element *VAR* of the list *LIST* in the expression *EXPR* is evaluated in a new context. The result is the map of those evaluations.

The result of each evaluation *must* be a map with a single entry which will be added to the result map.

Usage

```
listmap (integer k, [1,2,3], { return $[k, "xy"]; })  -> $[ 1:"xy", 2:"xy" ]
listmap (integer k, [1,2,3], { any a = k+10;  any b = sformat ("x%1", k);   map ret = $[a,b];   return ret;
```

Name

flatten -- Flatten List

flatten

```
list flatten (LIST);
list<list> LIST ;
```

Parameters

list<list> *LIST*

Return

list

Description

Gets a list of lists *LIST* and creates a single list that is the concatenation of those lists in *LIST*.

Usage

```
flatten ([ [1, 2], [3, 4] ]) -> [1, 2, 3, 4]
```

Name

toset -- Sort list and remove duplicates

toset

```
list toset (LIST);
list LIST ;
```

Parameters

list *LIST*

Return

list	Sorted list with unique items

Description

Scans a list for duplicates, removes them and sorts the list.

Usage

```
toset ([1, 5, 3, 2, 3, true, false, true]) -> [false, true, 1, 2, 3, 5]
```

Name

sort -- Sort A List according to the YCP builtin predicate >

sort

```
list sort (LIST);
list LIST ;
```

Parameters

list *LIST*

Return

list	Sorted list

Description

Sort the list LIST according to the YCP builtin predicate >. Duplicates are not removed.

Usage

```
sort ([2, 1, true, 1]) -> [true, 1, 1, 2]
```

Name

sort -- Sort list using an expression

sort

```
list sort (x, y, LIST, EXPR);
any x ;
any y ;
list LIST ;
block EXPR ;
```

Parameters

any *x*
any *y*
list *LIST*
block *EXPR*

Return

list

Description

Sorts the list *LIST*. You have to specify an order on the list elements by naming formal variables *x* and *y* and specify an expression *EXPR* that evaluates to a boolean value depending on *x* and *y*. Return true if *x*>*y* to sort the list ascending.

The comparison must be an irreflexive one, that is ">" instead of ">=".

It is because we no longer use bubblesort (yuck) but *std::sort* which requires a <a href="http://www.sgi.com/tech/stl/StrictWeakOrdering.html">strict weak ordering</a>.

Usage

```
sort (integer x, integer y, [ 3,6,2,8 ], ``(x < y)) -> [ 2, 3, 6, 8 ]
```

Name

splitstring -- Split a string

splitstring

```
list<string> splitstring (STR, DELIM);
string STR ;
string DELIM ;
```

Parameters

string *STR*
string *DELIM*

Return

list<string>

Description

Splits *STR* into sub-strings at delimiter chars *DELIM*. the resulting pieces do not contain *DELIM*

If *STR* starts with *DELIM*, the first string in the result list is empty If *STR* ends with *DELIM*, the last string in the result list is empty. If *STR* does not contain *DELIM*, the result is a singleton list with *STR*.

Usage

```
splitstring ("/abc/dev/ghi", "/") -> ["", "abc", "dev", "ghi" ]
splitstring ("abc/dev/ghi/", "/") -> ["abc", "dev", "ghi", "" ]
splitstring ("abc/dev/ghi/", ".") -> ["abc/dev/ghi/" ]
splitstring ("text/with:different/separators", "/:") -> ["text", "with", "different", "separators"]
```

Name

change -- Change a list

change

```
list change (LIST, value);
list LIST ;
any value ;
```

Parameters

list *LIST*
any *value*

Return

list

Description

DO NOT use this yet. Its for a special requst, not for common use!!! changes the list LIST adds a new element

Usage

```
change ([1, 4], 8) -> [1, 4, 8]
```

Name

add -- Create a new list with a new element

add

```
list add (LIST, VAR);
list LIST ;
any VAR ;
```

Parameters

list *LIST*
any *VAR*

Return

list	The new list

Description

Creates a new list that is identical to the list *LIST* but has the value *VAR* appended as additional element.

Usage

```
add ([1, 4], 8) -> [1, 4, 8]
```

Name

size -- Return size of list

size

```
integer size (LIST);
list LIST ;
```

Parameters

list *LIST*

Return

integer	size of the list.

Description

Returns the number of elements of the list *LIST*

Name

remove -- Remove element from a list

remove

```
list remove (LIST, e);
list LIST ;
integer e ;
```

Parameters

list *LIST*	
integer *e*	element index

Return

list	Returns nil if the index is invalid.

Description

Remove the *i*'th value from a list. The first value has the index 0. The call remove ([1,2,3], 1) thus returns [1,3].

Usage

```
remove ([1, 2], 0) -> [2]
```

Name

select -- Selet a list element

select

```
any select (LIST, INDEX, DEFAULT);
list LIST ;
integer INDEX ;
any DEFAULT ;
```

Parameters

list *LIST*
integer *INDEX*
any *DEFAULT*

Return

any

Description

Gets the *INDEX*'th value of a list. The first value has the index 0. The call select([1,2,3], 1) thus returns 2. Returns *DEFAULT* if the index is invalid or if the found entry has a different type than the default value.

Usage

```
select ([1, 2], 22, 0) -> 0
select ([1, "two"], 0, "no") -> "no"
```

Name

foreach -- Process the content of a list

foreach

```
any foreach (VAR, LIST, EXPR);
any VAR ;
list LIST ;
block EXPR ;
```

Parameters

any *VAR*
list *LIST*
block *EXPR*

Return

any	return value of last execution of EXPR

Description

For each element of the list *LIST* the expression *EXPR* is executed in a new context, where the variable *VAR* is assigned to that value. The return value of the last execution of *EXPR* is the value of the *foreach* construct.

Usage

```
foreach (integer v, [1,2,3], { return v; }) -> 3
```

Name

tolist -- Converts a value to a list.

tolist

```
list tolist (VAR);
any VAR ;
```

Parameters

any *VAR*

Return

list

Description

If the value can't be converted to a list, nillist is returned.

Map Builtins

Name

haskey -- Check if map has a certain key

haskey

```
boolean haskey (MAP, KEY);
map MAP ;
any KEY ;
```

Parameters

map *MAP*
any *KEY*

Return

boolean

Name

filter -- Filter a Map

filter

```
map filter (KEY, VALUE, MAP, EXPR);
any KEY ;
any VALUE ;
map MAP ;
blocl EXPR ;
```

Parameters

any *KEY*
any *VALUE*
map *MAP*
blocl *EXPR*

Return

map

Description

For each key/value pair of the map *MAP* the expression *EXPR* is evaluated in a new context, where the variable *KEY* is assigned to the key and *VALUE* to the value of the pair. If the expression evaluates to true, the key/value pair is appended to the returned map.

Usage

```
filter (`k, `v, $[1:"a", 2:"b", 3:3, 5:5], { return (k == v); }) -> $[3:3, 5:5]
```

Name

mapmap -- Maps an operation onto all key/value pairs of a map

mapmap

```
map mapmap (KEY, VALUE, MAP, EXPR);
any KEY ;
any VALUE ;
map MAP ;
block EXPR ;
```

Parameters

any *KEY*
any *VALUE*
map *MAP*
block *EXPR*

Return

map

Description

Maps an operation onto all key/value pairs of the map *MAP* and thus creates a new map. For each key/value pair of the map *MAP* the expression *EXPR* is evaluated in a new context, where the variable *KEY* is assigned to the key and *VALUE* to the value of the pair. The result is the map of those evaluations.

The result of each evaluation *must* be a map with a single entry which will be added to the result map.

Usage

```
mapmap (`k, `v, $[1:"a", 2:"b"], { return ($[k+10 : v+"x"]); }) -> $[ 11:"ax", 12:"bx" ]
mapmap (`k, `v, $[1:"a", 2:"b"], { any a = k+10; any b = v+"x"; map ret = $[a:b]; return (ret); }) -> $[ 11
```

Name

maplist -- Maps an operation onto all elements key/value and create a list

maplist

```
list maplist (KEY, VALUE, MAP, EXPR);
any KEY ;
any VALUE ;
map MAP ;
block EXPR ;
```

Parameters

any *KEY*
any *VALUE*
map *MAP*
block *EXPR*

Return

list

Description

Maps an operation onto all elements key/value pairs of a map and thus creates a list.

Usage

```
maplist (`k, `v, $[1:"a", 2:"b"], { return [k+10, v+"x"]; }) -> [ [11, "ax"], [ 12, "bx" ] ]
```

Name

union -- Union of 2 maps

union

```
map **union** (MAP1, MAP2);
map MAP1 ;
map MAP2 ;
```

Parameters

map *MAP1*
map *MAP2*

Return

map

Description

Interprets two maps as sets and returns a new map that has all elements of the first map *MAP1*and all of the second map *MAP2*. If elements have identical keys, values from *MAP2* overwrite elements from *MAP1*.

Name

add -- Add a key/value pair to a map

add

```
map add (MAP, KEY, VALUE);
map MAP ;
any KEY ;
any VALUE ;
```

Parameters

map *MAP*
any *KEY*
any *VALUE*

Return

map

Description

Adds the key/value pair `k : v` to the map *MAP* and returns the newly Created map. If the key *KEY* exists in *KEY*, the old key/value pair is replaced with the new one.

Usage

```
add ($[a: 17, b: 11], `b, nil) -> $[a:17, b:nil].
```

Name

change -- Change element pair in a map

change

```
change (MAP, KEY, VALUE);
map MAP ;
any KEY ;
any VALUE ;
```

Parameters

map *MAP*
any *KEY*
any *VALUE*

Description

DO NOT use this yet. It's for a special requst, not for common use!!!

Adds the key/value pair *KEY : VALUE* to the map *MAP* and returns the map. *MAP is* modified. If the key *KEY* exists in *KEY*, the old key/value pair is replaced with the new one.

Usage

```
change ($[.a: 17, .b: 11], .b, nil) -> $[.a:17, .b:nil].
```

Name

size -- Size of a map

size

```
integer size (MAP);
map MAP ;
```

Parameters

map *MAP*

Return

integer

Name

foreach -- Process the content of a map

foreach

```
map foreach (KEY, VALUE, MAP, EXPR);
any KEY ;
any VALUE ;
map MAP ;
any EXPR ;
```

Parameters

any *KEY*
any *VALUE*
map *MAP*
any *EXPR*

Return

map

Description

For each key:value pair of the map *MAP* the expression *EXPR* is executed in a new context, where the variables *KEY* is bound to the key and *VALUE* is bound to the value. The return value of the last execution of exp is the value of the *foreach* construct.

Usage

```
foreach (integer k, integer v, $[1:1,2:4,3:9], { y2debug("v = %1", v); return v; }) -> 9
```

Name

tomap -- Converts a value to a map.

tomap

```
map tomap (VALUE);
any VALUE ;
```

Parameters

any *VALUE*

Return

map

Name

remove -- Remove key/value pair from a map

remove

```
map remove (MAP, KEY);
map MAP ;
any KEY ;
```

Parameters

map *MAP*
any *KEY*

Return

map

Usage

```
remove($[1:2], 0) -> nil
remove ($[1:2, 3:4], 1) -> $[3:4]
```

Name

lookup -- Select a map element

lookup

```
any lookup (MAP, KEY, DEFAULT);
map MAP ;
any KEY ;
any DEFAULT ;
```

Parameters

map *MAP*
any *KEY*
any *DEFAULT*

Return

any

Description

Gets the *KEY*'s value of a map. Returns *DEFAULT* if the key does not exist. Returns nil if the found entry has a different type than the default value.

Usage

```
lookup ($["a":42], "b", 0) -> 0
```

Miscellaneous YCP Builtins

Name

time -- Return the number of seconds since 1.1.1970.

time

```
integer time ();
```

Return

integer

Name

sleep -- Sleeps a number of milliseconds.

sleep

```
void sleep (MILLISECONDS);
integer MILLISECONDS ;
```

Parameters

integer *MILLISECONDS*	Time in milliseconds

Return

void

Name

random -- Random number generator.

random

```
integer random (MAX);
integer MAX ;
```

Parameters

integer *MAX*

Return

integer	Returns integer in the interval (0,MAX).

Name

srandom -- Initialize random number generator

srandom

```
integer srandom ();
```

Return

integer

Name

srandom -- Initialize random number generator.

srandom

```
void **srandom** (SEED);
integer SEED ;
```

Parameters

integer *SEED*

Return

void

Name

eval -- Evaluate a YCP value.

eval

eval ();

Description

See also the builtin ``, which is kind of the counterpart to eval.

Usage

```
eval (``(1+2)) -> 3
{ term a = ``add(); a = add(a, [1]); a = add(a, 4); return eval(a); } -> [1,4]
```

Name

sformat -- Format a String

sformat

```
string sformat (FORM, PAR1, PAR2, ...);
string FORM ;
any PAR1 ;
any PAR2 ;
any ... ;
```

Parameters

string *`FORM`*
any *`PAR1`*
any *`PAR2`*
any *`...`*

Return

string

Description

FORM is a string that may contains placeholders %1, %2, ... Each placeholder is substituted with the argument converted to string whose number is after the %. Only 1-9 are allowed by now. The percentage sign is donated with %%.

Usage

```
sformat ("%2 is greater %% than %1", 3, "five") -> "five is greater % than 3"
```

Name

y2debug -- Log a message to the y2log.

y2debug

```
void y2debug (FORMAT);
string FORMAT ;
```

Parameters

string *FORMAT*

Return

void

Usage

```
y2debug ("%1 is smaller than %2", 7, "13");
```

Name

y2milestone -- Log a milestone to the y2log.

y2milestone

```
void y2milestone (FORMAT);
string FORMAT ;
```

Parameters

string *FORMAT*

Return

void

Name

y2warning -- Log a warning to the y2log.

y2warning

```
void y2warning (FORMAT);
string FORMAT ;
```

Parameters

string *FORMAT*

Return

void

Name

y2error -- Log an error to the y2log.

y2error

```
void y2error (FORMAT);
string FORMAT ;
```

Parameters

string *FORMAT*

Return

void

Name

y2security -- Log a security message to the y2log.

y2security

```
void y2security (FORMAT);
string FORMAT ;
```

Parameters

string *FORMAT*

Return

void

Name

y2internal -- Log an internal message to the y2log.

y2internal

```
void **y2internal** (FORMAT);
string FORMAT ;
```

Parameters

string *FORMAT*

Return

void

YCP Path Builtins

Name

size -- Returns the number of path elements

size

```
integer **size** (PATH);
path PATH ;
```

Parameters

path *PATH*

Return

integer	Number of elements in the path

Usage

```
size (.hello.world) -> 2
size (.) -> 0
```

Name

add -- Add a path element to existing path

add

```
path add (PATH, STR);
path PATH ;
string STR ;
```

Parameters

path *PATH*
string *STR*

Return

path

Usage

```
add (.aaa, "anypath...\n\"") -> .aaa."anypath...\n\""
```

Name

topath -- Converts a value to a path.

topath

```
path topath (STR);
string STR ;
```

Parameters

string *STR*

Return

path

Usage

```
topath ("path") -> .path
topath (".some.path") -> .some.path
```

YCP String Builtins

Name

size -- Returns the number of characters of the string *s*

size

```
integer size (s);
string s ;
```

Parameters

string *s*	String

Return

integer	Size of string

Name

issubstring -- searches for a specific string within another string

issubstring

```
boolean issubstring (s, substring);
string s ;
string substring ;
```

Parameters

string *s*	String to be searched
string *substring*	Pattern to be searched for

Return

boolean

Description

Return true, if *substring* is a substring of *s*.

Usage

```
issubstring ("some text", "tex") -> true
```

Name

tohexstring -- Converts a integer to a hexadecimal string.

tohexstring

```
string tohexstring (number);
integer number ;
```

Parameters

integer *number*	Number

Return

string	Number in Hex

Usage

```
tohexstring (31) -> "0x1f"
```

Name

substring -- Return part of a string

substring

```
string substring (s, start, end);
string s ;
inetger start ;
integer end ;
```

Parameters

string *s*	Original String
inetger *start*	Start posistion

Optional Arguments

integer *end*	End Posistion

Return

string

Description

Returns the portion of string specified by the start and length parameters.

Usage

```
substring ("some text", 5) -> "text"
substring ("some text", 42) -> ""
substring ("some text", 5, 2) -> "te"
substring ("some text", 42, 2) -> ""
```

Name

substring -- Extract a substring

substring

```
string substring (STRING, START, LENGTH);
string STRING ;
integer START ;
integer LENGTH ;
```

Parameters

string *STRING*
integer *START*
integer *LENGTH*

Return

string

Description

Extract a substring of the string *STRING*, starting at *START* after the first one with length of at most *LENGTH*.

Usage

```
substring ("some text", 5, 2) -> "te"
substring ("some text", 42, 2) -> ""
```

Name

find -- Return position of a substring

find

```
integer find (STRING1, STRING2);
string STRING1 ;
string STRING2 ;
```

Parameters

string *STRING1*	String
string *STRING2*	Substring

Return

integer	If substring is not found find returns `-1'.

Description

The `find' function searches string for a specified substring (possibly a single character) and returns its starting position.

Returns the first position in *STRING1* where the string *STRING2* is contained in *STRING1*.

Usage

```
find ("abcdefghi", "efg") -> 4
find ("aaaaa", "z") -> -1
```

Name

tolower -- Make a string lowercase

tolower

```
string tolower (s);
string s ;
```

Parameters

string *s*	String

Return

string	String in lower case

Description

Returns string with all alphabetic characters converted to lowercase.

Usage

```
tolower ("aBcDeF") -> "abcdef"
```

Name

toupper -- Make a string uppercase

toupper

toupper ();

Description

Returns string with all alphabetic characters converted to uppercase.

Usage

```
tolower ("aBcDeF") -> "ABCDEF"
```

Name

toascii -- FIXME

toascii

```
string toascii (STRING);
string STRING ;
```

Parameters

string *STRING*

Return

string

Description

Returns a string that results from string *STRING* by copying each character that is below 0x7F (127).

Usage

```
toascii ("axx") -> "aB"
```

Name

deletechars -- Delete charachters from a string (FIXME)

deletechars

```
string deletechars (STRING, REMOVE);
string STRING ;
string REMOVE ;
```

Parameters

string *STRING*	
string *REMOVE*	Charachters to be removed

Return

string

Description

Returns a string that results from string *STRING* by removing all characters that occur in *REMOVE*.

Usage

```
deletechars ("a", "abcdefghijklmnopqrstuvwxyz") -> ""
```

Name

filterchars -- Filter charachters out of a String

filterchars

```
string filterchars (STRING, include);
string STRING ;
string include ;
```

Parameters

string *STRING*	
string *include*	String to be included

Return

string

Description

Returns a string that results from string *STRING* by removing all characters that do not occur in *include*.

Usage

```
filterchars ("a", "abcdefghijklmnopqrstuvwxyz") -> "ac"
```

Name

mergestring -- Join list elements with a string

mergestring

```
string mergestring (PIECES, GLUE);
list<string> PIECES ;
string GLUE ;
```

Parameters

list<string> *PIECES*	A List of strings
string *GLUE*	

Return

string

Description

Returns a string containing a string representation of all the list elements in the same order, with the glue string between each element.

List elements which are not of type strings are ignored.

Usage

```
mergestring (["", "abc", "dev", "ghi"], "/") -> "/abc/dev/ghi"
mergestring (["abc", "dev", "ghi", ""], "/") -> "abc/dev/ghi/"
mergestring ([1, "a", 3], ".") -> "a"
mergestring ([], ".") -> ""
mergestring (["abc", "dev", "ghi"], "") -> "abcdevghi"
mergestring (["abc", "dev", "ghi"], "123") -> "abc123dev123ghi"
```

Name

findfirstnotof -- Search string for first non matching chars

findfirstnotof

```
integer **findfirstnotof** (STRING, CHARS);
string STRING ;
string CHARS ;
```

Parameters

string *STRING*
string *CHARS*

Return

integer	the position of the first character in *STRING* that is not contained in *CHARS*.

Description

The `findfirstnotof' function searches the first element of string that doesn't match any character stored in chars and returns its position.

Usage

```
findfirstnotof ("abcdefghi", "abcefghi") -> 3
findfirstnotof ("aaaaa", "a") -> nil
```

Name

findfirstof -- Find position of first matching charachters in string

findfirstof

```
integer findfirstof (STRING, CHARS);
string STRING ;
string CHARS ;
```

Parameters

string *STRING*	
string *CHARS*	Charachters to find

Return

integer	the position of the first character in *STRING* that is contained in *CHARS*.

Description

The `findfirstof' function searches string for the first match of any character stored in chars and returns its position.

If no match is found findfirstof returns `nil'.

Usage

```
findfirstof ("abcdefghi", "cxdv") -> 2
findfirstof ("aaaaa", "z") -> nil
```

Name

findlastof -- Searches string for the last match

findlastof

```
integer findlastof (STRING, CHARS);
string STRING ;
string CHARS ;
```

Parameters

string *STRING*	String
string *CHARS*	Charachters to find

Return

integer	the position of the last character in *STRING* that is contained in *CHARS*.

Description

The `findlastof' function searches string for the last match of any character stored in chars and returns its position.

Usage

```
findlastof ("abcdecfghi", "cxdv") -> 5
findlastof ("aaaaa", "z") -> nil
```

Name

findlastnotof -- Searches the last element of string that doesn't match

findlastnotof

```
integer findlastnotof (STRING, CHARS);
string STRING ;
string CHARS ;
```

Parameters

string *STRING*	
string *CHARS*	Charachters

Return

integer	The position of the last character in *STRING* that is NOT contained in *CHARS*.

Description

The `findlastnotof' function searches the last element of string that doesn't match any character stored in chars and returns its position.

If no match is found the function returns `nil'.

Usage

```
findlastnotof( "abcdefghi", "abcefghi" ) -> 3 ('d')
findlastnotof("aaaaa", "a") -> nil
```

Name

regexpmatch -- Searches a string for a POSIX Extended Regular Expression match.

regexpmatch

```
boolean regexpmatch (INPUT, PATTERN);
string INPUT ;
string PATTERN ;
```

Parameters

string *INPUT*
string *PATTERN*

Return

boolean

Usage

```
regexpmatch ("aaabbbccc", "ab") -> true
regexpmatch ("aaabbbccc", "^ab") -> false
regexpmatch ("aaabbbccc", "ab+c") -> true
regexpmatch ("aaa(bbb)ccc", "\\(.*\\)") -> true
```

Name

regexppos -- Returns a pair with position and length of the first match.

regexppos

```
list regexppos (INPUT, PATTERN);
string INPUT ;
string PATTERN ;
```

Parameters

string *INPUT*
string *PATTERN*

Return

list

Description

If no match is found it returns an empty list.

Usage

```
regexppos ("abcd012efgh345", "[0-9]+") -> [4, 3]
("aaabbb", "[0-9]+") -> []
```

Name

regexpsub -- Regex Substitution

regexpsub

```
string regexpsub (INPUT, PATTERN, OUTPUT);
string INPUT ;
string PATTERN ;
string OUTPUT ;
```

Parameters

string *INPUT*
string *PATTERN*
string *OUTPUT*

Return

string

Description

Searches a string for a POSIX Extended Regular Expression match and returns *OUTPUT* with the matched subexpressions substituted or *nil* if no match was found.

Usage

```
regexpsub ("aaabbb", "(.*ab)", "s_\\1_e") -> "s_aaab_e"
regexpsub ("aaabbb", "(.*ba)", "s_\\1_e") -> nil
```

Name

regexptokenize -- Regex tokenize

regexptokenize

```
list regexptokenize (INPUT, PATTERN);
string INPUT ;
string PATTERN ;
```

Parameters

string *INPUT*
string *PATTERN*

Return

list

Description

Searches a string for a POSIX Extended Regular Expression match and returns a list of the matched subexpressions

If the pattern does not match, the list is empty. Otherwise the list contains then matchted subexpressions for each pair of parenthesize in pattern.

If the pattern is invalid, 'nil' is returned.

```
 Examples:
list e = regexptokenize ("aaabbBb", "(.*[A-Z]).*");
```

// e == ["aaabbB"]

list h = regexptokenize ("aaabbb", "(.*ab)(.*)");

// h == ["aaab", "bb"]

list h = regexptokenize ("aaabbb", "(.*ba).*");

// h == []

list h = regexptokenize ("aaabbb", "(.*ba).*(");

// h == nil

Name

tostring -- Converts a value to a string.

tostring

```
string tostring (VALUE);
any VALUE ;
```

Parameters

any *VALUE*

Return

string

Name

timestring -- Return time string

timestring

```
string timestring (FORMAT, TIME, UTC);
string FORMAT ;
integer TIME ;
boolean UTC ;
```

Parameters

string *FORMAT*
integer *TIME*
boolean *UTC*

Return

string

Description

Combination of standard libc functions gmtime or localtime and strftime.

Usage

```
timestring ("%F %T %Z", time (), false) -> "2004-08-24 14:55:05 CEST"
```

Name

crypt -- Encrypt a string

crypt

```
string crypt (UNENCRYPTED);
string UNENCRYPTED ;
```

Parameters

string *UNEN-CRYPTED*

Return

string

Usage

```
crypt ("readable") -> "Y2PEyAiaeaFy6"
```

Name

cryptmd5 -- Encrypt a string using md5

cryptmd5

```
string cryptmd5 (UNENCRYPTED);
string UNENCRYPTED ;
```

Parameters

string *UNEN-CRYPTED*

Return

string

Usage

```
cryptmd5 ("readable") -> "$1$BBtzrzzz$zc2vEB7XnA3Iq7pOgDsxD0"
```

Name

cryptbigcrypt -- Encrypt a string using bigcrypt

cryptbigcrypt

```
string cryptbigcrypt (UNENCRYPTED);
string UNENCRYPTED ;
```

Parameters

string *UNEN-CRYPTED*

Return

string

Usage

```
cryptbigcrypt ("readable") -> "d4brTQmcVbtNg"
```

Name

cryptblowfish -- Encrypt a string with blowfish

cryptblowfish

```
string **cryptblowfish** (UNENCRYPTED);
string UNENCRYPTED ;
```

Parameters

string *UNEN-
CRYPTED*

Return

string

Description

Encrypt the string *UNENCRYPTED* using blowfish password encryption. The password is not truncated.

Usage

```
cryptblowfish ("readable") -> "$2a$05$B3lAUExB.Bqpy8Pq0TpZt.s7EydrmxJRuhOZR04YG01ptwOUR147C"
```

Name

dgettext -- Translates the text using the given text domain

dgettext

```
string dgettext (textdomain, text);
string textdomain ;
string text ;
```

Parameters

string *textdomain*

string *text*

Return

string

Description

Translates the text using the given text domain into the current language.

This is a special case builtin not intended for general use. See _() instead.

Usage

```
dgettext ("base", "No") -> "Nie"
```

Name

dngettext -- Translates the text using a locale-aware plural form handling

dngettext

```
string dngettext (textdomain, singular, plural, value);
string textdomain ;
string singular ;
string plural ;
integer value ;
```

Parameters

string *textdomain*
string *singular*
string *plural*
integer *value*

Return

string

Description

Translates the text using a locale-aware plural form handling using the given textdomain.

The chosen form of the translation depend on the *value*.

This is a special case builtin not intended for general use. See _() instead.

Usage

```
dngettext ("base", "%1 File", "%1 Files", 2) -> "%1 soubory"
```

YCP Term Builtins

Name

size -- Returns the number of arguments of the term `t`.

size

```
integer size (TERM);
term TERM ;
```

Parameters

term *TERM*

Return

integer	Size of the term

Name

add -- Add value to term

add

```
term add (TERM, VALUE);
term TERM ;
any VALUE ;
```

Parameters

term *TERM*
any *VALUE*

Return

term

Description

Adds the value *VALUE* to the term *TERM* and returns the newly created term. As always in YCP, *TERM* is not modified.

Usage

```
add (sym (a), b) -> sym (a, b)
```

Name

symbolof -- Returns the symbol of the term *t*.

symbolof

```
symbol **symbolof** (TERM);
term TERM ;
```

Parameters

term *TERM*

Return

symbol

Usage

```
symbolof (`hrombuch (18, false)) -> `hrombuch
```

Name

select -- Select item from term

select

```
select (TERM, ITEM, DEFAULT);
term TERM ;
integer ITEM ;
any DEFAULT ;
```

Parameters

term *TERM*
integer *ITEM*
any *DEFAULT*

Description

Gets the *i*'th value of the term *t*. The first value has the index 0. The call *select ([1, 2, 3], 1)* thus returns 2. Returns the *default* if the index is invalid or the found value has a diffetent type that *default*.

Usage

```
select (`hirn (true, false), 33, true) -> true
```

Name

toterm -- Converts a value to a term.

toterm

```
term toterm (VALUE);
any VALUE ;
```

Parameters

any *VALUE*

Return

term

Name

remove -- Remove item from term

remove

```
term remove (TERM, i);
term TERM ;
integer i ;
```

Parameters

term *TERM*
integer *i*

Return

term

Usage

```
remove (`fun(1, 2), 1) -> `fun(2)
```

Name

argsof -- Returns the arguments of a term.

argsof

```
list argsof (TERM);
term TERM ;
```

Parameters

term *TERM*

Return

list

Usage

```
argsof (`fun(1, 2)) -> [1, 2]
```

Part II. User Interface Reference

Table of Contents

Event-related UI Builtin Functions

Name

UI::UserInput -- Waits for user input and returns a widget ID.

UI::UserInput

Description

UI::UserInput() waits for the user to do some input. Normally this means it waits until the user clicks on a push button.

Widgets that have the notify option set can also cause UserInput() to return - i.e. to resume the control flow in the YCP code with the next statement after UserInput().

As long as the user does not do any such action, UserInput() waits, i.e. execution of the YCP code stops in UserInput(). In particular, entering text in input fields (TextEntry widgets) or selecting an entry in a list (SelectionBox widget) does not make UserInput() continue unless the respective widget has the notify option set.

UserInput() returns the ID of the widget that caused it to return. This is usually a button ID. It does *not* return any text entered etc.; use UI::QueryWidget() to retrieve the contents of the dialog's widgets.

Such a widget ID can be of any valid YCP type, but using simple types like *symbol*, *string* or maybe *integer* is strongly recommended.

Although it is technically still possible, using complex data types like *map*, *list* or even *term* (which might even contain YCP code to be executed with *eval()*) is discouraged. Support for this may be dropped without notice in future versions.

Since it depends on exactly what types the YCP application developer choses for his widgets, UserInput()'s return type is *any*. You may safely use a variable of the actual type you are using (usually *symbol* or *string*).

Usage:

```
any widget_id = UI::UserInput();
```

Example:

```
// UserInput.ycp
//
// Example for common usage of UI::UserInput()

{
    // Build dialog with two input fields and three buttons.
    //
    // Output goes to the log file: ~/.y2log for normal users
    // or /var/log/YaST2/y2log for root.

    string name = "Tux";
    string addr = "Antarctica";

    UI::OpenDialog(
                   `VBox(
                         `TextEntry(`id(`name), "&Name:",    name ),
                         `TextEntry(`id(`addr), "&Address:", addr ),
                         `HBox(
                               `PushButton(`id(`ok     ), "&OK" ),
                               `PushButton(`id(`cancel ), "&Cancel" ),
                               `PushButton(`id(`help   ), "&Help"   )
                               )
                         )
                   );

    symbol widget_id = nil; // All widget IDs used here are symbols

    // Event loop
    repeat
    {
        widget_id = UI::UserInput();

        if ( widget_id == `ok )
        {
            // process "OK" button

            y2milestone( "OK button activated" );


            // Retrieve widget contents

            name = UI::QueryWidget(`id(`name ), `Value );
            addr = UI::QueryWidget(`id(`addr ), `Value );
        }
        else if ( widget_id == `cancel )
        {
            // process "Cancel" buttton
            // or window manager close button (this also returns `cancel)

            y2milestone( "Cancel button activated" );
        }
        else if ( widget_id == `help )
        {
            // process "Help" button

            y2milestone( "Help button activated" );
        }

        // No other "else" branch necessary: None of the TextEntry widget has
        // the `notify option set, so none of them can make UserInput() return.

    } until ( widget_id == `ok || widget_id == `cancel );



    // Close the dialog - but only after retrieving all information that may
    // still be stored only in its widgets: QueryWidget() works only for
    // widgets that are still on the screen!

    UI::CloseDialog();


    // Dump the values entered into the log file

    y2milestone( "Name: %1 Address: %2", name, addr );
}
```

Name

UI::PollInput -- Checks for pending user input. Does not wait. Returns a widget ID or *nil* if no input is available.

UI::PollInput

Description

PollInput() is very much like UserInput(), but it doesn't wait. It only checks if there is a user event pending - the user may have clicked on a button since the last call to PollInput() or UserInput().

If there is one, the ID of the widget (usually a button unless other widgets have the notify option set) is returned. If there is none, *nil* (the YCP value for "nothing", "invalid") is returned.

Use PollInput() to check if the user wishes to abort operations of long duration that are performed in a loop. Notice that PollInput() will result in a "busy wait", so don't simply use it everywhere instead of UserInput().

Notice there is also TimeoutUserInput() and WaitForEvent() that both accept a millisecond timeout argument.

Usage:

```
any widget_id = UI::PollInput();
```

Example:

```
// PollInput.ycp
//
// Example for common usage of UI::PollInput()

{
    // Build dialog with two labels and a "stop" button.

    integer count     = 0;
    integer count_max = 10000;

    UI::OpenDialog(
                   `VBox(
                         `Label( "Calculating..." ),
                         `Label(`id(`count ), sformat( "%1 of %2", count, count_max ) ),
                         `PushButton(`id(`stop), "&Stop" )
                         )
                  );

    any widget_id = nil;

    // Event loop

    repeat
    {
        widget_id = UI::PollInput();


        // Simulate heavy calculation

        sleep(200); // milliseconds

        // Update screen to show that the program is really busy
        count = count + 1;
        UI::ChangeWidget(`id(`count), `Value, sformat( "%1 of %2", count, count_max ) );
        UI::RecalcLayout(); // Might be necessary when the label becomes wider

    } until ( widget_id == `stop || count >= count_max );

    UI::CloseDialog();
}
```

Name

UI::TimeoutUserInput -- Waits for user input and returns a widget ID. Returns ID *`timeout* if no input is available for *timeout* milliseconds.

UI::TimeoutUserInput

Description

TimeoutUserInput() is very much like UserInput(), but it returns a predefined ID *`timeout* if no user input is available within the specified (millisecond) timeout.

This is useful if there is a reasonable default action that should be done in case of a timeout - for example, for popup messages that are not important enough to completely halt a longer operation forever.

User interface style hint: Use this with caution. It is perfectly OK to use timeouts for informational messages that are not critical in any way ("*Settings are written*", "*Rebooting the newly installed kernel*"), but definitely not if there are several alternatives the user can choose from. As a general rule of thumb, if a dialog contains just an "OK" button and nothing else, TimeoutUserInput() is appropriate. If there are more buttons, chances are that the default action will cause disaster for some users.

Remember, timeouts are nearly always a desperate means. They are always both too short and too long at the same time: Too short for users who know what message will come and too long for users who left to get some coffee while the machine is busy.

Another possible use of TimeoutUserInput() would be to periodically update the screen with data that keep changing (time etc.) while waiting for user input.

Usage:

```
any widget_id = UI::TimeoutUserInput( integer timeout_millisec );
```

Example

```
// TimeoutUserInput.ycp
//
// Example for common usage of UI::TimeoutUserInput()

{
    // Build dialog with two labels and an "OK" button.

    integer countdown_sec       = 30;
    integer interval_millisec   = 200;
    integer countdown           = countdown_sec * 1000 / interval_millisec;

    UI::OpenDialog(
                   `VBox(
                         `Label( "Rebooting Planet Earth..." ),
                         `Label(`id(`seconds ), sformat( "%1", countdown_sec ) ),
                         `PushButton(`id(`ok ), `opt(`default ), "&OK" )
                         )
                  );

    any id = nil;

    // Event loop

    repeat
    {
        id = UI::TimeoutUserInput( interval_millisec );

        if ( id == `timeout )
        {
            // Periodic screen update

            countdown = countdown - 1;
            integer seconds_left = countdown * interval_millisec / 1000;
            UI::ChangeWidget(`id(`seconds ), `Value, sformat( "%1", seconds_left ) );
        }

    } until ( id == `ok || countdown <= 0 );
    UI::CloseDialog();
}
```

Name

UI::WaitForEvent -- Waits for user input and returns an event map. Returns ID *`timeout* if no input is available for *timeout* milliseconds.

UI::WaitForEvent

Description

WaitForEvent() is an extended combination of UserInput() and TimeoutUserInput(): It waits until user input is available or until the (millisecond) timeout is expired. It returns an event map rather than just a simple ID.

In the case of timeout, it returns a map with a timeout event.

The timeout argument is optional. If it isn't specified, WaitForEvent() (like UserInput()) keeps waiting until user input is available.

Use WaitForEvent() for more fine-grained control of events. It is useful primarily to tell the difference between different types of events of the same widget - for example, if different actions should be performed upon selecting an item in a SelectionBox or a Table widget. Notice that you still need the notify option to get those events in the first place.

On the downside, using WaitForEvent() means accessing the ID that caused an event requires a map lookup.

Notice that you still need UI::QueryWidget() to get the contents of the widget that caused the event. In the general case you'll need to QueryWidget most widgets on-screen anyway so delivering that one value along with the event wouldn't help too much.

Important: Don't blindly rely on getting each and every individual event that you think should come. The UI keeps track of only one pending event (which is usually the last one that occured). If many events occur between individual WaitForEvent() calls, all but the last will be lost. Read the introduction for the answer why. It is relatively easy to programm defensively in a way that losing individual events doesn't matter: Also use QueryWidget() to get the status of all your widgets. Don't keep redundant information about widget status in your code. Ask them. Always.

Usage

```
map event = UI::WaitForEvent();
```

```
map event = UI::WaitForEvent( integer timeout_millisec );
```

Example

```
// WaitForEvent.ycp
//
// Example for common usage of UI::WaitForEvent()

{
    // Build dialog with a selection box and some buttons.
    //
    // Output goes to the log file: ~/.y2log for normal users
    // or /var/log/YaST2/y2log for root.

    integer timeout_millisec = 20 * 1000;

    UI::OpenDialog(
                   `VBox(
                         `SelectionBox(`id(`pizza ), `opt(`notify, `immediate ),
                                       "Select your Pi&zza:",
                                        [
                                         `item(`id(`napoli      ), "Napoli"            ),
                                         `item(`id(`funghi      ), "Funghi"            ),
                                         `item(`id(`salami      ), "Salami"              ),
                                         `item(`id(`prociutto   ), "Prosciutto"          ),
                                         `item(`id(`stagioni    ), "Quattro Stagioni"    ),
                                         `item(`id(`chef        ), "A la Chef", true    )
                                         ]
                                        ),
                         `HBox(
                               `PushButton(`id(`ok     ), "&OK" ),
                               `PushButton(`id(`cancel ), "&Cancel" ),
                               `HSpacing(),
                               `PushButton(`id(`details ), "&Details..." )
                               )
                         )
                   );

    map event = $[];
    any id    = nil;

    // Event loop

    repeat
    {
        event = UI::WaitForEvent( timeout_millisec );
        id    = event["ID"]:nil; // We'll need this often - cache it

        if ( id == `pizza )
        {
            if ( event["EventReason"]:nil == "Activated" )
            {
                // Handle pizza "activate" (double click or space pressed)

                y2milestone( "Pizza activated" );
                id = `details;  // Handle as if "Details" button were clicked

            }
            else if ( event["EventReason"]:nil == "SelectionChanged" )
            {
                // Handle pizza selection change

                y2milestone( "Pizza selected" );
            }
        }

        if ( id == `details )
        {
            y2milestone( "Show details" );
        }

        if ( id == `timeout )
        {
            // Handle timeout

            y2milestone( "Timeout detected by ID" );
        }

        if ( event["EventType"]:nil == "TimeoutEvent" ) // Equivalent
        {
            // Handle timeout

            y2milestone( "Timeout detected by event type" );

            // Open a popup dialog

            UI::OpenDialog( `VBox(
                                  `Label( "Not hungry?" ),
                                  `PushButton(`opt(`default ), "&OK" )
                                  )
                            );
            UI::TimeoutUserInput( 10 * 1000 ); // Automatically close after 10 seconds
            UI::CloseDialog();
        }


    } until ( id == `ok || id == `cancel );


    UI::CloseDialog();
}
```

UI builtin commands

Name

UI::SetModulename -- Set Module Name

UI::SetModulename

```
void SetModulename (module);
string module ;
```

Parameters

string *module*

Return

void

Usage

```
SetModulename( "inst_environment" )
```

Name

UI::GetModulename -- Get the name of a Module

UI::GetModulename

`string` **`GetModulename`** `();`

Return

string

Usage

```
GetModulename()
```

Name

UI::SetLanguage -- Set the language of the UI

UI::SetLanguage

```
void SetLanguage (lang, encoding);
string lang ;
string encoding ;
```

Parameters

string *lang*	Language selected by user

Optional Arguments

string *encoding*

Return

void

Usage

```
SetLanguage( "de_DE@euro" )
SetLanguage( "en_GB" )
```

Name

UI::GetProductName -- Get Product Name

UI::GetProductName

`string **GetProductName** ();`

Return

string	Product Name

Description

Returns the current product name ("SuSE Linux", "United Linux", etc.) for display in dialogs. This can be set with SetProductName().

Note: In help texts in RichText widgets, a predefined macro &amp;product; can be used for the same purpose.

Usage

```
sformat( "Welcome to %1", GetProductName() );
```

Name

UI::SetProductName -- set Product Name

UI::SetProductName

```
void SetProductName (prod);
string prod ;
```

Parameters

string *prod*

Return

void

Description

Set the current product name ("SuSE Linux", "United Linux", etc.) for display in dialogs and in RichText widgets (for help text) with the RichText &amp;product; macro.

This product name should be concise and meaningful to the user and not cluttered with detailed version information. Don't use something like "SuSE Linux 12.3-i786 Professional". Use something like "SuSE Linux" instead.

Usage

```
SetProductName( "SuSE HyperWall" );
```

Name

UI::SetConsoleFont -- Set Console Font

UI::SetConsoleFont

```
void SetConsoleFont (console_magic, font, screen_map, unicode_map,
encoding);
string console_magic ;
string font ;
string screen_map ;
string unicode_map ;
string encoding ;
```

Parameters

string *console_magic*

string *font*

string *screen_map*

string *unicode_map*

string *encoding*

Return

void

Usage

```
SetConsoleFont( "( K", "lat2u-16.psf", "latin2u.scrnmap", "lat2u.uni", "latin1" )
```

Name

UI::SetKeyboard -- Set Keyboard

UI::SetKeyboard

`void` **`SetKeyboard`** `();`

Return

void

Usage

```
SetKeyboard( )
```

Name

UI::GetLanguage -- Get Language

UI::GetLanguage

```
string GetLanguage (strip_encoding);
boolean strip_encoding ;
```

Parameters

boolean
strip_encoding

Return

string

Description

Retrieves the current language setting from of the user interface. Since YaST2 is a client server architecture, we distinguish between the language setting of the user interface and that of the configuration modules. If the module or the translator wants to know which language the user currently uses, it can call *GetLanguage*. The return value is an ISO language code, such as "de" or "de_DE".

If "strip_encoding" is set to "true", all encoding or similar information is cut off, i.e. everything from the first "." or "@" on. Otherwise the current contents of the "LANG" environment variable is returned (which very likely ends with ".UTF-8" since this is the encoding YaST2 uses internally).

Name

UI::UserInput -- User Input

UI::UserInput

```
any UserInput ();
```

Return

any

Description

Waits for the user to click some button, close the window or activate some widget that has the `` `notify `` option set. The return value is the id of the widget that has been selected or `` `cancel `` if the user selected the implicit cancel button (for example he closes the window).

Name

UI::PollInput -- Poll Input

UI::PollInput

`any PollInput ();`

Return

any

Description

Doesn't wait but just looks if the user has clicked some button, has closed the window or has activated some widget that has the *`notify* option set. Returns the id of the widget that has been selected or *`cancel* if the user selected the implicite cancel button (for example he closes the window). Returns nil if no user input has occured.

Name

UI::TimeoutUserInput -- User Input with Timeout

UI::TimeoutUserInput

```
any TimeoutUserInput (timeout_millisec);
integer timeout_millisec ;
```

Parameters

integer
timeout_milli sec

Return

any

Description

Waits for the user to click some button, close the window or activate some widget that has the `` `notify `` option set or until the specified timeout is expired. The return value is the id of the widget that has been selected or `` `cancel `` if the user selected the implicit cancel button (for example he closes the window). Upon timeout, `` `timeout `` is returned.

Name

UI::WaitForEvent -- Wait for Event

UI::WaitForEvent

```
map WaitForEvent ();
```

Optional Arguments

timeout_millisec

Return

map

Description

Extended event handling - very much like UserInput(), but returns much more detailed information about the event that occured in a map.

Name

UI::OpenDialog -- Opens a new dialog.

UI::OpenDialog

```
boolean OpenDialog (widget);
term widget ;
```

Parameters

term *widget*

Return

boolean	Returns true on success.

Description

Opens a new dialog. *widget* is a term representation of the widget being displayed.

See the widget documentation for details what widgets are available. All open dialogs are arranged in a stack. A newly opened dialog is put on top of the stack. All operations implicitely refer to the topmost dialog. The user can interact only with that dialog. The application does not terminate if the last dialog is closed.

Usage

```
OpenDialog( `Label( "Please wait..." ) )
```

Name

UI::OpenDialog -- Open a Dialog with options

UI::OpenDialog

```
boolean OpenDialog (options, widget);
term options ;
term widget ;
```

Parameters

term *options*
term *widget*

Return

boolean

Description

Same as the OpenDialog with one argument, but you can specify options with a term of the form `` `opt``.

The option `` `defaultsize`` makes the dialog be resized to the default size, for example for the Qt interface the -geometry option is honored and for ncurses the dialog fills the whole window.

The option `` `centered`` centers the dialog to the desktop. This has no effect for popup dialogs that are a child of a `defaultsize dialog that is currently visible.

The option `` `decorated`` add a window border around the dialog, which comes in handy if no window manager is running. This option may be ignored in non-graphical UIs.

`` `smallDecorations`` tells the window manager to use only minimal decorations - in particular, no title bar. This is useful for very small popups (like only a one line label and no button). Don't overuse this. This option is ignored for `defaultsize dialogs.

The option `` `warncolor`` displays the entire dialog in a bright warning color.

The option `` `infocolor`` is a less intrusive color.

Usage

```
OpenDialog( `opt( `defaultsize ), `Label( "Hi" ) )
```

Name

UI::CloseDialog() -- Close an open dialog

UI::CloseDialog()

`boolean` **`CloseDialog()`** `();`

Return

boolean	Returns true on success.

Description

Closes the most recently opened dialog. It is an error to call *`CloseDialog`* if no dialog is open.

Name

UI::ChangeWidget -- Change widget contents

UI::ChangeWidget

```
boolean ChangeWidget (widgetId, property, newValue);
symbol widgetId ;
symbol property ;
any newValue ;
```

Parameters

symbol *widgetId*	Can also be specified as `id(any widgetId)
symbol *property*	
any *newValue*	

Return

boolean	Returns true on success.

Description

Changes a property of a widget of the topmost dialog. *id* specified the widget to change, *property* specifies the property that should be changed, *newvalue* gives the new value.

Name

UI::QueryWidget -- Query Widget contents

UI::QueryWidget

```
any QueryWidget (widgetId, property);
symbol widgetId ;
symbol|term property ;
```

Parameters

symbol *widgetId*	Can also be specified as `id(any id)
symbol|term *property*	

Return

any

Description

Queries a property of a widget of the topmost dialog. For example in order to query the current text of a TextEntry with id `name, you write *`QueryWidget(`id(`name), `Value )`*. In some cases the propery can be given as term in order to further specify it. An example is *`QueryWidget( `id(`table ), `Item(17))`* for a table where you query a certain item.

Name

UI::ReplaceWidget --

UI::ReplaceWidget

```
boolean ReplaceWidget (id, newWidget);
symbol id ;
term newWidget ;
```

Parameters

symbol *id*
term *newWidget*

Return

boolean

Description

Replaces a complete widget (or widget subtree) with an other widget (or widget tree). You can only replace the widget contained in a *ReplacePoint*. As parameters to *ReplaceWidget* specify the id of the ReplacePoint and the new widget.

Name

UI::WizardCommand -- Run a wizard command

UI::WizardCommand

```
boolean WizardCommand (wizardCommand);
term wizardCommand ;
```

Parameters

term *wizard-Command*

Return

boolean	Returns true on success.

Description

Issue a command to a wizard widget with ID 'wizardId'. < *This builtin is not for general use. Use the Wizard.ycp module instead.*

Name

UI::SetFocus -- Set Focus to the specified widget

UI::SetFocus

```
boolean SetFocus (widgetId);
symbol widgetId ;
```

Parameters

symbol *widget-Id*

Return

boolean	Returns true on success (i.e. the widget accepted the focus).

Name

UI::BusyCursor -- Sets the mouse cursor to the busy curso

UI::BusyCursor

`void **BusyCursor** ();`

Return

void

Description

Sets the mouse cursor to the busy cursor, if the UI supports such a feature.

This should normally not be necessary. The UI handles mouse cursors itself: When input is possible (i.e. inside UserInput()), there is automatically a normal cursor, otherwise, there is the busy cursor. Override this at your own risk.

Name

UI::RedrawScreen -- Redraws the screen

UI::RedrawScreen

```
void RedrawScreen ();
```

Return

void

Description

Redraws the screen after it very likely has become garbled by some other output.

This should normally not be necessary: The (specific) UI redraws the screen automatically whenever required. Under rare circumstances, however, the screen might have changes due to circumstances beyond the UI's control: For text based UIs, for example, system commands that cause output to every tty might make this necessary. Call this in the YCP code after such a command.

Name

UI::NormalCursor -- Sets the mouse cursor to the normal cursor

UI::NormalCursor

```
void NormalCursor ();
```

Return

void

Description

Sets the mouse cursor to the normal cursor (after BusyCursor), if the UI supports such a feature.

This should normally not be necessary. The UI handles mouse cursors itself: When input is possible (i.e. inside UserInput()), there is automatically a normal cursor, otherwise, there is the busy cursor. Override this at your own risk.

Name

UI::MakeScreenShot -- Make Screen Shot

UI::MakeScreenShot

```
void MakeScreenShot (filename);
string filename ;
```

Parameters

string *filename*

Return

void

Description

Make a screen shot if the specific UI supports that. The Qt UI opens a file selection box if filename is empty.

Name

UI::DumpWidgetTree -- Debugging function

UI::DumpWidgetTree

```
void DumpWidgetTree ();
```

Return

void

Description

Debugging function: Dump the widget tree of the current dialog to the log file.

Name

UI::RecordMacro -- Record Macro into a file

UI::RecordMacro

```
void RecordMacro (macroFileName);
string macroFileName ;
```

Parameters

string *macroFileName*

Return

void

Name

UI::StopRecordingMacro -- Stop recording macro

UI::StopRecordingMacro

```
void StopRecordingMacro ();
```

Return

void

Description

Stop macro recording. This is only necessary if you don't wish to record everything until the program terminates.

Name

UI::PlayMacro -- Play a recorded macro

UI::PlayMacro

```
void PlayMacro (macroFileName);
string macroFileName ;
```

Parameters

string *macroFileName*

Return

void

Description

Execute everything in macro file "macroFileName". Any errors are sent to the log file only. The macro can be terminated only from within the macro file.

Name

UI::FakeUserInput -- Fake User Input

UI::FakeUserInput

`void` **`FakeUserInput`** `();`

Optional Arguments

any *`nextUserInput`*

Return

void

Description

Prepare a fake value for the next call to UserInput() - i.e. the next UserInput() will return exactly this value. This is only useful in connection with macros.

If called without a parameter, the next call to UserInput() will return "nil".

Name

UI::Glyph -- Return a special character (a 'glyph')

UI::Glyph

```
string Glyph (glyph);
symbol glyph ;
```

Parameters

symbol *glyph*

Return

string

Description

Return a special character (a 'glyph') according to the symbol specified.

Not all UIs may be capable of displaying every glyph; if a specific UI doesn't support it, a textual representation (probably in plain ASCII) will be returned.

This is also why there is only a limited number of predefined glyphs: An ASCII equivalent is required which is sometimes hard to find for some characters defined in Unicode / UTF-8.

Please note the value returned may consist of more than one character; for example, Glyph(`ArrowRight) may return something like "-&gt;".

If an unknown glyph symbol is specified, 'nil' is returned.

Name

UI::GetDisplayInfo -- Get Display Info

UI::GetDisplayInfo

```
map GetDisplayInfo ();
```

Return

map

Description

Get information about the current display and the UI's capabilities.

Name

UI::RecalcLayout -- Racalculate Layout

UI::RecalcLayout

```
void RecalcLayout ();
```

Return

void

Description

Recompute the layout of the current dialog.

This is a very expensive operation.

Use this after changing widget properties that might affect their size - like the a Label widget's value. Call this once (!) after changing all such widget properties.

Name

UI::PostponeShortcutCheck -- Postpone Shortcut Check

UI::PostponeShortcutCheck

`void **PostponeShortcutCheck** ();`

Return

void

Description

Postpone keyboard shortcut checking during multiple changes to a dialog.

Normally, keyboard shortcuts are checked automatically when a dialog is created or changed. This can lead to confusion, however, when multiple changes to a dialog (repeated ReplaceWidget() calls) cause unwanted intermediate states that may result in shortcut conflicts while the dialog is not final yet. Use this function to postpone this checking until all changes to the dialog are done and then explicitly check with *`CheckShortcuts()`*. Do this before the next call to *`UserInput()`* or *`PollInput()`* to make sure the dialog doesn't change "on the fly" while the user tries to use one of those shortcuts.

The next call to *`UserInput()`* or *`PollInput()`* will automatically perform that check if it hasn't happened yet, any an error will be issued into the log file.

Use only when really necessary. The automatic should do well in most cases.

The normal sequence looks like this:

```
PostponeShortcutChecks();
ReplaceWidget( ... );
ReplaceWidget( ... );
...
ReplaceWidget( ... );
CheckShortcuts();
...
UserInput();
```

Name

UI::CheckShortcuts -- Perform an explicit shortcut check after postponing shortcut checks.

UI::CheckShortcuts

`void` **`CheckShortcuts`** `();`

Return

void

Description

Perform an explicit shortcut check after postponing shortcut checks. Use this after calling *`PostponeShortcutCheck()`*.

The normal sequence looks like this:

```
PostponeShortcutChecks();
ReplaceWidget( ... );
ReplaceWidget( ... );
...
ReplaceWidget( ... );
CheckShortcuts();
...
UserInput();
```

Name

UI::WidgetExists -- Check whether or not a widget with the given ID currently exists

UI::WidgetExists

```
boolean WidgetExists (widgetId);
symbol widgetId ;
```

Parameters

symbol *widgetId*

Return

boolean

Description

Check whether or not a widget with the given ID currently exists in the current dialog. Use this to avoid errors in the log file before changing the properties of widgets that might or might not be there.

Name

UI::RunPkgSelection -- Initialize and run the PackageSelector widget

UI::RunPkgSelection

```
any RunPkgSelection (pkgSelId);
any pkgSelId ;
```

Parameters

any *pkgSelId*

Return

any	Returns `cancel if the user wishes to cancel his selections.

Description

Not to be used outside the package selection

Initialize and run the PackageSelector widget identified by 'pkgSelId'.

Black magic to everybody outside. ;-)

Name

UI::AskForExistingDirectory -- Open a directory selection box and prompt the user for an existing directory.

UI::AskForExistingDirectory

```
string AskForExistingDirectory (startDir, headline);
string startDir ;
string headline ;
```

Parameters

string *`startDir`*	is the initial directory that is displayed.
string *`headline`*	is an explanatory text for the directory selection box. Graphical UIs may omit that if no window manager is running.

Return

string	Returns the selected directory name or *nil* if the user canceled the operation.

Description

Open a directory selection box and prompt the user for an existing directory.

Name

UI::AskForExistingFile -- Open a file selection box and prompt the user for an existing file.

UI::AskForExistingFile

```
string AskForExistingFile (startWith, filter, headline);
string startWith ;
string filter ;
string headline ;
```

Parameters

string *startWith*	is the initial directory or file.
string *filter*	is one or more blank-separated file patterns, e.g. "*.png *.jpg"
string *headline*	is an explanatory text for the file selection box. Graphical UIs may omit that if no window manager is running.

Return

string	Returns the selected file name or *nil* if the user canceled the operation.

Description

Open a file selection box and prompt the user for an existing file.

Name

UI::AskForSaveFileName -- Open a file selection box and prompt the user for a file to save data to.

UI::AskForSaveFileName

```
string AskForSaveFileName (startWith, filter, headline);
string startWith ;
string filter ;
string headline ;
```

Parameters

string *startWith*	is the initial directory or file.
string *filter*	is one or more blank-separated file patterns, e.g. "*.png *.jpg"
string *headline*	is an explanatory text for the file selection box. Graphical UIs may omit that if no window manager is running.

Return

string	Returns the selected file name or *nil* if the user canceled the operation.

Description

Open a file selection box and prompt the user for a file to save data to. Automatically asks for confirmation if the user selects an existing file.

Name

UI::SetFunctionKeys -- Set the (default) function keys for a number of buttons.

UI::SetFunctionKeys

```
void SetFunctionKeys (fkeys);
map fkeys ;
```

Parameters

map *fkeys*

Return

void

Description

This function receives a map with button labels and the respective function key number that should be used if on other `opt( `key_F..) is specified.

Any keyboard shortcuts in those labels are silently ignored so this is safe to use even if the UI's internal shortcut manager rearranges shortcuts.

Each call to this function overwrites the data of any previous calls.

Usage

```
SetFunctionKeys( $[ "Back": 8, "Next": 10, ... ] );
```

Name

UI::WFM/SCR -- callback

UI::WFM/SCR

```
any WFM/SCR (expression);
block expression ;
```

Parameters

block *expression*	

Return

any

Description

This is used for a callback mechanism. The expression will be sent to the WFM interpreter and evaluated there. USE WITH CAUTION.

Name

UI::Recode -- Recode encoding of string from or to "UTF-8" encoding.

UI::Recode

```
any **Recode** (from, to, text);
string from ;
string to ;
string text ;
```

Parameters

string *from*
string *to*
string *text*

Return

any

Description

Recode encoding of string from or to "UTF-8" encoding. One of from/to must be "UTF-8", the other should be an iso encoding specifier (i.e. "ISO-8859-1" for western languages, "ISO-8859-2" for eastern languages, etc.)

Standard (mandatory) widgets

Name

AAA_All-Widgets -- Generic options for all widgets

AAA_All-Widgets

`AAA_All-Widgets` `();`

Options

`notify`
: Make UserInput() return on any action in this widget. Normally UserInput() returns only when a button is clicked; with this option on you can make it return for other events, too, e.g. when the user selects an item in a SelectionBox (if `opt( `notify) is set for that SelectionBox). Only widgets with this option set are affected.

`disabled`
: Set this widget insensitive, i.e. disable any user interaction.

: The widget will show this state by being greyed out (depending on the specific UI).

`hstretch`
: Make this widget stretchable in the horizontal dimension.

`vstretch`
: Make this widget stretchable in the vertical dimension.

`hvstretch`
: Make this widget stretchable in both dimensions.

`autoShortcut`
: Automatically choose a keyboard shortcut for this widget and don't complain in the log file about the missing shortcut. Don't use this regularly for all widgets - manually chosen keyboard shortcuts are almost always better than those automatically assigned. Refer to the style guide for details. This option is intended used for automatically generated data, e.g., RadioButtons for software selections that come from file or from some other data base.

`key_F1`
: (NCurses only) activate this widget with the F1 key

`key_F2`
: (NCurses only) activate this widget with the F2 key

`key_Fxx`
: (NCurses only) activate this widget with the Fxx key

`key_F24`
: (NCurses only) activate this widget with the F24 key

`key_none`
: (NCurses only) no function key for this widget

keyEvents	(NCurses only) Make UserInput() / WaitForEvent() return on keypresses within this widget. Exactly which keys trigger such a key event is UI specific. This is not for general use.

Properties

boolean *Enabled*	the current enabled/disabled state
boolean *Notify*	the current notify state (see also `opt( `notify))
string *Widget-Class*	the widget class of this widget (YLabel, YPushButton, ...)
string *DebugLa-bel*	a (possibly translated) text describing this widget for debugging
string *Dialog-DebugLabel*	a (possibly translated) text describing this dialog for debugging

Description

This is not a widget for general usage, this is just a placeholder for descriptions of options that all widgets have in common.

Use them for any widget whenever it makes sense.

Usage

```
        ---
```

Examples

```
        {
    // (Minimalistic) Demo for automatically generated shortcuts.
    //
    // See 'AutoShortcut2.ycp' for a more realistic example.
    //
    // Please note this is _not_ how this option is meant to be used:
    // It is intended for automatically generated data, not for fixed widgets.
    // If you know your widget label at this point, manually add a keyboard
    // shortcut; this will almost always be much better than anything what can
    // be automatically generated.
    //
    //
    // There shouldn't be any complaints about shortcuts in the log file when this is started.

    UI::OpenDialog(
               `VBox(
                     `RadioButtonGroup(
                                `Frame( "Software Selection",
                                        `HVSquash(
                                                  `VBox(
                                                        `Left(  `RadioButton(`opt(`autoShortcut),  "Minimum Syste
                                                        `Left(  `RadioButton(`opt(`autoShortcut),  "Minimum X11 S
                                                        `Left(  `RadioButton(`opt(`autoShortcut),  "Gnome System"
                                                        `Left(  `RadioButton(`opt(`autoShortcut),  "Default (KDE
                                                        `Left(  `RadioButton(`opt(`autoShortcut),  "Default + Off
                                                        `Left(  `RadioButton(`opt(`autoShortcut),  "Almost Everyt
                                                        )
                                                  )
                                        )
                                ),
                     `PushButton( "&OK" )
                     )
               );

    UI::UserInput();
    UI::CloseDialog();
}
```

```
        {
    // Demo for automatically generated shortcuts.
    //
    // This is a more realistic example - it points out how the `autoShortcut
    // option is intended to be used. See 'AutoShortcut1.ycp' for a simpler example.
    //
    // There shouldn't be any complaints about shortcuts in the log file when this is started.


    list sw_selections =
        [
         "Minimum System",
         "Minimum X11 System",
         "Gnome System",
         "Default (KDE)",
         "Office System (KDE Based)",
         "Almost Everything",
        ];

    term radio_box = `VBox();

    foreach ( `sel, sw_selections, ``{
        radio_box = add( radio_box, `Left( `RadioButton(`opt(`autoShortcut), sel ) ) );
    } );

    y2milestone( "radio_box: %1", radio_box );

    UI::OpenDialog(
               `VBox(
                     `RadioButtonGroup(
                                       `Frame( "Software Selection",
                                               `HVSquash( radio_box )
                                               )
                                       ),
                     `PushButton( "&OK" )
                     )
               );

    UI::UserInput();
    UI::CloseDialog();
}
```

Name

ReplacePoint -- Pseudo widget to replace parts of a dialog

ReplacePoint

```
ReplacePoint (child);
term child ;
```

Parameters

term *child*	the child widget

Description

A ReplacePoint can be used to dynamically change parts of a dialog. It contains one widget. This widget can be replaced by another widget by calling *`ReplaceWidget( `id(id), new-child)`*, where *`id`* is the the id of the new child widget of the replace point. The ReplacePoint widget itself has no further effect and no optical representation.

Usage

```
        `ReplacePoint( `id( `rp ), `Empty() )
```

Examples

```
          {
    UI::OpenDialog(
               `VBox(
                     `ReplacePoint(`id(`rp), `Label("This is a label")),
                     `PushButton(`id(`change), "Change")));
    UI::UserInput();
    UI::ReplaceWidget(`id(`rp), `PushButton("This is a PushButton"));
    UI::UserInput();
    UI::ReplaceWidget(`id(`rp), `CheckBox("This is a CheckBox"));
    UI::UserInput();
    UI::ReplaceWidget(`id(`rp), `HBox(`PushButton("Button1"), `PushButton("Button2")));
    UI::UserInput();
    UI::CloseDialog();
}
```

Name

Empty -- Stretchable space for layout

Empty, HStretch, VStretch, HVStretch

Empty ();
HStretch ();
VStretch ();
HVStretch ();

Description

These four widgets denote an empty place in the dialog. They differ in whether they are stretchable or not. *`Empty`* is not stretchable in either direction. It can be used in a *`` `ReplacePoint ``*, when currently no real widget should be displayed. *`HStretch`* and *`VStretch`* are stretchable horizontally or vertically resp., *`HVStretch`* is stretchable in both directions. You can use them to control the layout.

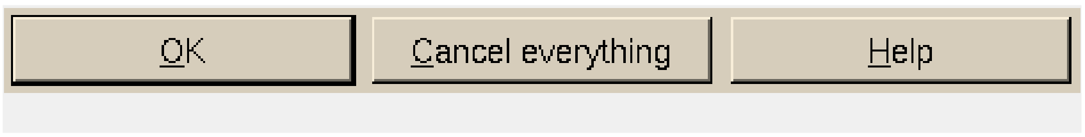
Usage

```
         `HStretch()
```

Examples

```
          {
  UI::OpenDialog(
             `VBox(
                   `Label("Some text goes here"),
                   `Label("This is some more text, that is quite long, as you can see."),
                   `HBox(
                         `PushButton("&OK"),
                         `HStretch()
                         )
                   )
             );
  any ret = UI::UserInput();
  UI::CloseDialog();
  return ret;
}
```

```
          {
    // Layout example:
    //
    // Build a dialog with three equal sized buttons.
    //
    // The equal `HWeight()s will make the buttons equal sized.
    // When resized larger, all buttons will retain their size.
    // Excess space will go to the HStretch() widgets between the
    // buttons, i.e. there will be empty space between the buttons.

    UI::OpenDialog(
               `HBox(
                     `HWeight(1, `PushButton( `opt(`default), "&OK" ) ),
                     `HStretch(),
                     `HWeight(1, `PushButton( "&Cancel everything" ) ),
                     `HStretch(),
                     `HWeight(1, `PushButton( "&Help"   ) )
                     )
               );

    UI::UserInput();
    UI::CloseDialog();
}
```


OK
Cancel everything
Help

Name

HSpacing -- Fixed size empty space for layout

HSpacing, VSpacing

```
HSpacing ();
VSpacing ();
```

Optional Arguments

integer|float *size*

Description

These widgets can be used to create empty space within a dialog to avoid widgets being cramped together - purely for aesthetical reasons. There is no functionality attached.

Do not try to use spacings with excessive sizes to create layouts! This is very likely to work for just one UI. Use spacings only to separate widgets from each other or from dialog borders. For other purposes, use *`HWeight* and *`VWeight* and describe the dialog logically rather than physically.

The *size* given is measured in units roughly equivalent to the size of a character in the respective UI. Fractional numbers can be used here, but text based UIs may choose to round the number as appropriate - even if this means simply ignoring a spacing when its size becomes zero.

If *size* is omitted, it defaults to 1. *HSpacing* will create a horizontal spacing with default width and zero height. *VSpacing* will create a vertical spacing with default height and zero width.

With options *hstretch* or *vstretch*, the spacing will at least take the amount of space specified with *size*, but it will be stretchable in the respective dimension. Thus, *`HSpacing( `opt(`hstretch )* is equivalent to *`HBox(`HSpacing( 0.5 ), `HSpacing(0.5))*

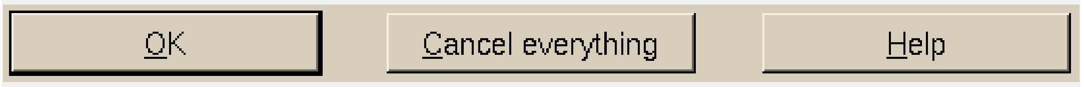
Usage

```
        `HSpacing( 0.3 )
```

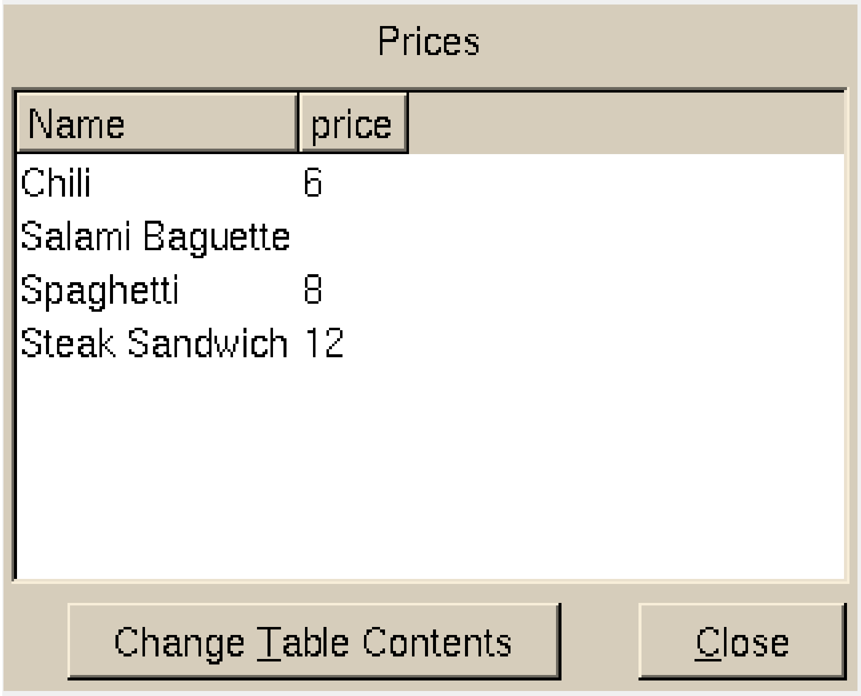
Examples

```
         {
    // Build dialog with one text entry field, 4 Beatles buttons and an OK button.
    UI::OpenDialog(
                `VBox(
                      `VSpacing(),
                      `HBox(
                            `Label("Name:"),
                            `TextEntry(`id(`name), "")
                            ),
                      `VSpacing(0.2),
                      `HBox(
                            `PushButton(`id(`john), "&John" ), `HSpacing(0.5),
                            `PushButton(`id(`paul), "&Paul" ), `HSpacing(3),
                            `PushButton(`id(`george), "&George"), `HSpacing(0.5),
                            `PushButton(`id(`ringo), "&Ringo" )
                            ),
                      `VSpacing(0.5),
                      `PushButton(`id(`ok), "&OK")
                      )
                );
    // Wait for user input.
    any button = nil;

    // Input loop that only the OK button will leave.
    // The 4 Beatles buttons will just propose a name.
    repeat
    {
        button = UI::UserInput();

        if      ( button == `john ) UI::ChangeWidget(`id(`name), `Value, "John Lennon");
        else if ( button == `paul ) UI::ChangeWidget(`id(`name), `Value, "Paul McCartney");
        else if ( button == `george ) UI::ChangeWidget(`id(`name), `Value, "George Harrison");
        else if ( button == `ringo ) UI::ChangeWidget(`id(`name), `Value, "Ringo Starr" );

    } until ( button == `ok );

    UI::CloseDialog();
}
```

```
        {
    // Layout example:
    //
    // Build a dialog with three equal sized buttons,
    // this time with some spacing in between.
    //
    // The equal `HWeight()s will make the buttons even sized.
    // When resized larger, all buttons will retain their size.
    // Excess space will go to the HSpacing() widgets between the
    // buttons, i.e. there will be empty space between the buttons.
    //
    // Notice the importance of `opt(`hstretch) for the `HSpacing()s
    // here: This is what makes the HSpacing()s grow. Otherwise, they
    // would retain a constant size, and the buttons would grow.

    UI::OpenDialog(
               `HBox(
                     `HWeight(1, `PushButton( `opt(`default), "&OK" ) ),
                     `HSpacing(`opt(`hstretch), 3),
                     `HWeight(1, `PushButton( "&Cancel everything" ) ),
                     `HSpacing(`opt(`hstretch), 3),
                     `HWeight(1, `PushButton( "&Help"   ) )
                     )
               );

    UI::UserInput();
    UI::CloseDialog();
}
```

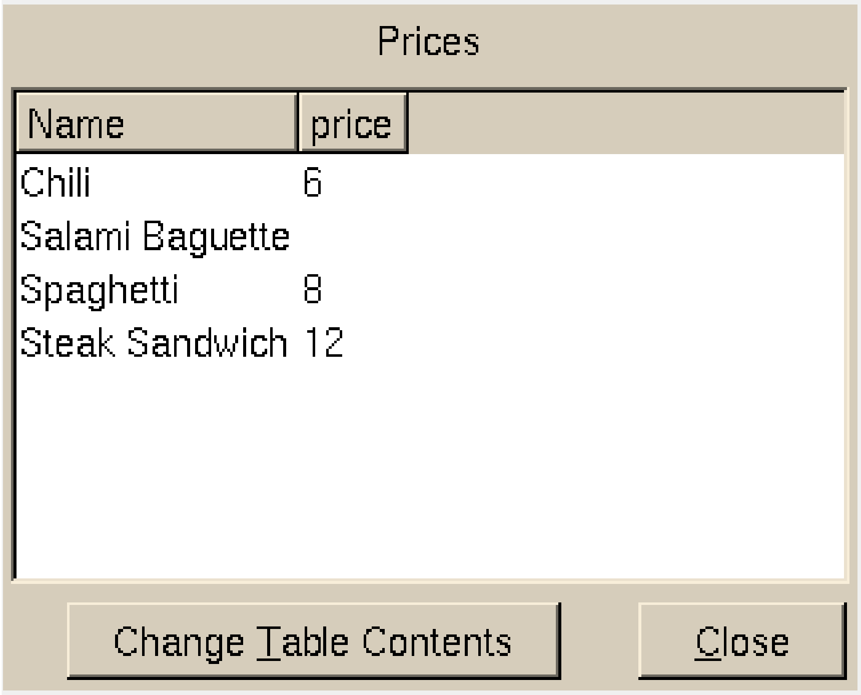


```
        {
    list itemlist1 =
        [
         `item(`id(3), "Spaghetti",   8),
         `item(`id(4), "Steak Sandwich",  12),
         `item(`id(1), "Chili",           6),
         `item(`id(2), "Salami Baguette", nil)
        ];

    list itemlist2 =
        [
         `item(`id(0), "Mercedes", 60000),
         `item(`id(1), "AUDI",         50000),
         `item(`id(2), "VW",         40000),
         `item(`id(3), "BMW",         60000),
         `item(`id(3), "Porsche", 80000)
        ];

    list itemslists = [ itemlist1, itemlist2 ];

    integer listnum = 0;

    UI::OpenDialog(
               `VBox(
                     `Label("Prices"),
                     `HSpacing(40), // make the table and thus the dialog wide enough
                     `HBox(
                           `VSpacing(10),
                              `Table(`id(`table), `header("Name", "price"), itemlist1)
                              ),
                        `HBox(
                              `HCenter(`PushButton(`id(`next), "Change &Table Contents")),
                              `PushButton(`id(`cancel), "&Close")
                              )
                        )
                  );

    while (UI::UserInput() != `cancel)
    {
        listnum = 1 - listnum;
        UI::ChangeWidget(`id(`table), `Items, select(itemslists, listnum, nil));
    }

    UI::CloseDialog();
}
```



```
      {
    list itemlist1 =
        [
         `item(`id(3), "Spaghetti",   8),
         `item(`id(4), "Steak Sandwich",  12),
         `item(`id(1), "Chili",           6),
         `item(`id(2), "Salami Baguette", nil)
        ];

    list itemlist2 =
        [
         `item(`id(0), "Mercedes", 60000),
         `item(`id(1), "AUDI",         50000),
         `item(`id(2), "VW",         40000),
         `item(`id(3), "BMW",          60000),
         `item(`id(3), "Porsche", 80000)
        ];

    list itemslists = [ itemlist1, itemlist2 ];

    integer listnum = 0;

    UI::OpenDialog(
                  `VBox(
                        `Label("Prices"),
                        `HSpacing(40), // make the table and thus the dialog wide enough
                        `HBox(
                              `VSpacing(10),
                              `Table(`id(`table), `header("Name", "price"), itemlist1)
                              ),
                        `HBox(
                              `HCenter(`PushButton(`id(`next), "Change &Table Contents")),
                              `PushButton(`id(`cancel), "&Close")
```

```
                            )
                        )
                    );

    while (UI::UserInput() != `cancel)
    {
        listnum = 1 - listnum;
        UI::ChangeWidget(`id(`table), `Items, select(itemslists, listnum, nil));
    }

    UI::CloseDialog();
}
```



Prices

Name	price
Chili	6
Salami Baguette	
Spaghetti	8
Steak Sandwich	12

Name

Left -- Layout alignment

Left, Right, Top, Bottom, HCenter, VCenter, HVCenter

```
Left (child, enabled);
term child ;
boolean enabled ;
Right (child, enabled);
term child ;
boolean enabled ;
Top (child, enabled);
term child ;
boolean enabled ;
Bottom (child, enabled);
term child ;
boolean enabled ;
HCenter (child, enabled);
term child ;
boolean enabled ;
VCenter (child, enabled);
term child ;
boolean enabled ;
HVCenter (child, enabled);
term child ;
boolean enabled ;
```

Parameters

term *child*	The contained child widget

Optional Arguments

boolean *enabled*	true if ...

Description

The Alignment widgets are used to control the layout of a dialog. They are useful in situations, where to a widget is assigned more space than it can use. For example if you have a VBox containing four CheckBoxes, the width of the VBox is determined by the CheckBox with the longest label. The other CheckBoxes are centered per default.

With *`Left(widget)* you tell widget that it should be layouted leftmost of the space that is available to it. *Right, Top* and *Bottom* are working accordingly. The other three widgets center their child widget horizontally, vertically or in both directions.

The important fact for all alignment widgets is, that they make their child widget *stretchable* in the dimension it is aligned.

Usage

```
        `Left( `CheckBox( "Crash every five minutes" ) )
```

Examples

```
          {
    UI::OpenDialog(
               `VBox(
                     `Label("This is a long label which makes space"),
                     `HBox(
                           `Label("A"),
                           `HCenter(`Label("B")),
                           `Label("C")
                           )
                     )
               );
    UI::UserInput();
    UI::CloseDialog();
}
```

```
          {
    UI::OpenDialog(
               `VBox(
                     `Label("This is a very long label that makes space"),
                     `HBox(
                           `PushButton("Normal"),
                           `HCenter(`PushButton("HCenter"))
                           )
                     )
               );
    UI::UserInput();
    UI::CloseDialog();
}
```

```
          {
    UI::OpenDialog(`opt(`defaultsize),
               `VBox(
                     `VCenter(`PushButton(`opt(`vstretch), "Button 1")),
                     `VCenter(`PushButton(`opt(`vstretch), "Button 2")),
                     `VCenter(`PushButton(`opt(`vstretch), "Button 3"))
                     )
               );
    UI::UserInput();
    UI::CloseDialog();
}
```

```
          {
    UI::OpenDialog(
               `VBox(
                     `PushButton("This is a very long button - it reserves extra space for the label."),
                     `HBox(
                           `PushButton(`opt(`hstretch), "Stretchable button"),
                           `ReplacePoint(`id(`rp), `Label("Label"))
                           )
                     )
               );
    UI::UserInput();

    UI::ReplaceWidget(`id(`rp), `Left(`Label("Left")));
    UI::UserInput();

    UI::ReplaceWidget(`id(`rp), `Right(`Label("Right")));
    UI::UserInput();

    UI::ReplaceWidget(`id(`rp), `HCenter(`Label("HCenter")));
    UI::UserInput();
}
```

Name

Frame -- Frame with label

Frame

```
Frame (label, child);
string label ;
term child ;
```

Parameters

string *label*	title to be displayed on the top left edge
term *child*	the contained child widget

Properties

string *Value*	the label text

Description

This widget draws a frame around its child and displays a title label within the top left edge of that frame. It is used to visually group widgets together. It is very common to use a frame like this around radio button groups.

Usage

```
        `Frame( `RadioButtonGroup( `id( rb ), `VBox( ... ) ) );
```

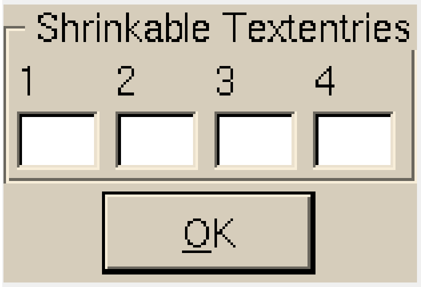
Examples

```
          {
    UI::OpenDialog(
               `VBox(
                     `Frame ( "Important",
                              `Label("Hello, World!")
                              ),
                     `PushButton("&OK")
                     )
               );
    UI::UserInput();
    UI::CloseDialog();
}
```

```
          {
    UI::OpenDialog( `VBox(
                      `Frame ( "CPU &Speed",
                               `RadioButtonGroup(
                                                  `VBox(
                                                        `Left(`RadioButton("Normal" )),
                                                        `Left(`RadioButton("Overclocked" )),
                                                        `Left(`RadioButton("Red Hot" )),
                                                        `Left(`RadioButton("Melting", true ))
                                                        )
                                                  )
```

```
                            ),
                    `PushButton("&OK")
                    )
              );
    UI::UserInput();
    UI::CloseDialog();
}
```

```
          {
    UI::OpenDialog(
              `VBox(
                   `Frame("Shrinkable Textentries",
                          `HBox(
                                `TextEntry(`opt(`shrinkable), "1"),
                                `TextEntry(`opt(`shrinkable), "2"),
                                `TextEntry(`opt(`shrinkable), "3"),
                                `TextEntry(`opt(`shrinkable), "4")
                                )
                          ),
                   `PushButton(`opt(`default), "&OK" )
                   )
              );
    UI::UserInput();
    UI::CloseDialog();
}
```



Name

HSquash -- Layout aid: Minimize widget to its nice size

HSquash, VSquash, HVSquash

```
HSquash (child);
term child ;
VSquash (child);
term child ;
HVSquash (child);
term child ;
```

Parameters

term `child`	the child widget

Description

The Squash widgets are used to control the layout. A `HSquash` widget makes its child widget *non-stretchable* in the horizontal dimension. A `VSquash` operates vertically, a `HVSquash` in both dimensions.

You can used this for example to reverse the effect of `` `Left `` making a widget stretchable. If you want to make a VBox containing for left aligned CheckBoxes, but want the VBox itself to be non-stretchable and centered, than you enclose each CheckBox with a `` `Left( .. ) `` and the whole VBox with a `HSquash( ... )`.

Usage

```
        HSquash( `TextEntry( "Name:" ) )
```

Examples

```
         {
    UI::OpenDialog(`opt(`defaultsize),
               `VBox(
                     `VCenter( // Makes the HSquash stretchable vertically
                              `HSquash( // Makes the VBox nonstretchable horizontally
                                       `VBox(
                                             `Left(`CheckBox("short")),
                                             `Left(`CheckBox("longer")),
                                             `Left(`CheckBox("even longer")),
                                             `Left(`CheckBox("yet even longer"))))),
                     `Left(`PushButton("bottom left"))
                     )
               );

    UI::UserInput();
    UI::CloseDialog();
}
```

Name

HWeight -- Control relative size of layouts

HWeight, VWeight

```
HWeight (weight, child);
integer weight ;
term child ;
VWeight (weight, child);
integer weight ;
term child ;
```

Parameters

integer *weight*	the new weight of the child widget
term *child*	the child widget

Description

This widget is used to control the layout. When a *HBox* or *VBox* widget decides how to devide remaining space amount two *stretchable* widgets, their weights are taken into account. This widget is used to change the weight of the child widget. Each widget has a vertical and a horizontal weight. You can change on or both of them. If you use *HVWeight*, the weight in both dimensions is set to the same value.

Note: No real widget is created (any more), just the weight value is passed to the child widget.

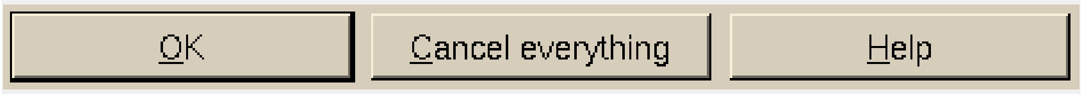
Usage

```
        `HWeight( 2, `SelectionBox( "Language" ) )
```

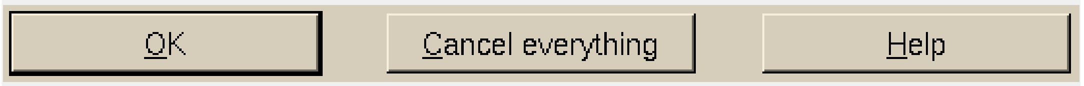
Examples

```
         {
    UI::OpenDialog(
               `HBox(
                     `HWeight(1, `PushButton("First Button (W: 50)")),
                     `PushButton("Small Button"),
                     `HWeight(1, `PushButton("Second Button (Weight 50 - this one determines the total width"
                     )
               );
    UI::UserInput();
    UI::CloseDialog();
}
```

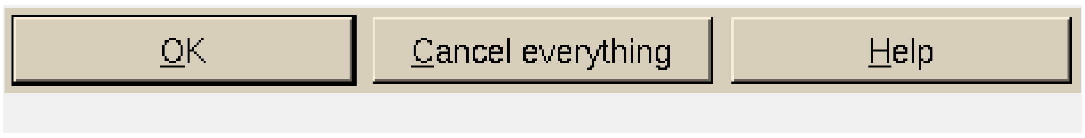
```
         {
    // Layout example:
    //
    // Build a dialog with three equal sized buttons.
    //
    // The equal `HWeight()s will make the buttons equal sized.
    // When resized larger, all buttons will retain their size.
    // Excess space will go to the HStretch() widgets between the
    // buttons, i.e. there will be empty space between the buttons.
    UI::OpenDialog(
               `HBox(
                     `HWeight(1, `PushButton( `opt(`default), "&OK" ) ),
                     `HStretch(),
                     `HWeight(1, `PushButton( "&Cancel everything" ) ),
                     `HStretch(),
                     `HWeight(1, `PushButton( "&Help"   ) )
                     )
               );

    UI::UserInput();
    UI::CloseDialog();
}
```



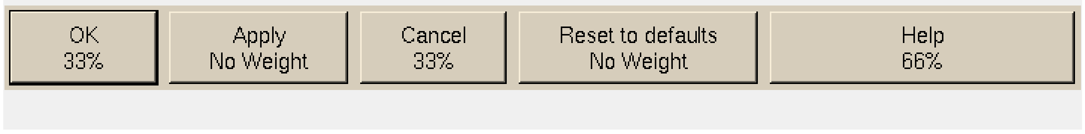
```
          {
    // Layout example:
    //
    // Build a dialog with three equal sized buttons,
    // this time with some spacing in between.
    //
    // The equal `HWeight()s will make the buttons even sized.
    // When resized larger, all buttons will retain their size.
    // Excess space will go to the HSpacing() widgets between the
    // buttons, i.e. there will be empty space between the buttons.
    //
    // Notice the importance of `opt(`hstretch) for the `HSpacing()s
    // here: This is what makes the HSpacing()s grow. Otherwise, they
    // would retain a constant size, and the buttons would grow.

    UI::OpenDialog(
               `HBox(
                     `HWeight(1, `PushButton( `opt(`default), "&OK" ) ),
                     `HSpacing(`opt(`hstretch), 3),
                     `HWeight(1, `PushButton( "&Cancel everything" ) ),
                     `HSpacing(`opt(`hstretch), 3),
                     `HWeight(1, `PushButton( "&Help"   ) )
                     )
               );

    UI::UserInput();
    UI::CloseDialog();
}
```



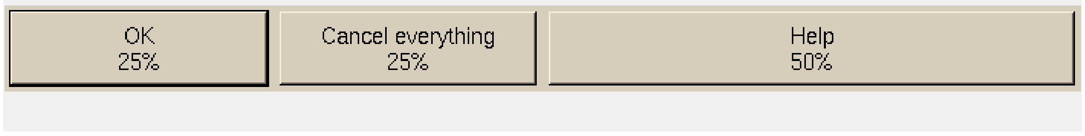
```
          {
    // Layout example:
    //
    // Build a dialog with three equal sized buttons.
    //
    // The equal `HWeight()s will make the buttons equal sized.
    // When resized, all buttons will resize equally in order to
    // maintain the equal layout weights.

    UI::OpenDialog(
               `HBox(
                     `HWeight(1, `PushButton( `opt(`default), "&OK" ) ),
                     `HWeight(1, `PushButton( "&Cancel everything" ) ),
                     `HWeight(1, `PushButton( "&Help"   ) )
                     )
               );

    UI::UserInput();
    UI::CloseDialog();
}
```



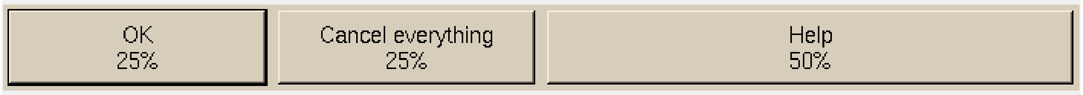
```
        {
    // Layout example:
    //
    // Build a dialog with three widgets with different weights and
    // two widgets without any weight.
    //
    // All widgets will get at least their "nice size". The weighted
    // ones may get even more to maintain their share of the overall
    // weight.
    //
    // Upon resize all widgets will resize to maintain their
    // respective weights at all times. The non-weighted widgets will
    // retain their "nice size" regardless whether or not they are
    // stretchable.
    //

    UI::OpenDialog(
               `HBox(
                     `HWeight( 33, `PushButton( `opt(`default), "OK\n33%" ) ),
                     `PushButton( `opt(`hstretch), "Apply\nNo Weight" ),
                     `HWeight( 33, `PushButton( "Cancel\n33%" ) ),
                     `PushButton( "Reset to defaults\nNo Weight" ),
                     `HWeight( 66, `PushButton( "Help\n66%"    ) )
                     )
               );

    UI::UserInput();
    UI::CloseDialog();
}
```



```
        {
    // Layout example:
    //
    // Build a dialog with three widgets with different weights.
    //
    // Weights do not need to add up to 100 or any other special
    // number, but it helps the application programmer to keep track
    // of the percentage of each part of the layout.
    //
    // Notice how the second button commands the overall size of the
    // dialog since it has the largest "nice size" to "weight" ratio.
    //
    // Upon resize all widgets will resize to maintain their
    // respective weights at all times.
    //

    UI::OpenDialog(
               `HBox(
                     `HWeight( 25, `PushButton( `opt(`default), "OK\n25%" ) ),
                     `HWeight( 25, `PushButton( "Cancel everything\n25%" ) ),
                     `HWeight( 50, `PushButton( "Help\n50%"    ) )
                     )
               );

    UI::UserInput();
    UI::CloseDialog();
}
```



```
          {
    // Layout example:
    //
    // Build a dialog with three widgets with different weights.
    //
    // Weights do not need to add up to 100 or any other special
    // number, but it helps the application programmer to keep track
    // of the percentage of each part of the layout.
    //
    // Notice how the second button commands the overall size of the
    // dialog since it has the largest "nice size" to "weight" ratio.
    //
    // Upon resize all widgets will resize to maintain their
    // respective weights at all times.
    //

    UI::OpenDialog(
               `HBox(
                     `HWeight( 1, `PushButton( `opt(`default), "OK\n25%" ) ),
                     `HWeight( 1, `PushButton( "Cancel everything\n25%" ) ),
                     `HWeight( 2, `PushButton( "Help\n50%"   ) )
                     )
               );

    UI::UserInput();
    UI::CloseDialog();
}
```



Name

HBox -- Generic layout: Arrange widgets horizontally or vertically

HBox, VBox

HBox ();
VBox ();

Options

debugLayout	verbose logging

Optional Arguments

term *child1*	the first child widget
term *child2*	the second child widget
term *child3*	the third child widget
term *child4*	the fourth child widget (and so on...)

Description

The layout boxes are used to split up the dialog and layout a number of widgets horizontally (*HBox*) or vertically (*VBox*).

Usage

```
        HBox( `PushButton( `id( `ok ), "OK" ), `PushButton( `id( `cancel ), "Cancel" ) )
```

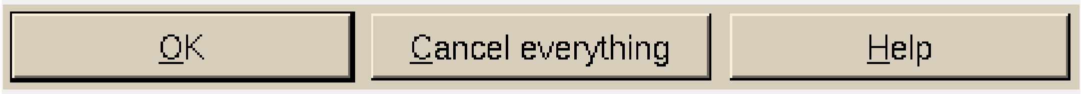
Examples

```
      {
    UI::OpenDialog(
               `VBox(
                     `PushButton("First"),
                     `PushButton("Second"),
                     `PushButton("Third")
                     )
               );
    UI::UserInput();
    UI::CloseDialog();
}
```

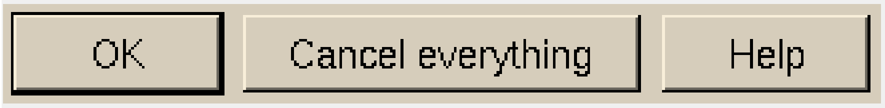
```
      {
    UI::OpenDialog(
               `HBox(
                     `PushButton("First" ),
                     `PushButton("Second"),
                     `PushButton("Third" )
                     )
```

```
            );
    UI::UserInput();
    UI::CloseDialog();
}
```

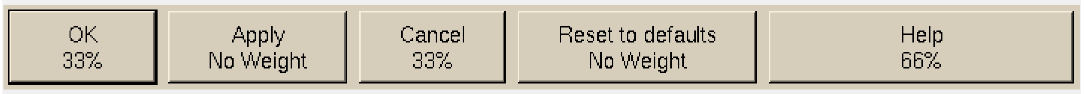
```
          {
    // Layout example:
    //
    // Build a dialog with three equal sized buttons.
    //
    // The equal `HWeight()s will make the buttons equal sized.
    // When resized, all buttons will resize equally in order to
    // maintain the equal layout weights.

    UI::OpenDialog(
             `HBox(
                   `HWeight(1, `PushButton( `opt(`default), "&OK" ) ),
                   `HWeight(1, `PushButton( "&Cancel everything" ) ),
                   `HWeight(1, `PushButton( "&Help"   ) )
                   )
             );

    UI::UserInput();
    UI::CloseDialog();
}
```



```
          {
    // Layout example:
    //
    // Build a dialog with three widgets without any weights.
    //
    // Each widget will get its "nice size", i.e. the size that makes
    // the widget's contents fit into it.
    //
    // Upon resize the widgets will keep their sizes if enlarged
    // (since none of them is stretchable), i.e. there will be empty
    // space to the right.
    //

    UI::OpenDialog(
             `HBox(
                   `PushButton( `opt(`default), "OK" ),
                   `PushButton( "Cancel everything"  ),
                   `PushButton( "Help" )
                   )
             );

    UI::UserInput();
    UI::CloseDialog();
}
```



```
          {
    // Layout example:
    //
    // Build a dialog with three widgets with different weights and
    // two widgets without any weight.
    //
    // All widgets will get at least their "nice size". The weighted
    // ones may get even more to maintain their share of the overall
    // weight.
    //
    // Upon resize all widgets will resize to maintain their
    // respective weights at all times. The non-weighted widgets will
    // retain their "nice size" regardless whether or not they are
    // stretchable.
    //

    UI::OpenDialog(
               `HBox(
                     `HWeight( 33, `PushButton( `opt(`default), "OK\n33%" ) ),
                     `PushButton( `opt(`hstretch), "Apply\nNo Weight" ),
                     `HWeight( 33, `PushButton( "Cancel\n33%" ) ),
                     `PushButton( "Reset to defaults\nNo Weight" ),
                     `HWeight( 66, `PushButton( "Help\n66%"   ) )
                     )
               );

    UI::UserInput();
    UI::CloseDialog();
}
```



Name

Label -- Simple static text

Label, Heading

```
Label (label);
string label ;
Heading (label);
string label ;
```

Parameters

string *label*

Options

outputField	make the label look like an input field in read-only mode

Properties

string *Value*	the label text

Description

A *Label* is some text displayed in the dialog. A *Heading* is a text with a font marking it as heading. The text can have more than one line, in which case line feed must be entered.

Usage

```
        `Label( "Here goes some text\nsecond line" )
```

Examples

```
         {
    UI::OpenDialog(
               `VBox(
                     `Label("Hello, World!"),
                     `PushButton("&OK")
                     )
               );
    UI::UserInput();
    UI::CloseDialog();
}
```

```
         {
    UI::OpenDialog(
               `VBox(
                     `Label("Labels can have\nmultiple lines." ),
                     `PushButton("&OK")
                     )
```

```
           );
    UI::UserInput();
    UI::CloseDialog();
}
```

```
      {
    // Build dialog with one label, 4 Beatles buttons and an OK button.
    UI::OpenDialog(
           `VBox(
                 `Label("Select your favourite Beatle:"),
                 `Label(`id(`beatle), `opt(`outputField), "   "),
                 `HBox(
                       `PushButton(`id(`john), "John" ),
                       `PushButton(`id(`paul), "Paul" ),
                       `PushButton(`id(`george), "George"),
                       `PushButton(`id(`ringo), "Ringo" )),
                 `PushButton(`id(`ok), "&OK")
                 )
           );

    // Wait for user input.
    any button = nil;

    // Input loop that only the OK button will leave.
    // The 4 Beatles buttons will just propose a name.
    repeat
    {
        button = UI::UserInput();

        if      ( button == `john ) UI::ChangeWidget(`id(`beatle), `Value, "John Lennon");
        else if ( button == `paul ) UI::ChangeWidget(`id(`beatle), `Value, "Paul McCartney");
        else if ( button == `george ) UI::ChangeWidget(`id(`beatle), `Value, "George Harrison");
        else if ( button == `ringo ) UI::ChangeWidget(`id(`beatle), `Value, "Ringo Starr" );

        // Recalculate the layout - this is necessary since the label widget
        // doesn't recompute its size upon changing its value.
        UI::RecalcLayout();

    } until ( button == `ok );


    // Retrieve the label's value.
    string name = (string) UI::QueryWidget(`id(`beatle), `Value);

    // Close the dialog.
    // Remember to read values from the dialog's widgets BEFORE closing it!
    UI::CloseDialog();

    // Pop up a new dialog to echo the input.
    UI::OpenDialog(
           `VBox(
                 `VSpacing(),
                 `HBox(
                       `Label("You selected:"),
                       `Label(`opt(`outputField), name),
                       `HSpacing()
                       ),
                 `PushButton(`opt(`default), "&OK")
                 )
           );
    UI::UserInput();
    UI::CloseDialog();
}
```

```
      {
    // Build dialog with one label, 4 Beatles buttons and an OK button.
    UI::OpenDialog(
           `VBox(
                 `Label("My favourite Beatle:"),
                 // `Heading(`id(`favourite), "Press one of the buttons below"),
                 `Heading(`id(`favourite), "(please select one)"),
                 `HBox(
                       `PushButton(`id(`john), "John" ),
                       `PushButton(`id(`paul), "Paul" ),
                       `PushButton(`id(`george), "George"),
                       `PushButton(`id(`ringo), "Ringo" )),
                 `PushButton(`id(`ok), "&OK")
                 )
           );

    // Wait for user input.
    any button = nil;

    // Input loop that only the OK button will leave.
    // The 4 Beatles buttons will just propose a name.
    repeat
    {
        button = UI::UserInput();

        if      ( button == `john ) UI::ChangeWidget(`id(`favourite), `Value, "John Lennon");
        else if ( button == `paul ) UI::ChangeWidget(`id(`favourite), `Value, "Paul McCartney");
        else if ( button == `george ) UI::ChangeWidget(`id(`favourite), `Value, "George Harrison");
        else if ( button == `ringo ) UI::ChangeWidget(`id(`favourite), `Value, "Ringo Starr" );

    } until ( button == `ok );
}
```

```
          {
    // Build dialog with one label, 4 Beatles buttons and an OK button.
    UI::OpenDialog(
               `VBox(
                     `Label("My favourite Beatle:"),
                     // `Heading(`id(`favourite), "Press one of the buttons below"),
                     `Heading(`id(`favourite), "(please select one)"),
                     `HBox(
                           `PushButton(`id(`john), "John" ),
                           `PushButton(`id(`paul), "Paul" ),
                           `PushButton(`id(`george), "George"),
                           `PushButton(`id(`ringo), "Ringo" )),
                     `PushButton(`id(`ok), "&OK")
                     )
               );

    // Wait for user input.
    any button = nil;

    // Input loop that only the OK button will leave.
    // The 4 Beatles buttons will just propose a name.
    repeat
    {
        button = UI::UserInput();

        if      ( button == `john ) UI::ChangeWidget(`id(`favourite), `Value, "John Lennon");
        else if ( button == `paul ) UI::ChangeWidget(`id(`favourite), `Value, "Paul McCartney");
        else if ( button == `george ) UI::ChangeWidget(`id(`favourite), `Value, "George Harrison");
        else if ( button == `ringo ) UI::ChangeWidget(`id(`favourite), `Value, "Ringo Starr" );

    } until ( button == `ok );
}
```

```
          {
    // Build dialog with one label, 4 Beatles buttons and an OK button.
    UI::OpenDialog(
               `VBox(
                     `Label("My favourite Beatle:"),
                     // `Heading(`id(`favourite), "Press one of the buttons below"),
                     `Heading(`id(`favourite), "(please select one)"),
                     `HBox(
                           `PushButton(`id(`john), "John" ),
                           `PushButton(`id(`paul), "Paul" ),
                           `PushButton(`id(`george), "George"),
                           `PushButton(`id(`ringo), "Ringo" )),
                     `PushButton(`id(`ok), "&OK")
                     )
               );

    // Wait for user input.
    any button = nil;

    // Input loop that only the OK button will leave.
    // The 4 Beatles buttons will just propose a name.
    repeat
    {
        button = UI::UserInput();

        if      ( button == `john ) UI::ChangeWidget(`id(`favourite), `Value, "John Lennon");
        else if ( button == `paul ) UI::ChangeWidget(`id(`favourite), `Value, "Paul McCartney");
        else if ( button == `george ) UI::ChangeWidget(`id(`favourite), `Value, "George Harrison");
        else if ( button == `ringo ) UI::ChangeWidget(`id(`favourite), `Value, "Ringo Starr" );

    } until ( button == `ok );
}
```

Name

RichText -- Static text with HTML-like formatting

RichText

```
RichText (text);
string text ;
```

Parameters

string *text*

Options

plainText	don't interpret text as HTML
autoScrollDown	automatically scroll down for each text change
shrinkable	make the widget very small

Properties

string *Value*	the RichText's text contents

Description

A *RichText* is a text area with two major differences to a *Label*: The amount of data it can contain is not restricted by the layout and a number of control sequences are allowed, which control the layout of the text.

Usage

```
        `RichText( "This is a bold text" )
```

Examples

```
         // Example for a RichText widget
{
    UI::OpenDialog( `opt(`defaultsize),
                `VBox(
                      `RichText( "<h3>RichText example</h3>"
                                + "<p>This is a <i>RichText</i> widget.</p>"
                                + "<p>It's very much like <i>HTML</i>, but not quite as powerful.</p>"
                                + "<p><b>bold</b> and <i>italic</i> you can rely on.</p>"
                                + "<p>"
                                + "<font color=blue>colored </font>"
                                + "<font color=red> text         </font>"
                                + "<font color=green> might         </font>"
                                + "<font color=magenta> or         </font>"
                                + "<font color=cyan> might         </font>"
                                + "<font color=blue> not </font>"
                                + "<font color=red> be </font>"
                                + "<font color=green> available,</font>"
```

```
                                 + "<font color=magenta> depending </font>"
                                 + "<font color=cyan> on         </font>"
                                 + "<font color=blue> the         </font>"
                                 + "<font color=red> UI.         </font>"
                                 + "</p>"
                                 + "<p>The product name is automatically replaced by the UI."
                                 + "Use the special macro <b>&amp;product;</b> for that."
                                 + "</p><p>"
                                 + "The current product name is <b>&product;</b>."
                                 + "</p>"
                                 ),
                      `PushButton(`opt(`default), "&OK")
                      )
             );
  UI::UserInput();
  UI::CloseDialog();
}
```

Name

LogView -- scrollable log lines like "tail -f"

LogView

```
**LogView** (label, visibleLines, maxLines);
string label ;
integer visibleLines ;
integer maxLines ;
```

Parameters

string *label*	(above the log lines)
integer *visibleLines*	number of visible lines (without scrolling)
integer *maxLines*	number of log lines to store (use 0 for "all")

Properties

string *Value*	All log lines. Set this property to replace or clear the entire contents. Can only be set, not queried.
string *LastLine*	The last log line. Set this property to append one or more line(s) to the log. Can only be set, not queried.
string *Label*	The label above the log text.

Description

A scrolled output-only text window where ASCII output of any kind can be redirected - very much like a shell window with "tail -f".

The LogView will keep up to "maxLines" of output, discarding the oldest lines if there are more. If "maxLines" is set to 0, all lines will be kept.

"visibleLines" lines will be visible by default (without scrolling) unless you stretch the widget in the layout.

Use *ChangeWidget(`id( `log), `LastLine, "bla blurb...\n")* to append one or several line(s) to the output. Notice the newline at the end of each line!

Use *ChangeWidget(`id( `log), `Value, "bla blurb...\n")* to replace the entire contents of the LogView.

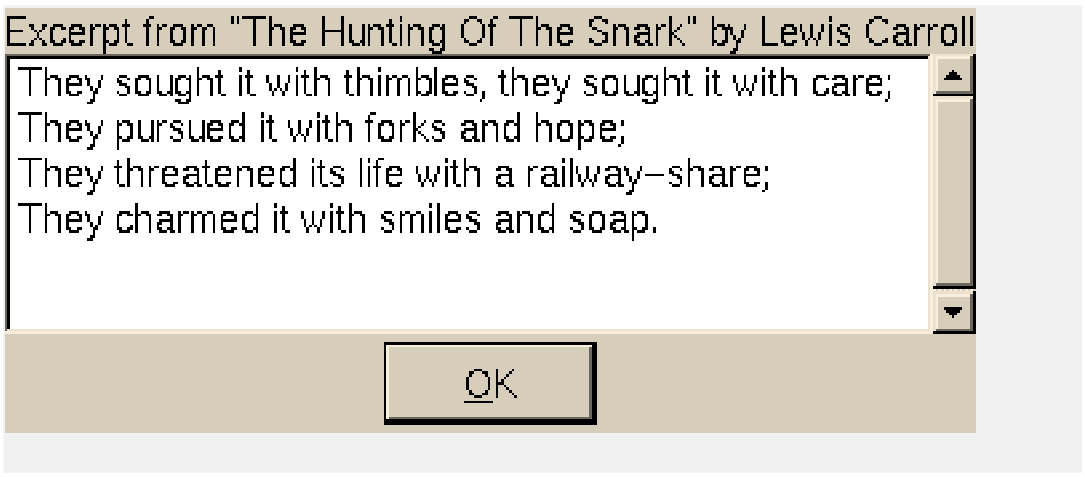
Use *ChangeWidget(`id( `log), `Value, "")* to clear the contents.

Usage

```
        `LogView( "Log file", 4, 200 );
```

Examples

```
        {
    string part1 =
        "They sought it with thimbles, they sought it with care;\n" +
        "They pursued it with forks and hope;\n"                 +
        "They threatened its life with a railway-share;\n"        +
        "They charmed it with smiles and soap. \n"                +
        "\n";

    string part2 =
        "Then the Butcher contrived an ingenious plan\n"         +
        "For making a separate sally;\n"                         +
        "And fixed on a spot unfrequented by man,\n"                +
        "A dismal and desolate valley. \n"                        +
        "\n";

    string part3 =
        "But the very same plan to the Beaver occurred:\n"        +
        "It had chosen the very same place:\n"                        +
        "Yet neither betrayed, by a sign or a word,\n"                +
        "The disgust that appeared in his face. \n"                +
        "\n";

    string part4 =
        "Each thought he was thinking of nothing but \"Snark\"\n" +
        "And the glorious work of the day;\n"                        +
        "And each tried to pretend that he did not remark\n"        +
        "That the other was going that way. \n"                        +
        "\n";

    string part5 =
        "But the valley grew narrow and narrower still,\n"        +
        "And the evening got darker and colder,\n"                +
        "Till (merely from nervousness, not from goodwill)\n"        +
        "They marched along shoulder to shoulder. \n"                +
        "\n";

    string part6 =
        "Then a scream, shrill and high, rent the shuddering sky,\n" +
        "And they knew that some danger was near:\n"                +
        "The Beaver turned pale to the tip of its tail,\n"        +
        "And even the Butcher felt queer. \n"                        +
        "\n";

    string part7 =
        "He thought of his childhood, left far far behind--\n"        +
        "That blissful and innocent state--\n"                        +
        "The sound so exactly recalled to his mind\n"                +
        "A pencil that squeaks on a slate! \n"                        +
        "\n";

    string part8 =
        "\"'Tis the voice of the Jubjub!\" he suddenly cried.\n" +
        "(This man, that they used to call \"Dunce.\")\n"         +
        "\"As the Bellman would tell you,\" he added with pride,\n" +
        "\"I have uttered that sentiment once.\n"                +
        "\n";

    string thats_it = "\n\n*** Press [OK] once more to exit. ***";


    UI::OpenDialog(
               `VBox(
                     `LogView(`id(`log),
                              "&Excerpt from \"The Hunting Of The Snark\" by Lewis Carroll",
                              5, // visible lines
                              10), // lines to store
                     `PushButton(`opt(`default), "&OK")
                     )
               );

    UI::ChangeWidget(`id(`log), `LastLine, part1 ); UI::UserInput();
    UI::ChangeWidget(`id(`log), `LastLine, part2 ); UI::UserInput();
    UI::ChangeWidget(`id(`log), `LastLine, part3 ); UI::UserInput();
    UI::ChangeWidget(`id(`log), `LastLine, part4 ); UI::UserInput();
    UI::ChangeWidget(`id(`log), `LastLine, part5 ); UI::UserInput();
    UI::ChangeWidget(`id(`log), `LastLine, part6 ); UI::UserInput();
    UI::ChangeWidget(`id(`log), `LastLine, part7 ); UI::UserInput();
    UI::ChangeWidget(`id(`log), `LastLine, part8 ); UI::UserInput();

    UI::ChangeWidget(`id(`log), `Value, thats_it); UI::UserInput();
    UI::CloseDialog();
}
```


Excerpt from "The Hunting Of The Snark" by Lewis Carroll

They sought it with thimbles, they sought it with care;
They pursued it with forks and hope;
They threatened its life with a railway–share;
They charmed it with smiles and soap.

Name

PushButton -- Perform action on click

PushButton, IconButton

```
PushButton (iconName, label);
string iconName ;
string label ;
IconButton (iconName, label);
string iconName ;
string label ;
```

Parameters

string *`iconName`*	(IconButton only)
string *`label`*	

Options

`default`	makes this button the dialogs default button

Properties

string *`Label`*	the text on the PushButton

Description

A *`PushButton`* is a button with a text label the user can press in order to activate some action. If you call *`UserInput()`* and the user presses the button, *`UserInput()`* returns with the id of the pressed button.

You can (and should) provide keybord shortcuts along with the button label. For example "&amp; Apply" as a button label will allow the user to activate the button with Alt-A, even if it currently doesn't have keyboard focus. This is important for UIs that don't support using a mouse.

An *`IconButton`* is pretty much the same, but it has an icon in addition to the text. If the UI cannot handle icons, it displays only the text, and the icon is silently omitted.

Icons are (at the time of this writing) loaded from the *theme* directory, <filename>/usr/share/YaST2/theme/current</filename>.

Usage

```
        `PushButton( `id( `click ), `opt( `default, `hstretch ), "Click me" )
```

Examples

```
          {
    // Build a dialog with one button.
    // Wait until that button is clicked,
    // then close the dialog and terminate.

    UI::OpenDialog(
               `PushButton( "&OK" )
               );

    UI::UserInput();
    UI::CloseDialog();
}
```

```
          {
    // Build dialog with three buttons.
    // "Cancel" is the default button, i.e. pressing "Return" will
    // activate it.

    UI::OpenDialog(
               `HBox(
                     `PushButton( `id(`ok),                 "&OK" ),
                     `PushButton( `id(`cancel), `opt(`default), "&Cancel" ),
                     `PushButton( `id(`help),                 "&Help"   )
                     )
               );

    // Wait for user input. The value returned is the ID of the widget
    // that makes UI::UserInput() terminate, i.e. the respective button ID.
    any button_id = UI::UserInput();

    // Close the dialog.
    UI::CloseDialog();


    // Process the input.
    string button_name = "";
    if                 ( button_id == `ok         )         button_name = "OK";
    else if ( button_id == `cancel )         button_name = "Cancel";
    else if ( button_id == `help         )         button_name = "Help";

    // Pop up a new dialog to display what button was clicked.
    UI::OpenDialog(
               `VBox(
                     `Label( "You clicked button \"" + button_name + "\"."),
                     `PushButton( `opt(`default), "&OK" )
                     )
               );
    UI::UserInput();
    UI::CloseDialog();
}
```

```
          {
    // Build a dialog with one icon button.
    // Wait until that button is clicked,
    // then close the dialog and terminate.

    UI::OpenDialog(
               `IconButton( "topRow_right.png", "&OK" )
               );

    UI::UserInput();
    UI::CloseDialog();
}
```

Name

MenuButton -- Button with popup menu

MenuButton

```
MenuButton (label, menu);
string label ;
itemList menu ;
```

Parameters

string *label*	
itemList *menu*	items

Properties

string *Label*	the text on the MenuButton

Description

This is a widget that looks very much like a *PushButton*, but unlike a *PushButton* it doesn't immediately start some action but opens a popup menu from where the user can select an item that starts an action. Any item may in turn open a submenu etc.

UserInput() returns the ID of a menu item if one was activated. It will never return the ID of the *MenuButton* itself.

Style guide hint: Don't overuse this widget. Use it for dialogs that provide lots of actions. Make the most frequently used actions accessible via normal *PushButtons*. Move less frequently used actions (e.g. "expert" actions) into one or more *MenuButtons*. Don't nest the popup menus too deep - the deeper the nesting, the more awkward the user interface will be.

You can (and should) provide keybord shortcuts along with the button label as well as for any menu item.

Usage

```
        `MenuButton( "button label", [ `item( `id( `doit ), "&amp; Do it" ), `item( `id( `something ), "&amp
```

Examples

```
          {
    // Build a dialog with one menu button.
    // Wait the user selects a menu entry,
    // then close the dialog and terminate.

    // Please note it's pretty pointless to create menu entries without an ID -
    // you'd never know what entry the user selected.

    UI::OpenDialog(
               `MenuButton( "&Create",
                            [
                             `item(`id(`folder), "&Folder"         ),
                             `item(`id(`text),   "&Text File" ),
                             `item(`id(`image),  "&Image" )
                             ]
                            )
              );

    any id = UI::UserInput();
    UI::CloseDialog();

    y2milestone( "Selected: %1", id );
}
```

```
          {
    // Build a dialog with one menu button with a submenu.
    // Wait the user selects a menu entry,
    // then close the dialog and terminate.

    // Please note it's pretty pointless to create menu entries without an ID -
    // you'd never know what entry the user selected.

    UI::OpenDialog(
               `MenuButton( "&Create",
                            [
                             `item(`id(`folder), "&Folder"         ),
                             `menu( "&Document",
                                    [
                                     `item(`id(`text),   "&Text File" ),
                                     `item(`id(`image),  "&Image" )
                                    ]
                                   )
                             ]
                            )
              );

    any id = UI::UserInput();
    UI::CloseDialog();

    y2milestone( "Selected: %1", id );
}
```

Name

CheckBox -- Clickable on/off toggle button

CheckBox

```
CheckBox (label, checked);
string label ;
boolean|nil checked ;
```

Parameters

string *label*	the text describing the check box

Optional Arguments

boolean|nil *checked*	whether the check box should start checked - nil means tristate condition, i.e. neither on nor off

Description

A checkbox widget has two states: Checked and not checked. It returns no user input but you can query and change its state via the *Value* property.

Usage

```
        `CheckBox( `id( `cheese ), "&amp; Extra cheese" )
```

Examples

```
         {
    UI::OpenDialog(
               `CheckBox("A &checked check box\nwith multi-line", true)
               );
    UI::UserInput();
    UI::CloseDialog();
}
```

```
         {
    UI::OpenDialog(
               `VBox(
                     `Label("Select your extras"),
                     `Left(`CheckBox(`id(`cheese), "Extra Cheese")),
                     `Left(`CheckBox(`id(`pepr),   "Pepperoni", true)),
                     `PushButton("&OK")
                     )
               );
    UI::UserInput();
    boolean cheese = (boolean) UI::QueryWidget(`id(`cheese), `Value);
    boolean pepr   = (boolean) UI::QueryWidget(`id(`pepr),   `Value);
    UI::CloseDialog();

    define string yesno(boolean b) ``{ if (b) return "yes"; else return "no"; };

    UI::OpenDialog(
               `VBox(
                     `Left(`Label("Extra Cheese: " + yesno(cheese))),
                     `Left(`Label("Pepperoni: "    + yesno(pepr))),
                     `PushButton("&OK")
                     )
               );
    UI::UserInput();
    UI::CloseDialog();
}
```

```
          {
    // Build dialog with one check box and buttons to set its state to
    // on, off or "don't care" (tri-state).

    UI::OpenDialog(
               `VBox(
                     `CheckBox(`id(`cb), "Format hard disk"),
                     `HBox(
                           `HWeight(1, `PushButton(`id(`setOn ),   "Set on"     ) ),
                           `HWeight(1, `PushButton(`id(`setOff),   "Set off"    ) ),
                           `HWeight(1, `PushButton(`id(`dontCare), "Don't care" ) )
                           ),
                     `PushButton(`id(`ok), "&OK")
                     )
               );


    // Input loop. Will be left only after 'OK' is clicked.

    any button = nil;

    repeat
        {
            button = UI::UserInput();

            if      ( button == `setOn    ) UI::ChangeWidget ( `id(`cb), `Value, true  );
            else if ( button == `setOff   ) UI::ChangeWidget ( `id(`cb), `Value, false );
            else if ( button == `dontCare ) UI::ChangeWidget ( `id(`cb), `Value, nil   );
        } until ( button == `ok );


    // Get the check box's value.
    //
    // Notice: The return value of UI::UserInput() does NOT return this value!
    // Rather, it returns the ID of the widget (normally the PushButton)
    // that caused UI::UserInput() to return.

    boolean cb_val = (boolean) UI::QueryWidget(`id(`cb), `Value);

    // Close the dialog.
    // Remember to read values from the dialog's widgets BEFORE closing it!
    UI::CloseDialog();

    // Convert the check box value to string.
    string valStr = "Don't care";
    if ( cb_val == true  ) valStr = "Yes";
    if ( cb_val == false ) valStr = "No";

    // Pop up a new dialog to echo the input.
    UI::OpenDialog(
               `VBox(
                     `Label("Your selection:"),
                     `Label(valStr),
                     `PushButton("&OK")
                     )
               );
    UI::UserInput();
    UI::CloseDialog();
}
```

Name

RadioButton -- Clickable on/off toggle button for radio boxes

RadioButton

```
RadioButton (label, selected);
string label ;
boolean selected ;
```

Parameters

string *label*

Optional Arguments

boolean *selected*

Properties

boolean *Value*	the state of the RadioButton (on or off)
string *Label*	the RadioButton's text

Description

A radio button is not usefull alone. Radio buttons are group such that the user can select one radio button of a group. It is much like a selection box, but radio buttons can be dispersed over the dialog. Radio buttons must be contained in a *RadioButtonGroup*.

Usage

```
        `RadioButton( `id( `now ), "Crash now", true )
```

Examples

```
        {
   UI::OpenDialog(
             `RadioButtonGroup(`id(`rb),
                              `VBox(
                                   `Label("How do you want to crash?"),
                                   `Left(`RadioButton(`id(0), "No&w")),
                                   `Left(`RadioButton(`id(1), "&Every now an then" )),
                                   `Left(`RadioButton(`id(2), "Every &five minutes", true)),
                                   `Left(`RadioButton(`id(3), "Ne&ver", true )),
                                   `HBox(
                                        `PushButton(`id(`next), "&Next"),
                                        `PushButton("&OK")
                                        )
                                   )
                              )
             );

   while (true)
   {
       any ret = UI::UserInput();
       if (ret == `next)
       {
           integer current = (integer) UI::QueryWidget(`id(`rb), `CurrentButton);
           current = (current + 1) % 4;
           UI::ChangeWidget(`id(`rb), `CurrentButton, current);
        }
        else break;
    }

    UI::CloseDialog();
}
```

```
          {
    UI::OpenDialog(
               `RadioButtonGroup(`id(`rb),
                                 `VBox(
                                       `Label("How do you want to crash?"),
                                       `Left(`RadioButton(`id(0), `opt(`notify), "No&w")),
                                       `Left(`RadioButton(`id(1), `opt(`notify), "&Every now an then" )),
                                       `Left(`RadioButton(`id(2), `opt(`notify), "Every &five minutes")),
                                       `Left(`RadioButton(`id(3), `opt(`notify), "Ne&ver", true )),
                                       `HBox(
                                             `PushButton(`id(`next),  "&Next"),
                                             `PushButton(`id(`close), "&Close")
                                             )
                                       )
                                 )
               );

    while (true)
    {
        any ret = UI::UserInput();

        if ( ret == `close ) break;
        else if (ret == `next)
        {
            y2milestone("Hit next");
            integer current = (integer) UI::QueryWidget(`id(`rb), `CurrentButton);
            current = (current + 1) % 4;
            UI::ChangeWidget(`id(`rb), `CurrentButton, current);
        }
        else
        {
            y2milestone("Hit RadioButton #%1", ret);
        }
    }

    y2milestone("Terminating.");

    UI::CloseDialog();
}
```

```
          {
    UI::OpenDialog( `VBox(
                      `Frame ( "CPU &Speed",
                               `RadioButtonGroup(
                                                 `VBox(
                                                       `Left(`RadioButton("Normal" )),
                                                       `Left(`RadioButton("Overclocked" )),
                                                       `Left(`RadioButton("Red Hot" )),
                                                       `Left(`RadioButton("Melting", true ))
                                                       )
                                                 )
                               ),
                      `PushButton("&OK")
                      )
                );
    UI::UserInput();
    UI::CloseDialog();
}
```

Name

RadioButtonGroup -- Radio box - select one of many radio buttons

RadioButtonGroup

```
RadioButtonGroup (child);
term child ;
```

Parameters

term *child*	the child widget

Properties

any *Current-Button*	The id of the currently selected radio button belonging to this group. If no button is selected, CurrentButton is nil.

Description

A *RadioButtonGroup* is a container widget that has neither impact on the layout nor has it a graphical representation. It is just used to logically group RadioButtons together so the one-out-of-many selection strategy can be ensured.

Radio button groups may be nested. Looking bottom up we can say that a radio button belongs to the radio button group that is nearest to it. If you give the *RadioButtonGroup* widget an id, you can use it to query and set which radio button is currently selected.

Usage

```
          `RadioButtonGroup( `id( rb ), `VBox( ... ) )
```

Examples

```
      {
UI::OpenDialog(
           `RadioButtonGroup(`id(`rb),
                             `VBox(
                                   `Label("How do you want to crash?"),
                                   `Left(`RadioButton(`id(0), "No&w")),
                                   `Left(`RadioButton(`id(1), "&Every now an then" )),
                                   `Left(`RadioButton(`id(2), "Every &five minutes", true)),
                                   `Left(`RadioButton(`id(3), "Ne&ver", true )),
                                   `HBox(
                                         `PushButton(`id(`next), "&Next"),
                                         `PushButton("&OK")
                                         )
                                   )
                             )
           );

while (true)
{
    any ret = UI::UserInput();
    if (ret == `next)
    {
        integer current = (integer) UI::QueryWidget(`id(`rb), `CurrentButton);
        current = (current + 1) % 4;
        UI::ChangeWidget(`id(`rb), `CurrentButton, current);
    }
    else break;
    }

    UI::CloseDialog();
}
```

```
          {
    UI::OpenDialog( `VBox(
                      `Frame ( "CPU &Speed",
                               `RadioButtonGroup(
                                                 `VBox(
                                                       `Left(`RadioButton("Normal" )),
                                                       `Left(`RadioButton("Overclocked" )),
                                                       `Left(`RadioButton("Red Hot" )),
                                                       `Left(`RadioButton("Melting", true ))
                                                       )
                                                 )
                               ),
                      `PushButton("&OK")
                      )
                );
    UI::UserInput();
    UI::CloseDialog();
}
```

Name

TextEntry -- Input field

TextEntry, Password

```
TextEntry (label, defaulttext);
string label ;
string defaulttext ;
Password (label, defaulttext);
string label ;
string defaulttext ;
```

Parameters

string *label*	the label describing the meaning of the entry

Options

shrinkable	make the input field very small

Optional Arguments

string *default-text*	The text contained in the text entry

Properties

string *Value*	the field's contents (the user input)
string *Label*	label above the field
string *Valid-Chars*	valid input characters

Description

This widget is a one line text entry field with a label above it. An initial text can be provided.

Note

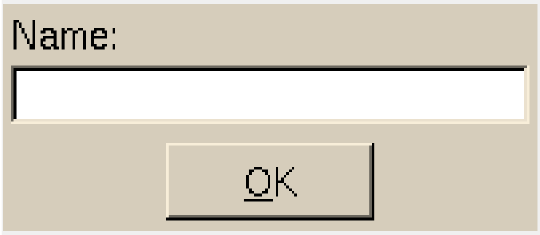
You can and should set a keyboard shortcut within the label. When the user presses the hotkey, the corresponding text entry widget will get the keyboard focus.

Usage

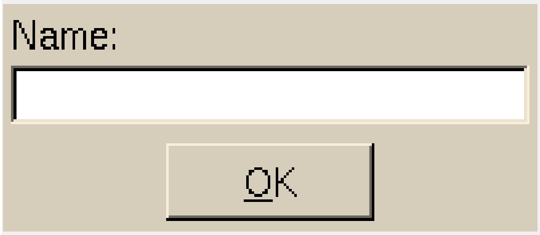
```
	`TextEntry( `id( `name ), "Enter your name:", "Kilroy" )
```

Examples

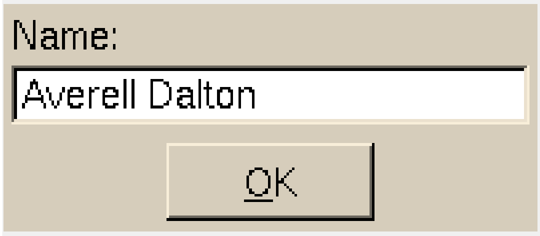
```
        {
    UI::OpenDialog(
               `VBox(
                     `TextEntry("Name:"),
                     `PushButton("&OK")
                     )
               );
    UI::UserInput();
    UI::CloseDialog();
}
```



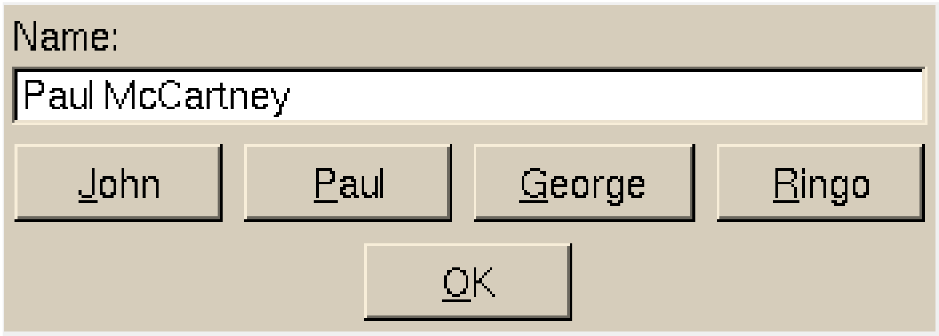
```
        {
    // Build dialog with one text entry field and an OK button.
    UI::OpenDialog(
               `VBox(
                     `TextEntry(`id(`name), "Name:"),
                     `PushButton("&OK")
                     )
               );

    // Wait for user input.
    UI::UserInput();

    // Get the input from the text entry field.
    //
    // Notice: The return value of UI::UserInput() does NOT return this value!
    // Rather, it returns the ID of the widget (normally the PushButton)
    // that caused UI::UserInput() to return.
    string name = (string) UI::QueryWidget(`id(`name), `Value);

    // Close the dialog.
    // Remember to read values from the dialog's widgets BEFORE closing it!
    UI::CloseDialog();

    // Pop up a new dialog to echo the input.
    UI::OpenDialog(
               `VBox(
                     `Label("You entered:"),
                     `Label(name),
                     `PushButton("&OK")
                     )
               );
    UI::UserInput();
    UI::CloseDialog();
}
```



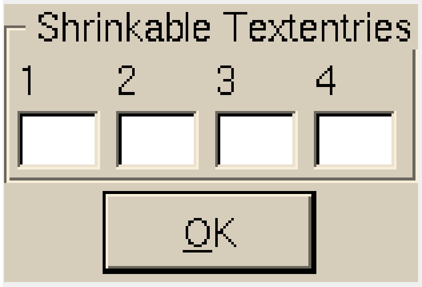
```
        {
    // Build dialog with one text entry field and an OK button.
    UI::OpenDialog(
               `VBox(
                     `TextEntry(`id(`name), "You will never see this:"),
                  `PushButton("&OK")
                  )
            );

    // Set an initial value for the text entry field.
    UI::ChangeWidget(`id(`name), `Value, "Averell Dalton");

    // Change the text entry field's label.
    UI::ChangeWidget(`id(`name), `Label, "Name:");

    // Wait for user input.
    UI::UserInput();

    // Get the input from the text entry field.
    //
    // Notice: The return value of UI::UserInput() does NOT return this value!
    // Rather, it returns the ID of the widget (normally the PushButton)
    // that caused UI::UserInput() to return.
    string name = (string) UI::QueryWidget(`id(`name), `Value);

    // Close the dialog.
    // Remember to read values from the dialog's widgets BEFORE closing it!
    UI::CloseDialog();


    // Pop up a new dialog to echo the input.
    UI::OpenDialog(
            `VBox(
                  `Label("You entered:"),
                  `Label(name),
                  `PushButton("&OK")
                  )
            );
    UI::UserInput();
    UI::CloseDialog();
}
```



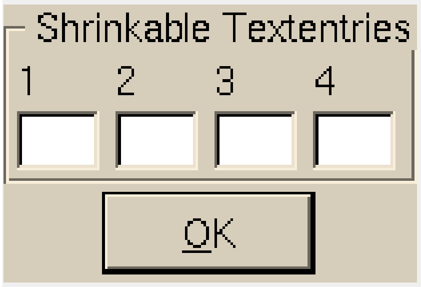
```
          {
    // Build dialog with one text entry field, 4 Beatles buttons and an OK button.
    UI::OpenDialog(
            `VBox(
                  `TextEntry(`id(`name), "Name:"),
                  `HBox(
                        `PushButton(`id(`john), "&John"  ),
                        `PushButton(`id(`paul), "&Paul"  ),
                        `PushButton(`id(`george), "&George"),
                        `PushButton(`id(`ringo), "&Ringo" )),
                  `PushButton(`id(`ok), "&OK")
                  )
            );

    // Wait for user input.
    any button = nil;

    // Input loop that only the OK button will leave.
    // The 4 Beatles buttons will just propose a name.
    repeat
    {
        button = UI::UserInput();

        if      ( button == `john ) UI::ChangeWidget(`id(`name), `Value, "John Lennon");
        else if ( button == `paul ) UI::ChangeWidget(`id(`name), `Value, "Paul McCartney");
        else if ( button == `george ) UI::ChangeWidget(`id(`name), `Value, "George Harrison");
        else if ( button == `ringo ) UI::ChangeWidget(`id(`name), `Value, "Ringo Starr" );

    } until ( button == `ok );

    UI::CloseDialog();
}
```



```
        {
    // Build dialog with one text entry field and an OK button.
    UI::OpenDialog(
               `VBox(
                     `TextEntry(`id(`hex_digits), "Hex number:"),
                     `PushButton("&OK")
                     )
               );

    UI::ChangeWidget(`id(`hex_digits), `ValidChars, "0123456789abcdefABCDEF" );

    // Wait for user input.
    UI::UserInput();

    // Get the input from the text entry field.
    //
    // Notice: The return value of UI::UserInput() does NOT return this value!
    // Rather, it returns the ID of the widget (normally the PushButton)
    // that caused UI::UserInput() to return.
    string name = (string) UI::QueryWidget(`id(`hex_digits), `Value);

    // Close the dialog.
    // Remember to read values from the dialog's widgets BEFORE closing it!
    UI::CloseDialog();

    // Pop up a new dialog to echo the input.
    UI::OpenDialog(
               `VBox(
                     `Label("You entered:"),
                     `Label(name),
                     `PushButton("&OK")
                     )
               );
    UI::UserInput();
    UI::CloseDialog();
}
```



```
        {
    // Build dialog with one text entry field and an OK button.
    UI::OpenDialog(
               `VBox(
                     `TextEntry(`id(`hex_digits), "Hex number:"),
                     `PushButton("&OK")
                     )
               );

    UI::ChangeWidget(`id(`hex_digits), `ValidChars, "0123456789abcdefABCDEF" );

    // Wait for user input.
    UI::UserInput();

    // Get the input from the text entry field.
    //
    // Notice: The return value of UI::UserInput() does NOT return this value!
    // Rather, it returns the ID of the widget (normally the PushButton)
    // that caused UI::UserInput() to return.
    string name = (string) UI::QueryWidget(`id(`hex_digits), `Value);

    // Close the dialog.
    // Remember to read values from the dialog's widgets BEFORE closing it!
    UI::CloseDialog();

    // Pop up a new dialog to echo the input.
    UI::OpenDialog(
               `VBox(
                     `Label("You entered:"),
                     `Label(name),
                     `PushButton("&OK")
                     )
               );
    UI::UserInput();
    UI::CloseDialog();
}
```



```
          {
    // Build dialog with two password fields, an "OK" and a "Cancel" button.
    UI::OpenDialog(
               `VBox(
                     `Password(`id(`pw1), "Enter password:"),
                     `Password(`id(`pw2), "Confirm password:"),
                     `HBox(
                           `PushButton(`id(`ok),     "&OK"     ),
                           `PushButton(`id(`cancel), "&Cancel")
                           )
                     )
               );
    any button = nil;
    string pw1 = "";
    string pw2 = "";


    // Input loop. Will be terminated when the same password has been
    // entered in both fields or when 'Cancel' has been clicked.
    repeat
    {
        // Wait for Input.
        button = UI::UserInput();

        // Get the values from both password fields.
        pw1 = (string) UI::QueryWidget(`id(`pw1), `Value );
        pw2 = (string) UI::QueryWidget(`id(`pw2), `Value );

        if ( button != `cancel )
        {
            if ( pw1 == "" && pw2 == "" )
            {
                // Error popup if nothing has been entered.
                UI::OpenDialog(
                           `VBox(
                                 `Label("You must enter a password."),
                                 `PushButton("&OK")
                                 )
                           );
                UI::UserInput();
                UI::CloseDialog();
            }
            else if ( pw1 != pw2 )
            {
                // Error popup if passwords differ.
                UI::OpenDialog(
                           `VBox(
                                 `Label("The two passwords mismatch."),
                                 `Label("Please try again."),
                                 `PushButton("&OK")
                                 )
                           );
                UI::UserInput();
                UI::CloseDialog();
            }
        }
    } until ( ( pw1 != "" && pw1 == pw2 ) ||
              button == `cancel );

    UI::CloseDialog();
}
```

```
          {
    // Build dialog with two password fields, an "OK" and a "Cancel" button.
    UI::OpenDialog(
               `VBox(
                     `Password(`id(`pw1), "Enter password:"),
                     `Password(`id(`pw2), "Confirm password:"),
                     `HBox(
                           `PushButton(`id(`ok),     "&OK"    ),
                           `PushButton(`id(`cancel), "&Cancel")
                           )
                     )
               );
    any button = nil;
    string pw1 = "";
    string pw2 = "";


    // Input loop. Will be terminated when the same password has been
    // entered in both fields or when 'Cancel' has been clicked.
    repeat
    {
        // Wait for Input.
        button = UI::UserInput();

        // Get the values from both password fields.
        pw1 = (string) UI::QueryWidget(`id(`pw1), `Value );
        pw2 = (string) UI::QueryWidget(`id(`pw2), `Value );

        if ( button != `cancel )
        {
            if ( pw1 == "" && pw2 == "" )
            {
                // Error popup if nothing has been entered.
                UI::OpenDialog(
                           `VBox(
                                 `Label("You must enter a password."),
                                 `PushButton("&OK")
                                 )
                           );
                UI::UserInput();
                UI::CloseDialog();
            }
            else if ( pw1 != pw2 )
            {
                // Error popup if passwords differ.
                UI::OpenDialog(
                           `VBox(
                                 `Label("The two passwords mismatch."),
                                 `Label("Please try again."),
                                 `PushButton("&OK")
                                 )
                           );
                UI::UserInput();
                UI::CloseDialog();
            }
        }
    } until ( ( pw1 != "" && pw1 == pw2 ) ||
              button == `cancel );

    UI::CloseDialog();
}
```

Name

MultiLineEdit -- multiple line text edit field

MultiLineEdit

```
MultiLineEdit (label, initialText);
string label ;
string initialText ;
```

Parameters

string *label*	label above the field

Optional Arguments

string *initialText*	the initial contents of the field

Properties

string *Value*	The text contents as one large string containing newlines.
string *Label*	The label above the log text.

Description

This widget is a multiple line text entry field with a label above it. An initial text can be provided.

Note

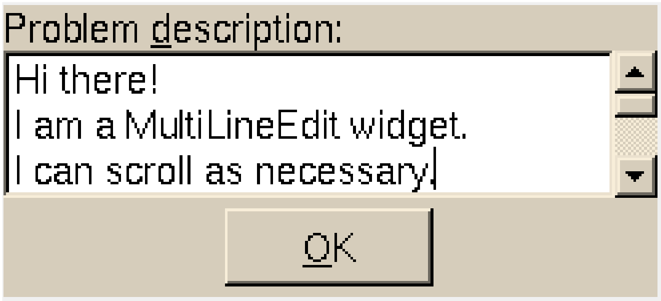
You can and should set a keyboard shortcut within the label. When the user presses the hotkey, the corresponding MultiLineEdit widget will get the keyboard focus.

Usage

```
	`MultiLineEdit( `id( `descr ), "Enter problem &amp; description:", "No problem here." )
```

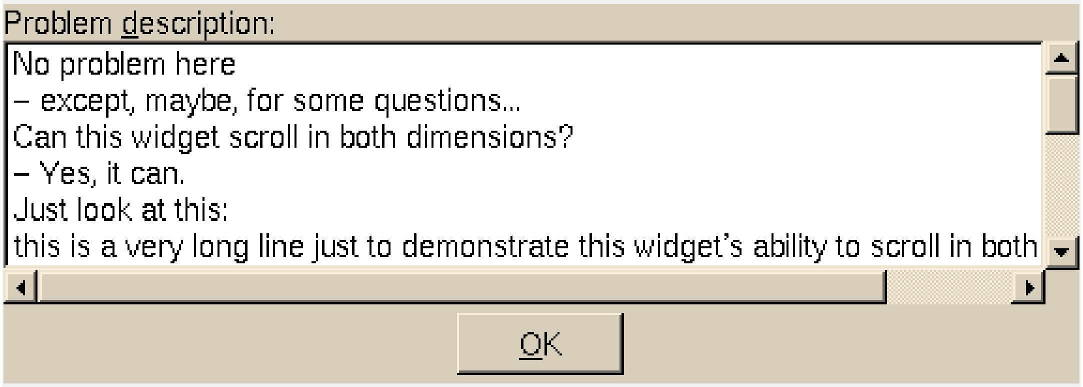
Examples

```
         {
    UI::OpenDialog(
                `VBox(
                      `MultiLineEdit("Problem &description:"),
                      `PushButton("&OK")
                      )
                );
    UI::UserInput();
    UI::CloseDialog();
}
```



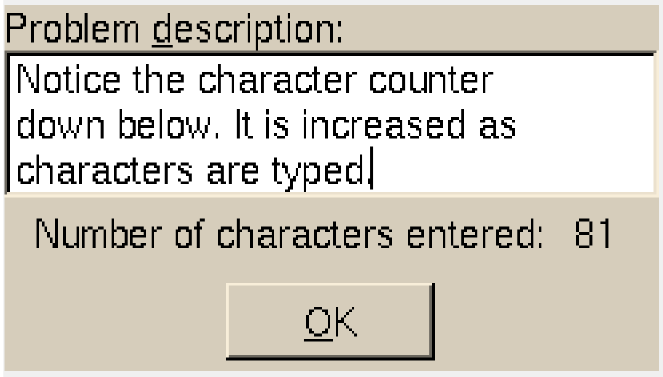
Hi there!
I am a MultiLineEdit widget.
I can scroll as necessary.

```
        {
    // Build dialog with one multi line edit field and an OK button.
    UI::OpenDialog(
               `VBox(
                     `HSpacing(60),         // force width
                     `HBox(
                           `VSpacing(7), // force height
                           `MultiLineEdit(`id(`problem),
                                          "Problem &description:", // label
                                          "No problem here")         // initial value
                           ),
                     `PushButton("&OK")
                     )
               );

    // Wait for user input.
    UI::UserInput();

    // Get the input from the MultiLineEdit.
    //
    // Notice: The return value of UI::UserInput() does NOT return this value!
    // Rather, it returns the ID of the widget (normally the PushButton)
    // that caused UI::UserInput() to return.
    string input = (string) UI::QueryWidget(`id(`problem), `Value);

    // Close the dialog.
    // Remember to read values from the dialog's widgets BEFORE closing it!
    UI::CloseDialog();

    // Pop up a new dialog to echo the input.
    UI::OpenDialog(
               `VBox(
                     `Label("You entered:"),
                     `Label(input),
                     `PushButton("&OK")
                     )
               );
    UI::UserInput();
    UI::CloseDialog();
}
```



No problem here
– except, maybe, for some questions...
Can this widget scroll in both dimensions?
– Yes, it can.
Just look at this:
this is a very long line just to demonstrate this widget's ability to scroll in both

```
        {
    // Build dialog with MuliLineEdit widget,
    // a character counter for the MuliLineEdit's contents as they are typed
    // and an OK button.
    UI::OpenDialog(
               `VBox(
                     `MultiLineEdit(`id(`problem),
                                    `opt(`notify), // make UI::UserInput() return on every keystroke
                                    "Problem &description:" ),
                     `HBox(
                           `Label("Number of characters entered:"),
                           `Label(`id(`char_count), "0    ")
                           ),
                     `PushButton(`id(`ok), "&OK")
                     )
               );

    any ret = nil;

    do
    {
        // Wait for user input.
        //
        // Since the MultiLineEdit is in "notify" mode, it, too, will cause
        // UI::UserInput() to return upon any single character entered.
        ret = UI::UserInput();

        if ( ret == `problem ) // User typed some text
        {
            // Set the `char_count label to the number of characters entered
            // into the MultiLineEdit widget.

            UI::ChangeWidget(`id(`char_count), `Value,
                         sformat( "%1", size( (string) UI::QueryWidget(`id(`problem), `Value) ) ) );
        }
    } while ( ret != `ok );

    // Get the input from the MultiLineEdit.
    //
    // Notice: The return value of UI::UserInput() does NOT return this value!
    // Rather, it returns the ID of the widget (normally the PushButton)
    // that caused UI::UserInput() to return.
    string input = (string) UI::QueryWidget(`id(`problem), `Value);

    // Close the dialog.
    // Remember to read values from the dialog's widgets BEFORE closing it!
    UI::CloseDialog();

    // Pop up a new dialog to echo the input.
    UI::OpenDialog(
               `VBox(
                     `Label("You entered:"),
                     `Label(input),
                     `PushButton("&OK")
                     )
               );
    UI::UserInput();
    UI::CloseDialog();
}
```


Problem description:

Notice the character counter down below. It is increased as characters are typed.

Number of characters entered: 81

Name

SelectionBox -- Scrollable list selection

SelectionBox

```
SelectionBox (label, items);
string label ;
list items ;
```

Parameters

string *label*

Options

shrinkable
: make the widget very small

immediate
: make `notify trigger immediately when the selected item changes

Optional Arguments

list *items*	the items contained in the selection box

Properties

string *Label*	The label above the list describing what it is all about
string *CurrentItem*	The currently selected item or its ID, if it has one.

Description

A selection box offers the user to select an item out of a list. Each item has a label and an optional id. When constructing the list of items, you have two way of specifying an item. Either you give a plain string, in which case the string is used both for the id and the label of the item. Or you specify a term *`item( term id, string label )* or *`item(term id, string label, boolean selected)*, where you give an id of the form *`id(any v)* where you can store an aribtrary value as id. The third argument controls whether the item is the selected item.

Usage

```
        `SelectionBox( `id( `pizza ), "select your Pizza:", [ "Margarita", `item( `id( `na ), "Napoli" ) ] )
```

Examples

```
         {
   UI::OpenDialog(
              `VBox(
                    `SelectionBox( "Select your Pizza:",
                                   [
                                      "Napoli",
                                      "Funghi",
                                      "Salami"
                                    ] ),
                    `PushButton("&OK")
                    )
              );
   UI::UserInput();
   UI::CloseDialog();
}
```

```
          {
    // Create a selection box with three entries.
    // All entries have IDs to identify them independent of locale
    // (The texts might have to be translated!).
    // Entry "Funghi" will be selected by default.
    UI::OpenDialog(
            `VBox(
                  `SelectionBox(`id(`pizza),
                                 "Select your Pizza:",
                                 [
                                  `item(`id(`nap), "Napoli"       ),
                                  `item(`id(`fun), "Funghi", true ),
                                  `item(`id(`sal), "Salami"       )
                                 ] ),
                  `PushButton("&OK")
                  )
            );
    UI::UserInput();

    // Get the input from the selection box.
    //
    // Notice: The return value of UI::UserInput() does NOT return this value!
    // Rather, it returns the ID of the widget (normally the PushButton)
    // that caused UI::UserInput() to return.
    any pizza = UI::QueryWidget( `id(`pizza), `CurrentItem);

    // Close the dialog.
    // Remember to read values from the dialog's widgets BEFORE closing it!
    UI::CloseDialog();

    // Evaluate selection

    string toppings = "nothing";

    if      ( pizza == `nap ) toppings = "Tomatoes, Cheese";
    else if ( pizza == `fun ) toppings = "Tomatoes, Cheese, Mushrooms";
    else if ( pizza == `sal ) toppings = "Tomatoes, Cheese, Sausage";

    // Pop up a new dialog to echo the selection.
    UI::OpenDialog(
            `VBox(
                  `Label("You will get a pizza with:"),
                  `Label(toppings),
                  `PushButton("&OK")
                  )
            );
    UI::UserInput();
    UI::CloseDialog();
}
```

```
          {
    // Create a selection box with three entries.
    // All entries have IDs to identify them independent of locale
    // (The texts might have to be translated!).
    // Entry "Funghi" will be selected by default.
    //
    // There are two buttons to select a "Today's special" and a
    // "veggie" pizza to demonstrate how to select list entries
    // programmatically.
    UI::OpenDialog(
            `VBox(
                  `SelectionBox(`id(`pizza),
                                "Select your Pizza:",
                                [
                                 `item(`id(`nap), "Napoli"       ),
                                 `item(`id(`fun), "Funghi", true ),
                                 `item(`id(`sal), "Salami"       )
                                ] ),
                  `HBox(
                        `PushButton(`id(`todays_special), `opt(`hstretch), "&Today's special" ),
                        `PushButton(`id(`veggie),         `opt(`hstretch), "&Veggie" )
                        ),
                  `PushButton(`id(`ok), `opt(`default), "&OK")
                  )
            );

    any id = nil;

    repeat
        {
            id = UI::UserInput();

            if      ( id == `todays_special ) UI::ChangeWidget( `id( `pizza ), `CurrentItem, `nap );
            else if ( id == `veggie         ) UI::ChangeWidget( `id( `pizza ), `CurrentItem, `fun );
        } until ( id == `ok );

    // Get the input from the selection box.
    //
    // Notice: The return value of UI::UserInput() does NOT return this value!
    // Rather, it returns the ID of the widget (normally the PushButton)
    // that caused UI::UserInput() to return.
    any pizza = UI::QueryWidget(`id(`pizza), `CurrentItem);

    // Close the dialog.
    // Remember to read values from the dialog's widgets BEFORE closing it!
    UI::CloseDialog();

    // Evaluate selection

    string toppings = "nothing";

    if      ( pizza == `nap ) toppings = "Tomatoes, Cheese";
    else if ( pizza == `fun ) toppings = "Tomatoes, Cheese, Mushrooms";
    else if ( pizza == `sal ) toppings = "Tomatoes, Cheese, Sausage";

    // Pop up a new dialog to echo the selection.
    UI::OpenDialog(
               `VBox(
                     `Label("You will get a pizza with:"),
                     `Label(toppings),
                     `PushButton(`opt(`default), "&OK")
                     )
               );
    UI::UserInput();

    UI::CloseDialog();
}
```

```
          {
    // Create a selection box with three entries.
    // All entries have IDs to identify them independent of locale
    // (The texts might have to be translated!).
    // Entry "Funghi" will be selected by default.
    //
    // There are two buttons to select a "Today's special" and a
    // "veggie" pizza to demonstrate how to select list entries
    // from within a YCP script - even without having to use item IDs.
    UI::OpenDialog(
               `VBox(
                     `SelectionBox(`id(`pizza),
                                   "Select your Pizza:",
                                   [
                                    "Napoli",
                                    "Funghi",
                                    "Salami",
                                    "Quattro Stagioni (a pizza which is devided into 4 parts each with a diff
                                    "Caprese",
                                    "Speciale",
                                    "Hawaii"
                                   ] ),
                     `HBox(
                           `PushButton(`id(`todays_special), `opt(`hstretch), "&Today's special" ),
                           `PushButton(`id(`veggie),         `opt(`hstretch), "&Veggie" )
                           ),
                     `PushButton(`id(`ok), `opt(`default), "&OK")
                     )
               );

    any id = nil;

    repeat
        {
            id = UI::UserInput();

            if      ( id == `todays_special ) UI::ChangeWidget( `id( `pizza ), `CurrentItem, "Napoli" );
            else if ( id == `veggie         ) UI::ChangeWidget( `id( `pizza ), `CurrentItem, "Funghi" );
        } until ( id == `ok );

    // Get the input from the selection box.
    //
    // Notice: The return value of UI::UserInput() does NOT return this value!
    // Rather, it returns the ID of the widget (normally the PushButton)
    // that caused UI::UserInput() to return.
    string pizza = (string) UI::QueryWidget(`id(`pizza), `CurrentItem);

    // Close the dialog.
    // Remember to read values from the dialog's widgets BEFORE closing it!
    UI::CloseDialog();


    // Pop up a new dialog to echo the selection.
    UI::OpenDialog(
               `VBox(
                     `Label("Pizza " + pizza + " coming right up"),
                     `PushButton(`opt(`default), "&OK")
                     )
               );
    UI::UserInput();

    UI::CloseDialog();
}
```

Name

MultiSelectionBox -- Selection box that allows selecton of multiple items

MultiSelectionBox

```
MultiSelectionBox (label, items);
string label ;
list items ;
```

Parameters

string *label*	
list *items*	the items initially contained in the selection box

Options

shrinkable	make the widget very small

Properties

string *Label*	The label above the list describing what it is all about
string *CurrentItem*	The item that currently has the keyboard focus
id_list *SelectedItems*	The items that are currently selected

Description

The MultiSelectionBox displays a (scrollable) list of items from which any number (even nothing!) can be selected. Use the MultiSelectionBox's `SelectedItems` property to find out which.

Each item can be specified either as a simple string or as `` `item( ... ) `` which includes an (optional) ID and an (optional) 'selected' flag that specifies the initial selected state ('not selected', i.e. 'false', is default).

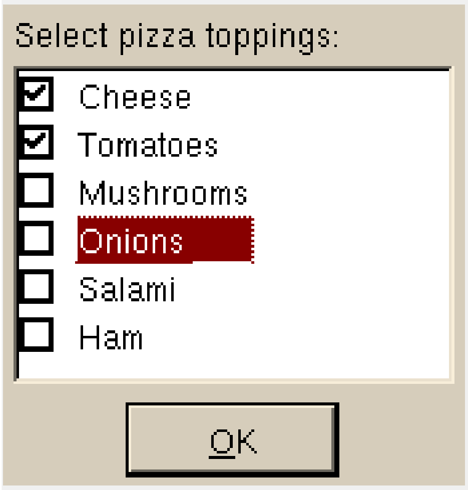
Usage

```
        `MultiSelectionBox( `id( `topping ), "select pizza toppings:", [ "Salami", `item( `id( `cheese ), "C
```

Examples

```
        {
    // Simple MultiSelectionBox example:
    //
    // All items are simple strings, none has an ID, no item preselected.

    UI::OpenDialog(
               `VBox(
                     `MultiSelectionBox( "Select pizza toppings:",
                                         [
                                          "Cheese",
                                          "Tomatoes",
                                          "Mushrooms",
                                          "Onions",
                                          "Salami",
                                          "Ham"
                                         ] ),
                     `PushButton("&OK")
                     )
               );
    UI::UserInput();
    UI::CloseDialog();
}
```

```
          {
    // More realistic MultiSelectionBox example:
    //
    // Items have IDs, some are preselected.
    // Notice 'false' is default anyway for the selection state,
    // so you may or may not explicitly specify that.

    UI::OpenDialog(
               `VBox(
                     `MultiSelectionBox( "Select pizza toppings:",
                                         [
                                          `item( `id(`cheese           ), "Cheese"   , true  ),
                                          `item( `id(`tomatoes         ), "Tomatoes" , true  ),
                                          `item( `id(`mush             ), "Mushrooms" , false ),
                                          `item( `id(`onions ), "Onions"          ),
                                          `item( `id(`sausage ), "Salami"          ),
                                          `item( `id(`pork ), "Ham" )
                                         ] ),
                     `PushButton( `opt(`default), "&OK")
                     )
               );
    UI::UserInput();
    UI::CloseDialog();
}
```



```
          {
    // Advanced MultiSelectionBox example:
    //
    // Retrieve the list of selected items and output it.

    UI::OpenDialog(
               `VBox(
                     `MultiSelectionBox( `id(`toppings), "Select pizza toppings:",
                                         [
                                          `item( `id(`cheese           ), "Cheese"   , true  ),
                                          `item( `id(`tomatoes         ), "Tomatoes" , true  ),
                                          `item( `id(`mush             ), "Mushrooms" , false ),
                                          `item( `id(`onions ), "Onions"          ),
                                          `item( `id(`sausage ), "Salami"          ),
                                          `item( `id(`pork ), "Ham" )
                                         ] ),
                     `PushButton( `opt(`default), "&OK")
```

```
                    )
              );
    UI::UserInput();
    list selected_items = (list) UI::QueryWidget( `id(`toppings), `SelectedItems );

    // Remember to retrieve the widget's data _before_ the dialog is closed,
    // i.e. before it is destroyed!

    UI::CloseDialog();



    // Concatenate the list of selected toppings to one multi-line string.

    string pizza_description = "";

    foreach ( `topping, selected_items, ``{
        pizza_description = sformat( "%1\n%2", pizza_description, topping );
    } );


    // Open a new dialog to echo the selection.

    UI::OpenDialog(
               `VBox(
                     `Label( "Your pizza will come with:\n" ),
                     `Label( pizza_description ),
                     `PushButton( `opt(`default), "&OK" )
                     )
              );
    UI::UserInput();
    UI::CloseDialog();
}
```

Name

ComboBox -- drop-down list selection (optionally editable)

ComboBox

```
ComboBox (label, items);
string label ;
list items ;
```

Parameters

string *label*

Options

editable	the user can enter any value.

Optional Arguments

list *items*	the items contained in the combo box

Properties

string *Label*	The label above the ComboBox describing what it is all about
string *Value*	The current value of the ComboBox. If this is a list item which has an ID, the ID will be used rather than the corresponding text. For editable ComboBox widgets it is OK to set *any* value - even if it doesn't exist in the list.
string *Valid-Chars*	valid input characters

Description

A combo box is a combination of a selection box and an input field. It gives the user a one-out-of-many choice from a list of items. Each item has a (mandatory) label and an (optional) id. When the 'editable' option is set, the user can also freely enter any value. By default, the user can only select an item already present in the list.

The items are very much like SelectionBox items: They can have an (optional) ID, they have a mandatory text to be displayed and an optional boolean parameter indicating the selected state. Only one of the items may have this parameter set to "true"; this will be the default selection on startup.

Note

You can and should set a keyboard shortcut within the label. When the user presses the hotkey, the combo box will get the keyboard focus.

Usage

```
        `ComboBox( `id( `pizza ), "select your Pizza:", [ "Margarita", `item( `id( `na ), "Napoli" ) ] )
```

Examples

```
          {
    UI::OpenDialog(
               `VBox(
                     `ComboBox( "Select your Pizza:",
                                [
                                 "Napoli",
                                 "Funghi",
                                 "Salami"
                                ] ),
                     `PushButton("&OK")
                     )
               );
    UI::UserInput();
    UI::CloseDialog();
}
```

```
          // Create a combo box with three entries.
// All entries have IDs to identify them independent of locale
// (The texts might have to be translated!).
// Entry "Funghi" will be selected by default.
{
    UI::OpenDialog(
               `VBox(
                     `ComboBox(`id(`pizza),
                                    "Select your Pizza:",
                                    [
                                     `item(`id(`nap), "Napoli"       ),
                                     `item(`id(`fun), "Funghi", true ),
                                     `item(`id(`sal), "Salami"       )
                                    ] ),
                     `PushButton("&OK")
                     )
               );
    UI::UserInput();

    // Get the input from the selection box.
    //
    // Notice: The return value of UI::UserInput() does NOT return this value!
    // Rather, it returns the ID of the widget (normally the PushButton)
    // that caused UI::UserInput() to return.
    any pizza=UI::QueryWidget(`id(`pizza), `Value);
    y2milestone( "Selected %1", pizza );

    // Close the dialog.
    // Remember to read values from the dialog's widgets BEFORE closing it!
    UI::CloseDialog();

    // Evaluate selection

    string toppings = "nothing";

    if      ( pizza == `nap ) toppings = "Tomatoes, Cheese";
    else if ( pizza == `fun ) toppings = "Tomatoes, Cheese, Mushrooms";
    else if ( pizza == `sal ) toppings = "Tomatoes, Cheese, Sausage";

    // Pop up a new dialog to echo the selection.
    UI::OpenDialog(
               `VBox(
                     `Label("You will get a pizza with:"),
                     `Label(toppings),
                     `PushButton("&OK")
                     )
               );
    UI::UserInput();
    UI::CloseDialog();
}
```

```
          // Create a combo box with three entries.
// All entries have IDs to identify them independent of locale
// (The texts might have to be translated!).
// Entry "Funghi" will be selected by default.
{
    UI::OpenDialog(
               `VBox(
                     `ComboBox(`id(`pizza), `opt(`editable),
                                    "Select your Pizza:",
                                    [
                                     `item(`id(`nap), "Napoli"       ),
                                     `item(`id(`fun), "Funghi", true ),
                                     `item(`id(`sal), "Salami"       )
                                    ] ),
                     `PushButton("&OK")
                     )
               );
    UI::UserInput();

    // Get the input from the selection box.
    //
    // Notice: The return value of UI::UserInput() does NOT return this value!
    // Rather, it returns the ID of the widget (normally the PushButton)
    // that caused UI::UserInput() to return.
    any pizza=UI::QueryWidget(`id(`pizza), `Value);
    y2milestone( "Selected %1", pizza );

    // Close the dialog.
    // Remember to read values from the dialog's widgets BEFORE closing it!
    UI::CloseDialog();

    // Evaluate selection

    string toppings = "nothing";

    if      ( pizza == `nap ) toppings = "Tomatoes, Cheese";
    else if ( pizza == `fun ) toppings = "Tomatoes, Cheese, Mushrooms";
    else if ( pizza == `sal ) toppings = "Tomatoes, Cheese, Sausage";

    // Pop up a new dialog to echo the selection.
    UI::OpenDialog(
               `VBox(
                     `Label("You will get a pizza with:"),
                     `Label(toppings),
                     `PushButton("&OK")
                     )
               );
    UI::UserInput();
    UI::CloseDialog();
}
```

```
          // Create an editable combo box with restricted input character set.
{
    UI::OpenDialog(
               `VBox(
                     `ComboBox(`id(`addr), `opt(`editable),
                                    "Enter hex address:",
                                    [
                                     "0cff",
                                     "8080",
                                     "D0C0",
                                     "ffff"
                                    ] ),
                     `PushButton("&OK")
                     )
               );
    // Set the valid input characters.
    UI::ChangeWidget(`id(`addr), `ValidChars, "0123456789abcdefABCDEF" );


    // Wait for user input.
    UI::UserInput();


    // Get the input from the selection box.
    //
    // Notice: The return value of UI::UserInput() does NOT return this value!
    // Rather, it returns the ID of the widget (normally the PushButton)
    // that caused UI::UserInput() to return.
    any val=UI::QueryWidget(`id(`addr), `Value);
    y2milestone( "Selected %1", val );

    // Close the dialog.
    // Remember to read values from the dialog's widgets BEFORE closing it!
    UI::CloseDialog();


    // Pop up a new dialog to echo the input.
    UI::OpenDialog(
               `VBox(
                     `Label("You entered:"),
                     `Label(val),
                     `PushButton("&OK")
                     )
               );
    UI::UserInput();
    UI::CloseDialog();
}
```

Name

Tree -- Scrollable tree selection

Tree

```
Tree (label, items);
string label ;
itemList items ;
```

Parameters

string *label*

Optional Arguments

itemList *items*	the items contained in the tree

```
      itemList ::=
              [
                      item
                      [ , item ]
                      [ , item ]
                      ...
              ]
      item ::=
              string |
              `item(
                      [ `id( string  ),]
                      string
                      [ , true | false ]
                      [ , itemList ]
              )
```

The boolean parameter inside `item() indicates whether or not the respective tree item should be opened by default - if it has any subitems and if the respective UI is capable of closing and opening subtrees. If the UI cannot handle this, all subtrees will always be open.

Properties

string *Label*	the label above the Tree
itemId *CurrentItem*	the currently selected item
OpenItems *a*	map of open items - can only be queried, not set

Description

A tree widget provides a selection from a hierarchical tree structure. The semantics are very much like those of a SelectionBox. Unlike the SelectionBox, however, tree items may have subitems that in turn may have subitems etc.

Each item has a label string, optionally preceded by an ID. If the item has subitems, they are specified as a list of items after the string.

The tree widget will not perform any sorting on its own: The items are always sorted by insertion order. The application needs to handle sorting itself, if desired.

Usage

```
    `Tree( `id( `treeID ), "treeLabel", [ "top1", "top2", "top3" ] );
```

Examples

```
      {
UI::OpenDialog(
           `VBox(
                 `Tree(`id(`dest_dir),
                               "Select destination directory:",
                               [
                                `item(`id(`root), "/" , true,
                                      [
                                       `item(`id(`etc), "etc",
                                             [
                                              `item("opt"),
                                              `item("SuSEconfig"),
                                              `item("X11")
                                             ]
                                             ),
                                       `item("usr", false,
                                             [
                                              "bin",
                                              "lib",
                                              `item("share",
                                                   [
                                                    "man",
                                                    "info",
                                                    "emacs"
                                                    ]
                                                   ),
                                              `item(`id(`usr_local),"local"),
                                              `item("X11R6",
                                                    [
                                                     "bin",
                                                     "lib",
                                                     "share",
                                                     "man",
                                                     "etc"
                                                     ]
                                                    )
                                              ]
                                             ),
                                       `item(`id(`opt), "opt", true,
                                             [
                                              "kde",
                                              "netscape",
                                              "Office51"
                                              ]
                                             ),
                                       `item("home"),
                                       "work",
                                       `item(`id(`other), "<other>")
                                       ]
                                      )
                               ] ),
                 `HBox(
                       `PushButton(`id(`sel_opt),         `opt(`hstretch), "/&opt" ),
                       `PushButton(`id(`sel_usr),         `opt(`hstretch), "/&usr" ),
                       `PushButton(`id(`sel_usr_local), `opt(`hstretch), "/usr/&local" )
                       ),
                 `PushButton(`id(`ok), `opt(`default), "&OK")
                 )
           );

any id = nil;

repeat
    {
        id = UI::UserInput();

        if      ( id == `sel_usr)         UI::ChangeWidget( `id( `dest_dir ), `CurrentItem, "usr"  );
        else if ( id == `sel_usr_local) UI::ChangeWidget( `id( `dest_dir ), `CurrentItem, `usr_local );
        else if ( id == `sel_opt)                 UI::ChangeWidget( `id( `dest_dir ), `CurrentItem, `opt );
    } until ( id == `ok );

// Get the input from the tree.
//
// Notice: The return value of UI::UserInput() does NOT return this value!
// Rather, it returns the ID of the widget (normally the PushButton)
// that caused UI::UserInput() to return.
string dest_dir = (string) UI::QueryWidget(`id(`dest_dir), `CurrentItem);

if ( dest_dir == nil )
    dest_dir = "";
    // Close the dialog.
    // Remember to read values from the dialog's widgets BEFORE closing it!
    UI::CloseDialog();


    // Pop up a new dialog to echo the selection.
    UI::OpenDialog(
                `VBox(
                      `Label("Selected destination directory: " + dest_dir),
                      `PushButton(`opt(`default), "&OK")
                      )
                );
    UI::UserInput();

    UI::CloseDialog();
}
```

```
          {
    // Build a dialog with a tree for directory selection, three
    // buttons with common values and a label that directly echoes any
    // selected directory.
    //
    // The tree in this example uses the `notify option that makes
    // UI::UserInput() return immediately as soon as the user selects a
    // tree item rather than the default behaviour which waits for the
    // user to activate a button.

    UI::OpenDialog(
                `VBox(
                      `Tree(`id(`dest_dir),
                            `opt(`notify),
                                    "Select destination directory:",
                                    [
                                     `item(`id(`root), "/" , true,
                                           [
                                            `item(`id(`etc), "etc",
                                                  [
                                                   `item("opt"),
                                                   `item("SuSEconfig"),
                                                   `item("X11")
                                                  ]
                                                  ),
                                            `item("usr", false,
                                                  [
                                                   "bin",
                                                   "lib",
                                                   `item("share",
                                                         [
                                                          "man",
                                                          "info",
                                                          "emacs"
                                                          ]
                                                         ),
                                                   `item(`id(`usr_local),"local"),
                                                   `item("X11R6",
                                                         [
                                                          "bin",
                                                          "lib",
                                                          "share",
                                                          "man",
                                                          "etc"
                                                          ]
                                                         )
                                                   ]
                                                  ),
                                            `item(`id(`opt), "opt", true,
                                                  [
                                                   "kde",
                                                   "netscape",
                                                   "Office51"
                                                   ]
                                                  ),
                                            `item("home"),
                                            "work",
                                            `item(`id(`other), "<other>")
                                            ]
                                           )
                                    ] ),
                      `HBox(
                            `PushButton(`id(`sel_opt),         `opt(`hstretch), "/&opt" ),
                            `PushButton(`id(`sel_usr),         `opt(`hstretch), "/&usr" ),
                            `PushButton(`id(`sel_usr_local), `opt(`hstretch), "/usr/&local" )
                            ),
                      `HBox(
                            `Label("Current Value:"),
                            `Label(`id(`echo), `opt(`outputField), "?????????")
                            ),
                      `PushButton(`id(`ok), `opt(`default), "&OK")
                      )
                );
    any id = nil;

    repeat
        {
            id = UI::UserInput();

            if      ( id == `sel_usr)         UI::ChangeWidget( `id( `dest_dir ), `CurrentItem, "usr"  );
            else if ( id == `sel_usr_local) UI::ChangeWidget( `id( `dest_dir ), `CurrentItem, `usr_local );
            else if ( id == `sel_opt)                 UI::ChangeWidget( `id( `dest_dir ), `CurrentItem, `opt );
            else if ( id == `dest_dir)
            {
                UI::ChangeWidget( `id( `echo ), `Value, "hello");
                any current_dir = UI::QueryWidget(`id(`dest_dir), `CurrentItem);

                if ( current_dir != nil )
                    UI::ChangeWidget( `id( `echo ), `Value, sformat( "%1", current_dir ) );
            }

            y2milestone( "Items:\n%1", UI::QueryWidget(`dest_dir, `Items ) );
            y2milestone( "OpenItems:\n%1", UI::QueryWidget(`dest_dir, `OpenItems ) );
        } until ( id == `ok );


    // Close the dialog.
    // Remember to read values from the dialog's widgets BEFORE closing it!
    UI::CloseDialog();

}
```

Name

Table -- Multicolumn table widget

Table

```
Table (header, items);
term header ;
list items ;
```

Parameters

term *header*	the headers of the columns

Options

immediate	make `notify trigger immediately when the selected item changes
keepSorting	keep the insertion order - don't let the user sort manually by clicking

Optional Arguments

list *items*	the items contained in the selection box

Properties

integer *CurrentItem*	the ID of the currently selected item
list(item) *Items*	a list of all table items
item *Item(id)*	read: a single item (string or term)
integer|string *Item(id,column)*	write: replacement for one specific cell (see example)

Description

A Table widget is a generalization of the SelectionBox. Information is displayed in a number of columns. Each column has a header. The number of columns and their titles are described by the first argument, which is a term with the symbol `header`. For each column you add a string specifying its title. For example `` `header( "Name", "Price" ) `` creates the two columns "Name" and "Price".

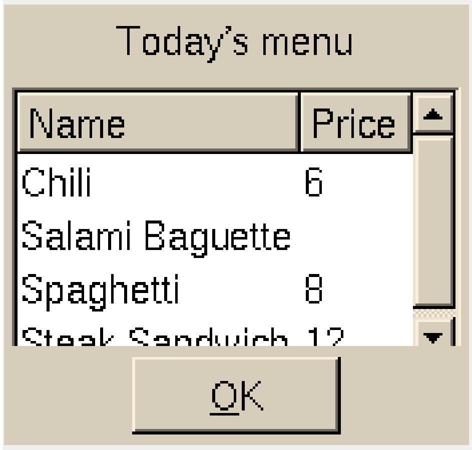
The second argument is an optional list of items (rows) that are inserted in the table. Each item has the form *`` `item( `id( ``* id *`), first column, second column, ... )`*. For each column one argument has to be specified, which must be of type void, string or integer. Strings are being left justified, integer right and a nil denote an empty cell, just as the empty string.

Usage

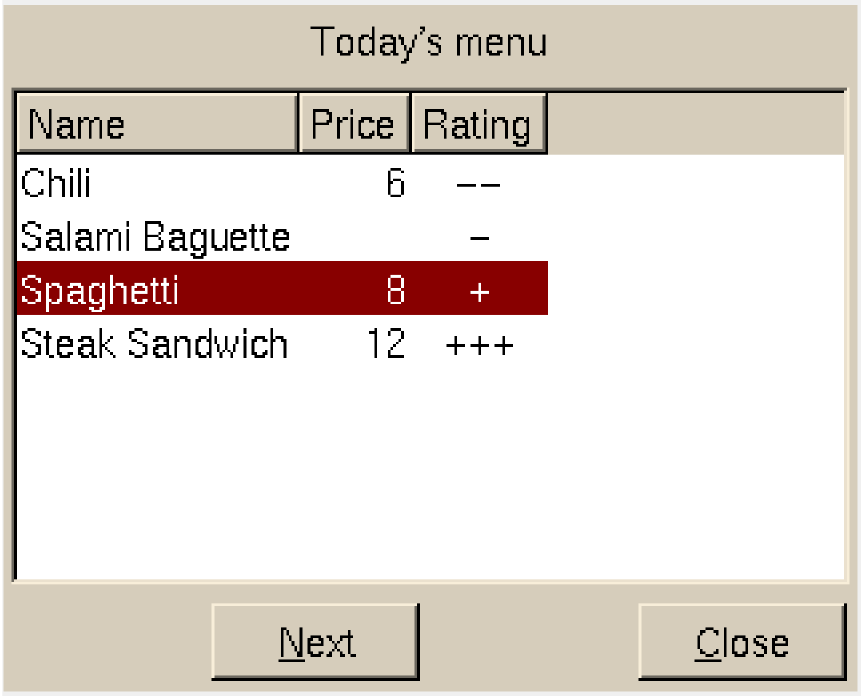
```
        `Table( `header( "Game", "Highscore" ), [ `item( `id(1), "xkobo", "1708" ) ] )
```

Examples

```
      {
    UI::OpenDialog(
               `VBox(
                     `Label("Today's menu"),
                     `Table(
                            `header("Name", "Price"),
                            [
                             `item(`id(1), "Chili",          6),
                             `item(`id(2), "Salami Baguette", nil),
                             `item(`id(3), "Spaghetti",          8),
                             `item(`id(4), "Steak Sandwich", 12)
                            ]
                            ),
                     `PushButton("&OK")
                     )
               );
    UI::UserInput();
    UI::CloseDialog();
}
```



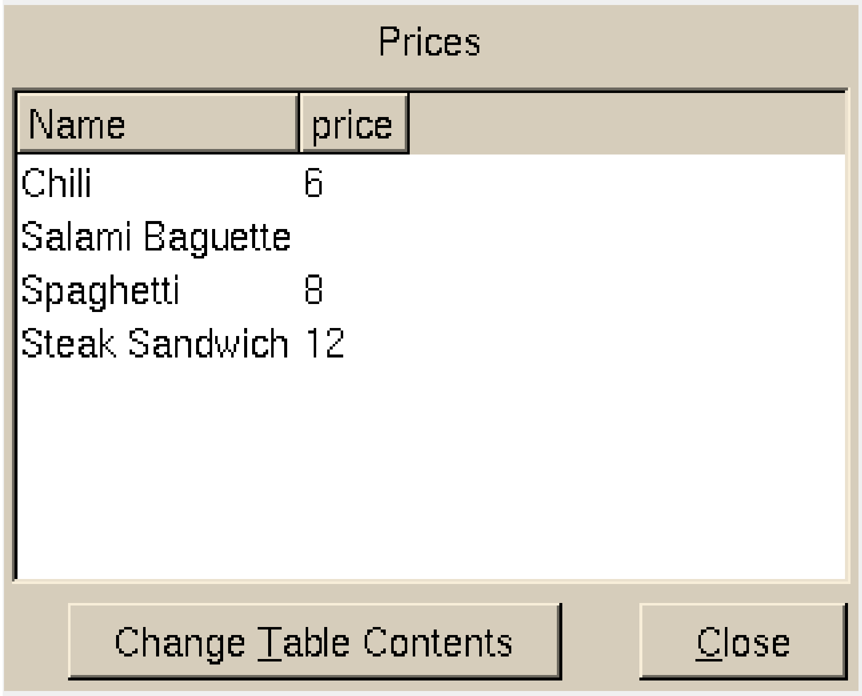
```
      {
    UI::OpenDialog(
               `VBox(
                     `Label("Today's menu"),
                     `HSpacing(40), // make the table and thus the dialog wider
                     `HBox(
                           `VSpacing(10), // make the table higher
                           `Table(
                                  `id(`table), `opt(`keepSorting),
                                  `header("Name", `Right("Price"), `Center("Rating")),
                                  [
                                   `item(`id(0), "Steak Sandwich", 12,   "+++"),
                                   `item(`id(1), "Salami Baguette", nil, "-"   ),
                                   `item(`id(2), "Chili",          6,    "--" ),
                                   `item(`id(3), "Spaghetti",          8,    "+"   )
                                  ]
                                  )
                           ),
                     `HBox(
                           `HCenter(`PushButton(`id(`next), "&Next")),
                           `PushButton(`id(`cancel), "&Close")
                           )
                     )
               );

    UI::ChangeWidget(`id(`table), `CurrentItem, 2);

    while (UI::UserInput() != `cancel)
    {
        UI::ChangeWidget(`id(`table), `CurrentItem,
                     ((integer) UI::QueryWidget(`id(`table), `CurrentItem) + 1) % 4);
    }
    UI::CloseDialog();
}
```



Today's menu

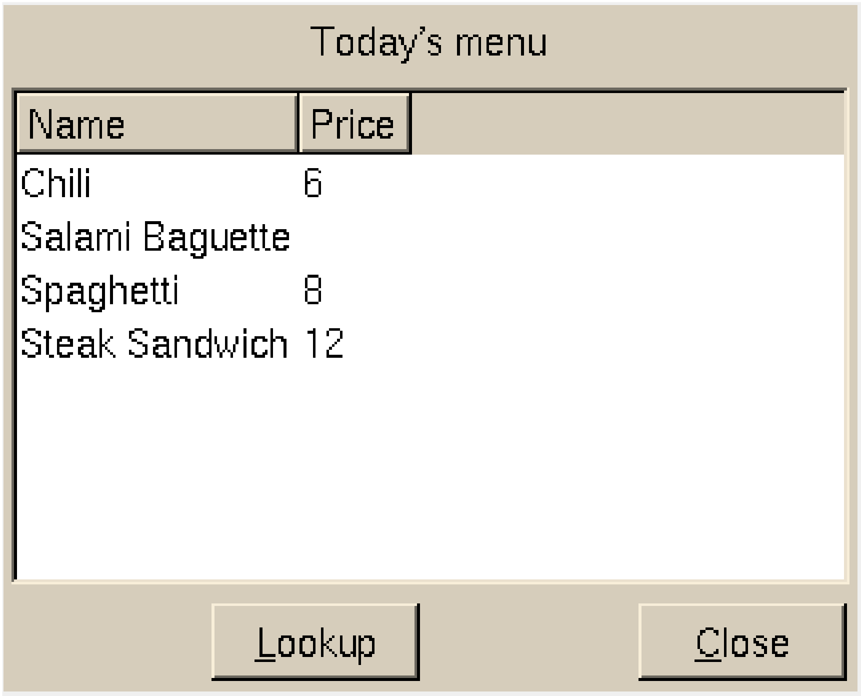
Name	Price	Rating
Chili	6	--
Salami Baguette		-
Spaghetti	8	+
Steak Sandwich	12	+++

```
      {
    list itemlist1 =
        [
         `item(`id(3), "Spaghetti",    8),
         `item(`id(4), "Steak Sandwich",  12),
         `item(`id(1), "Chili",            6),
         `item(`id(2), "Salami Baguette", nil)
        ];

    list itemlist2 =
        [
         `item(`id(0), "Mercedes", 60000),
         `item(`id(1), "AUDI",        50000),
         `item(`id(2), "VW",        40000),
         `item(`id(3), "BMW",         60000),
         `item(`id(3), "Porsche", 80000)
        ];

    list itemslists = [ itemlist1, itemlist2 ];

    integer listnum = 0;

    UI::OpenDialog(
               `VBox(
                     `Label("Prices"),
                     `HSpacing(40), // make the table and thus the dialog wide enough
                     `HBox(
                           `VSpacing(10),
                           `Table(`id(`table), `header("Name", "price"), itemlist1)
                           ),
                     `HBox(
                           `HCenter(`PushButton(`id(`next), "Change &Table Contents")),
                           `PushButton(`id(`cancel), "&Close")
                           )
                     )
               );

    while (UI::UserInput() != `cancel)
    {
        listnum = 1 - listnum;
        UI::ChangeWidget(`id(`table), `Items, select(itemslists, listnum, nil));
    }

    UI::CloseDialog();
}
```


Prices

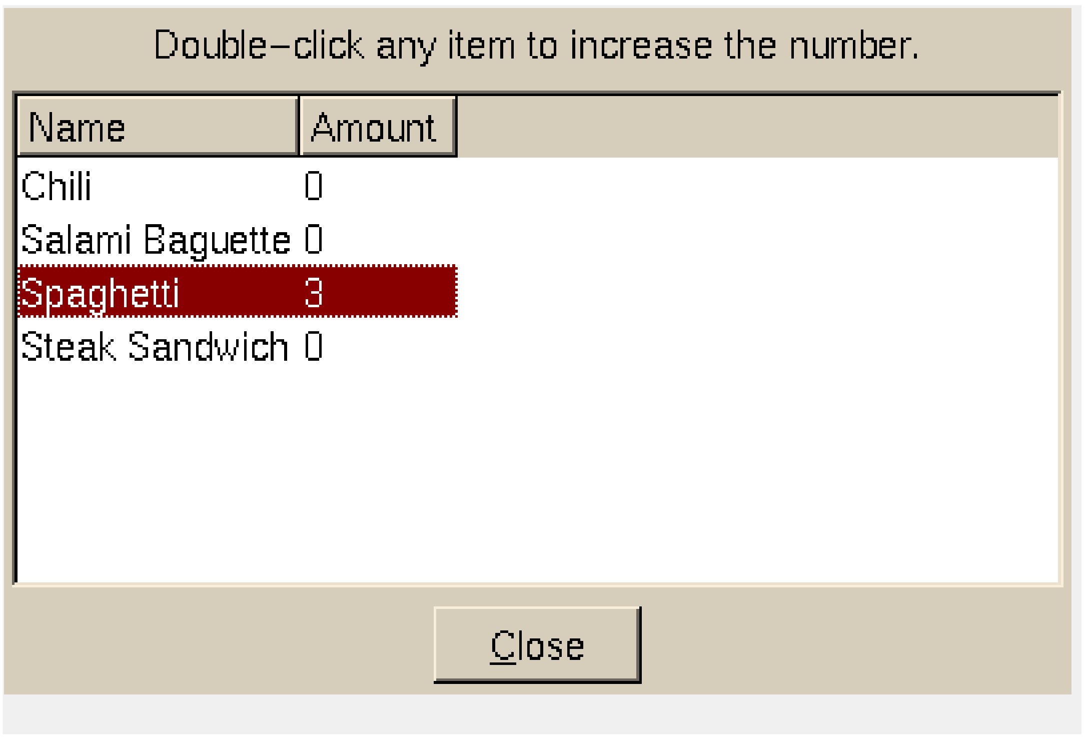
Name	price
Chili	6
Salami Baguette	
Spaghetti	8
Steak Sandwich	12

```
          {
    UI::OpenDialog(
               `VBox(
                     `Label("Today's menu"),
                     `HSpacing(40), // make the table and thus the dialog wider
                     `HBox(
                           `VSpacing(10), // make the table higher
                           `Table(`id(`table),
                                  `header("Name", "Price"),
                                  [
                                   `item(`id(1), "Chili",      6),
                                   `item(`id(2), "Salami Baguette", nil),
                                   `item(`id(3), "Spaghetti",      8),
                                   `item(`id(4), "Steak Sandwich",  12)
                                  ]
                                  )
                           ),
                     `HBox(
                           `HCenter(`PushButton("&Lookup")),
                           `PushButton(`id(`cancel), "&Close" )
                           )
                     )
               );

    while (UI::UserInput() != `cancel)
    {
        any id = UI::QueryWidget(`id(`table), `CurrentItem);
        if (is(id, integer)) {
            string text = sformat("Line: %1", UI::QueryWidget(`id(`table), `Item(id)));
            UI::OpenDialog(
                       `VBox(
                             `Label(text),
                             `PushButton("&OK")
                             )
                       );
            UI::UserInput();
            UI::CloseDialog();
        }
    }

    UI::CloseDialog();
}
```



Name	Price
Chili	6
Salami Baguette	
Spaghetti	8
Steak Sandwich	12

```
          {
    UI::OpenDialog(
                `VBox(
                      `HSpacing(50), // make the dialog wider
                      `Label("Double-click any item to increase the number."),
                      `HBox(
                            `VSpacing(10), // make the table higher
                            `Table(`id(`table),
                                   `opt(`notify),
                                   `header("Name", "Amount"),
                                   [
                                    `item(`id(1), "Chili",         0),
                                    `item(`id(2), "Salami Baguette", 0),
                                    `item(`id(3), "Spaghetti",         0),
                                    `item(`id(4), "Steak Sandwich", 0)
                                   ]
                                   )
                            ),
                      `PushButton(`id(`cancel), "&Close")
                      )
                );

    while ( UI::UserInput() != `cancel)
    {
        integer current_item  = (integer) UI::QueryWidget(`id(`table), `CurrentItem);
        integer current_value = tointeger(select((list) UI::QueryWidget(`id(`table), `Item(current_item)), 2,
        UI::ChangeWidget(`id(`table), `Item(current_item, 1), current_value+1);
    }

    UI::CloseDialog();
}
```



Double-click any item to increase the number.

Name	Amount
Chili	0
Salami Baguette	0
Spaghetti	3
Steak Sandwich	0

Name

ProgressBar -- Graphical progress indicator

ProgressBar

```
ProgressBar (label, maxvalue, progress);
string label ;
integer maxvalue ;
integer progress ;
```

Parameters

string *label*	the label describing the bar

Optional Arguments

integer *maxvalue*	the maximum value of the bar
integer *progress*	the current progress value of the bar

Properties

integer *Value*	the current progress
string *Label*	the label above the progress bar

Description

A progress bar is a horizontal bar with a label that shows a progress value. If you omit the optional parameter *maxvalue*, the maximum value will be 100. If you omit the optional parameter *progress*, the progress bar will set to 0 initially.

Usage

```
          `ProgressBar( `id( `pb ), "17 of 42 Packages installed", 42, 17 )
```

Examples

```
          // Simple ProgressBar example
{
    integer max_progress = 7;
    integer progress = 0;

    UI::OpenDialog(
                   `VBox(
                         `ProgressBar(`id(`pr), "Sample progress bar", max_progress, progress ),
                         `PushButton(`id(`next), "Next"),
                         `Right(`PushButton(`id(`close), "&Close" ) )
                         )
                   );


    while ( progress < max_progress )
    {
        symbol button = (symbol) UI::UserInput();
        if ( button == `next )
        {
            progress = progress + 1;
            UI::ChangeWidget(`id(`pr), `Value, progress);
            UI::ChangeWidget(`id(`pr), `Label, sformat("Progress %1 of %2", progress, max_progress ));
        }
        else if ( button == `close )
            break;
    }

    UI::CloseDialog();
}
```

```
          // ProgressBar example
{
    UI::OpenDialog(
                   `VBox(
                         `Heading("Adjust the volume"),
                         `ProgressBar(`id(`vol), "Volume", 100, 50),
                         `HBox(
                               `PushButton(`id(`down), "<<"),
                               `PushButton(`id(`up  ), ">>"),
                               `HStretch(),
                               `HSpacing(3),
                               `PushButton(`id(`cancel), "&Close" )
                               )
                         )
                   );

    while (true)
    {
        symbol button = (symbol) UI::UserInput();

        if (button == `cancel)
            break;
        else if ( button == `down || button == `up )
        {
            integer volume = (integer) UI::QueryWidget(`id(`vol), `Value);

            if ( button == `down ) volume = volume - 5;
            if ( button == `up  ) volume = volume + 5;

            y2milestone( "Volume: %1", volume );
            UI::ChangeWidget(`id(`vol), `Value, volume );
        }
    }

    UI::CloseDialog();
}
```

Name

Image -- Pixmap image

Image

```
Image (image, label);
symbol|byteblock|string image ;
string label ;
```

Parameters

symbol|byteblock|string *image*	specification which image to display
string *label*	label or default text of the image

Options

tiled	tile pixmap: repeat it as often as needed to fill all available space
scaleToFit	scale the pixmap so it fits the available space: zoom in or out as needed
zeroWidth	make widget report a nice width of 0
zeroHeight	make widget report a nice height of 0
animated	image data contain an animated image (e.g. MNG)

Description

Displays an image - if the respective UI is capable of that. If not, it is up to the UI to decide whether or not to display the specified default text instead (e.g. with the NCurses UI).

Use *`opt( `zeroWidth)* and / or *`opt( `zeroHeight)* if the real size of the image widget is determined by outside factors, e.g. by the size of neighboring widgets. With those options you can override the default "nice size" of the image widget and make it show just a part of the image. This is used for example in the YaST2 title graphics that are 2000 pixels wide even when only 640 pixels are shown normally. If more screen space is available, more of the image is shown, if not, the layout engine doesn't complain about the image widget not getting its nice size.

`opt( `tiled) will make the image repeat endlessly in both dimensions to fill up any available space. You might want to add `opt( `zeroWidth) or `opt( `zeroHeight) (or both), too to make use of this feature.

`opt( `scaleToFit) scales the image to fit into the available space, i.e. the image will be zoomed in or out as needed.

This option implicitly sets `opt( `zeroWidth) and `opt(zeroHeight), too since there is no useful default size for such an image.

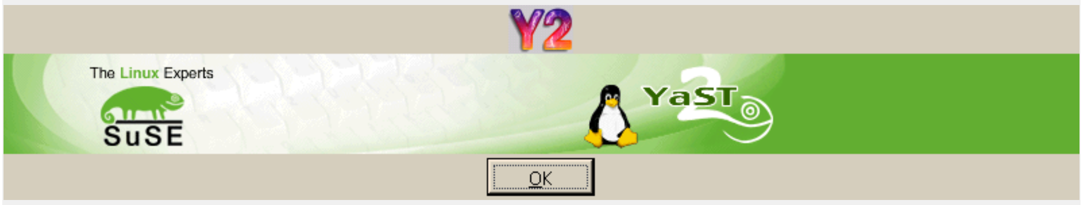
Please note that setting both `opt( `tiled) and `opt( `scaleToFit) at once doesn't make any sense.

Usage

```
         `Image( `suseheader, "SuSE Linux 7.0" )
```

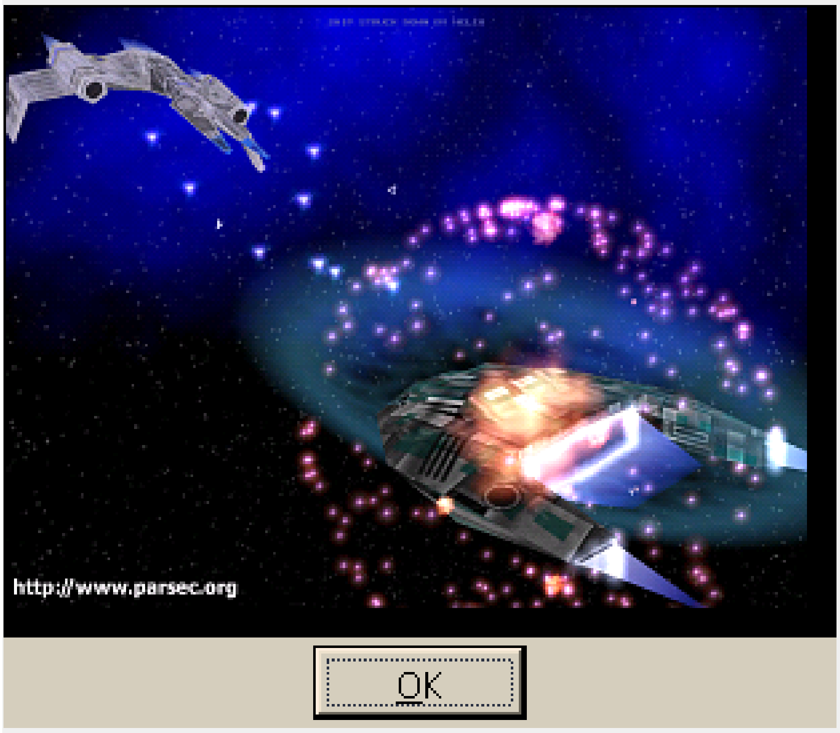
Examples

```
          /**
 * Example for simple images:
 * Creates one standard (predefined) image and a user-defined image.
 **/
{
    global define void ImageDemo(byteblock pixels) ``{
        UI::OpenDialog(
                   `VBox(
                         `Image(pixels, "Image 1"),
                         `PushButton("&OK")));
        UI::UserInput();
        UI::CloseDialog();
    };

    byteblock image_data = (byteblock) SCR::Read( .target.byte, "image1.png" );
    ImageDemo( image_data );
}
```

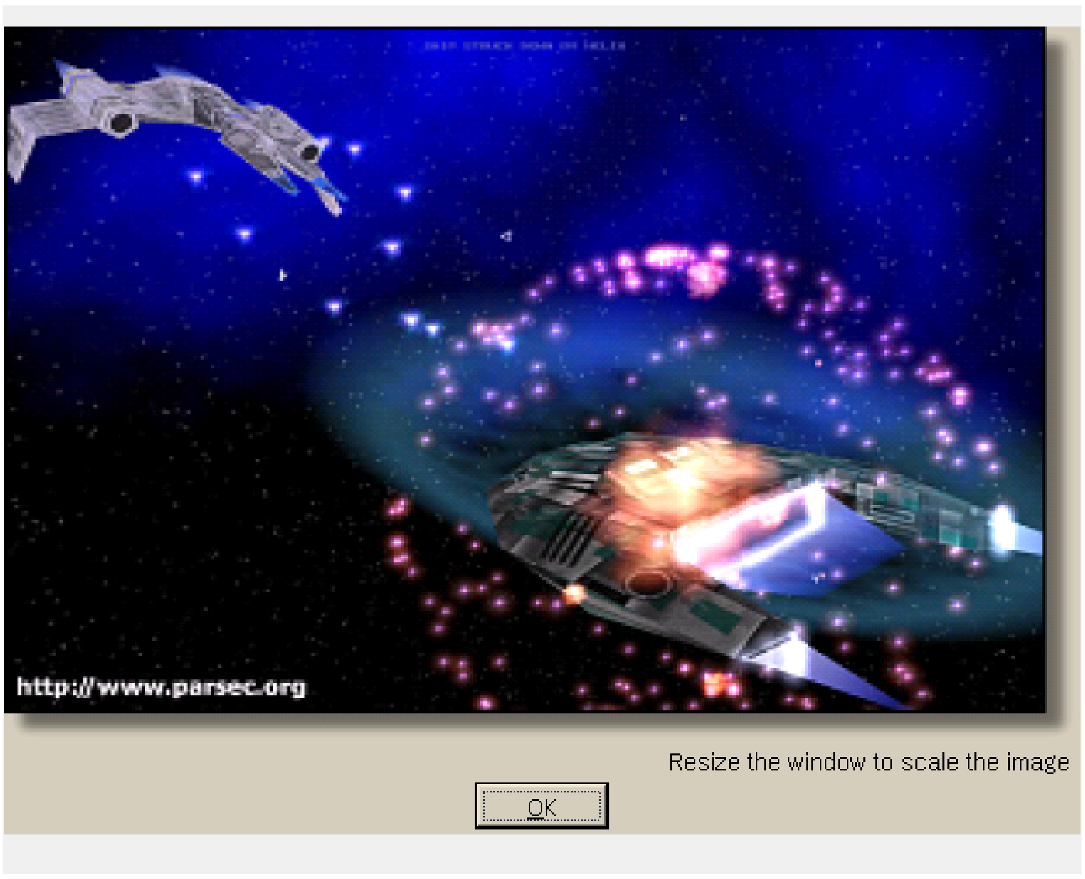


```
          /**
 * Example for an animated image
 **/
{
    global define void MovieDemo(byteblock movie) ``{
        UI::OpenDialog(
                   `VBox(
                         `Image(`opt(`animated), movie, "Movie"),
                         `PushButton("&OK")));
        UI::UserInput();
        UI::CloseDialog();
    };

    byteblock movie = (byteblock) SCR::Read( .target.byte, "/usr/share/splash/keys.mng" );
    MovieDemo( movie );
}
```

```
          /**
 * Advanced Image widget features:
 * Load image from local file. This may not always be supported, so check if
 * this feature is available prior to using it!
 **/
{

    if ( ! lookup( UI::GetDisplayInfo(), "HasLocalImageSupport", false ) )
    {
        UI::OpenDialog(
                   `VBox(
                         `Label("Loading images from local files not supported!"),
                         `PushButton(`opt(`default), "&OK")
                         )
                   );
        UI::UserInput();
        UI::CloseDialog();

        return;
    }


    UI::OpenDialog(
               `VBox(
                     `Image( "/usr/share/doc/susetour/img/games0.png", "Games" ),
                     `PushButton(`opt(`default), "&OK")
                     )
               );
    UI::UserInput();
    UI::CloseDialog();
}
```

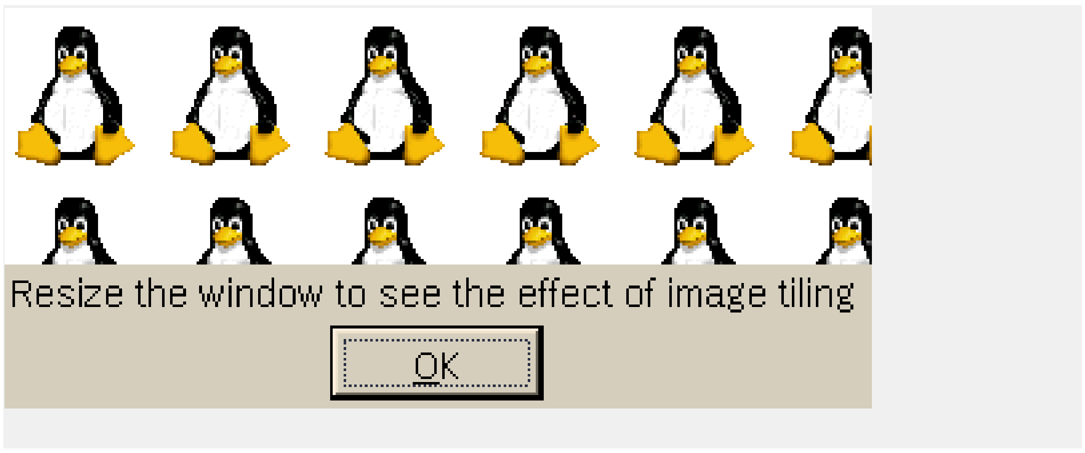


```
          /**
 * Advanced Image widget features:
 * Load image from local file and scale it to fit the available space.
 **/
{

    if ( ! lookup( UI::GetDisplayInfo(), "HasLocalImageSupport", false ) )
    {
        UI::OpenDialog(
                   `VBox(
                         `Label("Loading images from local files not supported!"),
                         `PushButton(`opt(`default), "&OK")
                         )
                   );
        UI::UserInput();
        UI::CloseDialog();

        return;
    }


    UI::OpenDialog( `opt(`defaultsize),
               `VBox(
                     `Image( `opt(`scaleToFit), "/usr/share/doc/susetour/img/games0.png", "Games" ),
                     `Right( `Label( "Resize the window to scale the image" ) ),
                     `PushButton(`opt(`default), "&OK")
                     )
               );
    UI::UserInput();
    UI::CloseDialog();
}
```



Resize the window to scale the image

```
        /**
 * Advanced Image widget features:
 * Load image from local file and scale it to fit the available space.
 **/
{

    if ( ! lookup( UI::GetDisplayInfo(), "HasLocalImageSupport", false ) )
    {
        UI::OpenDialog(
                   `VBox(
                         `Label("Loading images from local files not supported!"),
                         `PushButton(`opt(`default), "&OK")
                         )
                   );
        UI::UserInput();
        UI::CloseDialog();

        return;
    }


    UI::OpenDialog(
               `VBox(
                     `HBox(
                           `VSpacing( 5 ),
                           `Image(  `opt(`tiled ), "/opt/kde2/share/apps/amor/static/tux.png", "tux" )
                           ),
                     `Right( `Label( "Resize the window to see the effect of image tiling" ) ),
                     `PushButton(`opt(`default), "&OK")
                     )
               );
    UI::UserInput();
    UI::CloseDialog();
}
```


Resize the window to see the effect of image tiling

Name

IntField -- Numeric limited range input field

IntField

```
IntField (label, minValue, maxValue, initialValue);
string label ;
integer minValue ;
integer maxValue ;
integer initialValue ;
```

Parameters

string *label*	Explanatory label above the input field
integer *minValue*	minimum value
integer *maxValue*	maximum value
integer *initialValue*	initial value

Properties

integer *Value*	the numerical value
string *Label*	the slider label

Description

A numeric input field for integer numbers within a limited range. This can be considered a light-weight version of the <link linkend="Slider_widget">Slider</link> widget, even as a replacement for this when the specific UI doesn't support the Slider. Remember it always makes sense to specify limits for numeric input, even if those limits are very large (e.g. +/- MAXINT).

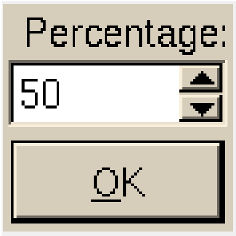
Fractional numbers are currently not supported.

Usage

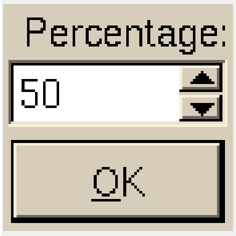
```
        `IntField( "Percentage", 1, 100, 50 )
```

Examples

```
        {
    UI::OpenDialog(
               `VBox(
                     `IntField( "Percentage:", 0, 100, 50),
                     `PushButton(`opt(`default), "&OK")
                     )
               );
    UI::UserInput();
    UI::CloseDialog();
}
```



```
        {
    UI::OpenDialog(
               `VBox(
                     `IntField( `id(`perc), "Percentage:", 0, 100, 50),
                     `PushButton(`opt(`default), "&OK")
                     )
               );
    UI::UserInput();

    integer percentage = (integer) UI::QueryWidget(`id(`perc), `Value);
    UI::CloseDialog();

    UI::OpenDialog(
               `VBox(
                     `Label( sformat( "You entered: %1%%", percentage) ),
                     `PushButton(`opt(`default), "&OK")
                     )
               );
    UI::UserInput();
}
```



Name

PackageSelector -- Complete software package selection

PackageSelector

`PackageSelector` `();`

Options

`youMode`
: start in YOU (YaST Online Update) mode

`updateMode`
: start in update Mode

Optional Arguments

string *`floppy-Device`*

Description

A very complex widget that handles software package selection completely transparently. Set up the package manager (the backend) before creating this widget and let the package manager and the package selector handle all the rest. The result of all this are the data stored in the package manager.

Use UI::RunPkgSelection() after creating a dialog with this widget. The result of UI::UserInput() in a dialog with such a widget is undefined - it may or may not return.

This widget gets the (best) floppy device as a parameter since the UI has no general way of finding out by itself what device can be used for saving or loading pacakge lists etc. - this is best done outside and passed here as a parameter.

Usage

```
	`PackageSelector( "/dev/fd0" )
```

Examples

```
	{
    Pkg::TargetInit( "/", // installed system
                     false );        // don't create a new RPM database
    UI::OpenDialog(`opt(`defaultsize), `PackageSelector(`id(`selector), `opt(`testMode, `searchMode), "/dev/f
    any input = UI::RunPkgSelection(`id(`selector) );
    UI::CloseDialog();

    y2milestone( "Input: %1", input );
}
```

Name

PkgSpecial -- Package selection special - DON'T USE IT

PkgSpecial

PkgSpecial ();

Description

Use only if you know what you are doing - that is, DON'T USE IT.

Usage

```
        `PkgSpecial( "subwidget_name" )
```

Special (optional) widgets

Name

HasSpecialWidget -- Checks for support of a special widget type.

HasSpecialWidget

`HasSpecialWidget ();`

Description

Checks for support of a special widget type. Use this prior to creating a widget of this kind. Do not use this to check for ordinary widgets like PushButton etc. - just the widgets where the widget documentation explicitly states it is an optional widget not supported by all UIs.

Returns true if the UI supports the special widget and false if not.

Name

BarGraph -- Horizontal bar graph (optional widget)

BarGraph

```
BarGraph (values, labels);
list values ;
list labels ;
```

Parameters

list *values*	the initial values (integer numbers)

Optional Arguments

list *labels*	the labels for each part; use "%1" to include the current numeric value. May include newlines.

Properties

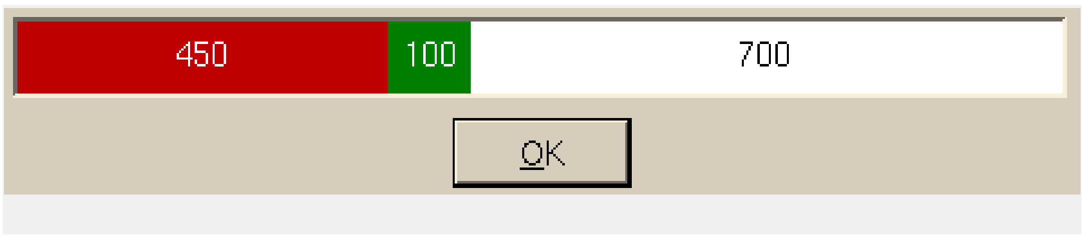
integer-list *Values*	The numerical values of each segment.
string-list *Labels*	The labels for each segment. "\n" allowed. Use "%1" as a placeholder for the current value.

Usage

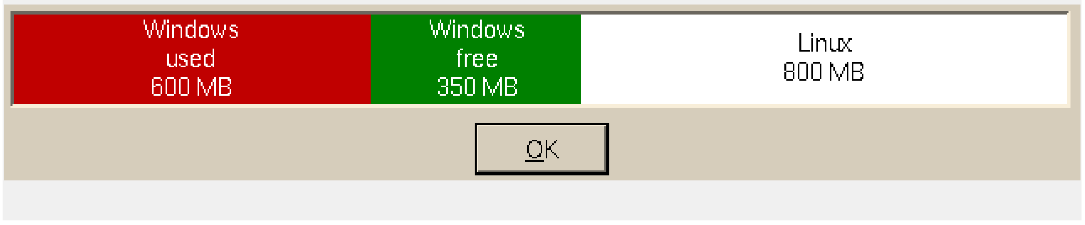
```
        if ( HasSpecialWidget( `BarGraph ) {...
        `BarGraph( [ 450, 100, 700 ],
        [ "Windows used\n%1 MB", "Windows free\n%1 MB", "Linux\n%1 MB" ] )
```

Examples

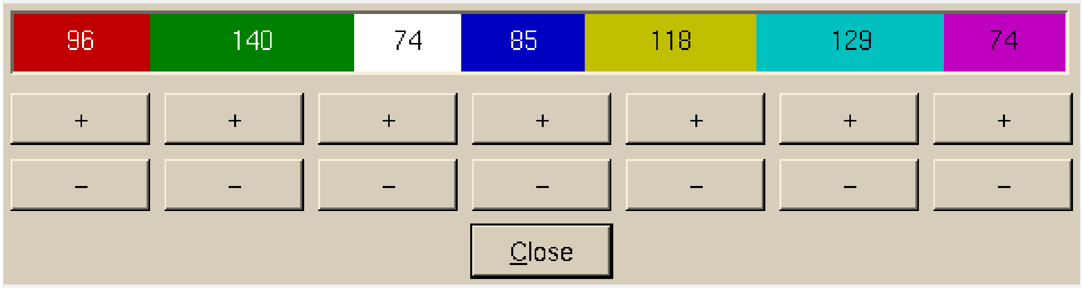
```
      {
    if ( ! UI::HasSpecialWidget(`BarGraph) )
    {
        UI::OpenDialog(
                   `VBox(
                         `Label("Error: This UI doesn't support the BarGraph widget!"),
                         `PushButton(`opt(`default), "&OK")
                         )
                   );
        UI::UserInput();
        UI::CloseDialog();

        return;
    }

    UI::OpenDialog(
               `VBox(
                     `HSpacing( 60 ), // wider default size
                     `BarGraph( [450, 100, 700] ),
                     `PushButton(`opt(`default), "&OK")
                     )
               );
    UI::UserInput();
    UI::CloseDialog();
}
```



```
        {
    if ( ! UI::HasSpecialWidget(`BarGraph) )
    {
        UI::OpenDialog(
                   `VBox(
                         `Label("Error: This UI doesn't support the BarGraph widget!"),
                         `PushButton(`opt(`default), "&OK")
                         )
                   );
        UI::UserInput();
        UI::CloseDialog();

        return;
    }

    UI::OpenDialog(
               `VBox(
                     `HSpacing(80),          // force width
                     `HBox(`opt(`debugLayout),
                           `BarGraph(
                                     `opt(`vstretch),
                                     [600, 350, 800],
                                     [
                                      "Windows\nused\n%1 MB",
                                      "Windows\nfree\n%1 MB",
                                      "Linux\n%1 MB"
                                     ]
                                     )
                           ),
                     `PushButton(`opt(`default), "&OK")
                     )
               );
    UI::UserInput();
    UI::CloseDialog();
}
```



```
          // Advanced BarGraph example:
//
// Create a dialog with a BarGraph with a number of segments
// and a "+" and a "-" button for each segment.

{
    // Check for availability of the BarGraph widget - this is necessary since
    // this is an optional widget that not all UIs need to support.

    if ( ! UI::HasSpecialWidget(`BarGraph) )
    {
        // Pop up error message if the BarGraph widget is not available

        UI::OpenDialog(
                   `VBox(
                         `Label("Error: This UI doesn't support the BarGraph widget!"),
                         `PushButton(`opt(`default), "&OK")
                         )
                   );
        UI::UserInput();
        UI::CloseDialog();

        return;
    }
    // list values = [ 100, 200, 300, 150, 250, 120, 200, 120 ];
    list<integer> values = [ 100, 100, 100, 100, 100, 100, 100 ];
    integer        inc = 10;  // increment / decrement for each button press

    // Create the main dialog:
    //
    // One BarGraph at the top, below that two rows of equal sized (thus the
    // weights) buttons, below that a "close" button.
    //
    // The "+" / "-" -buttons use an integer value as their ID which can be
    // used to point to the index of the value to be changed. If the ID is
    // negative it means subtract rather than add.

    term plus_buttons =  `HBox();
    term minus_buttons = `HBox();
    integer i = 1;

    foreach( `val, values, ``{
        plus_buttons  = add( plus_buttons,  `HWeight( 1, `PushButton(`id( i), "+" ) ) );
        minus_buttons = add( minus_buttons, `HWeight( 1, `PushButton(`id(-i), "-" ) ) );
        i = i+1;
    });

    UI::OpenDialog(
               `VBox(
                     `BarGraph(`id(`bar), values ),
                     plus_buttons,
                     minus_buttons,
                     `PushButton(`id(`close), `opt(`default), "&Close")
                     )
               );


    // Event processing loop - left only via the "close" button
    // or the window manager close button / function.

    any button_id = nil;

    do
    {
        button_id = UI::UserInput(); // wait for button click

        if ( button_id != `close && button_id != `cancel )
        {
            integer sign = 1;

            if ( button_id < 0 )
            {
                sign   = -1;
                button_id = -(integer)button_id;
            }

            // Loop over the values. Increment the value corresponding to the
            // clicked button, decrement all others as to maintain the total
            // sum of all values - or vice versa for negative button IDs
            // (i.e. "-" buttons).

            list<integer> new_values = [];
            integer i = 0;

            while ( i < size( values ) )
            {
                integer old_val = values[i]:0;

                if ( i+1 == button_id )
                    new_values = add( new_values, old_val + (sign*inc) );
                else
                    new_values = add( new_values, old_val + (-sign *(inc/( size(values)-1))) );

                i = i+1;
            }

            values = new_values;
            UI::ChangeWidget(`id(`bar), `Values, values );
        }

    } while ( button_id != `close && button_id != `cancel );

    UI::CloseDialog();
}
```



Name

ColoredLabel -- Simple static text with specified background and foreground color

ColoredLabel

```
ColoredLabel (label, foreground, background, margin);
string label ;
color foreground ;
color background ;
integer margin ;
```

Parameters

string *label*	
color *foreground*	color
color *background*	color
integer *margin*	around the widget in pixels

Properties

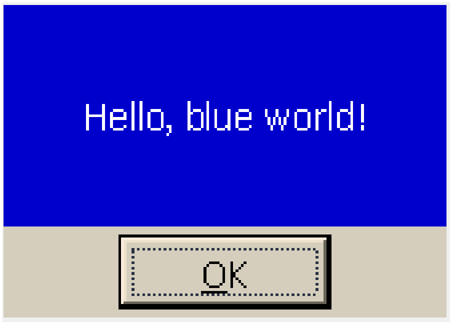
string *Value*	the label text

Usage

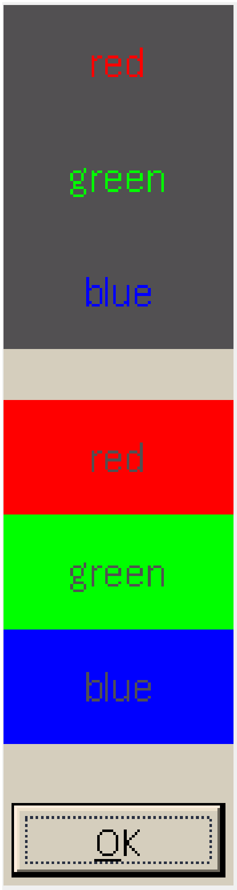
```
        `ColoredLabel( "Hello, World!", `rgb( 255, 0, 255 ), `rgb( 0, 128, 0 ), 20 )
```

Examples

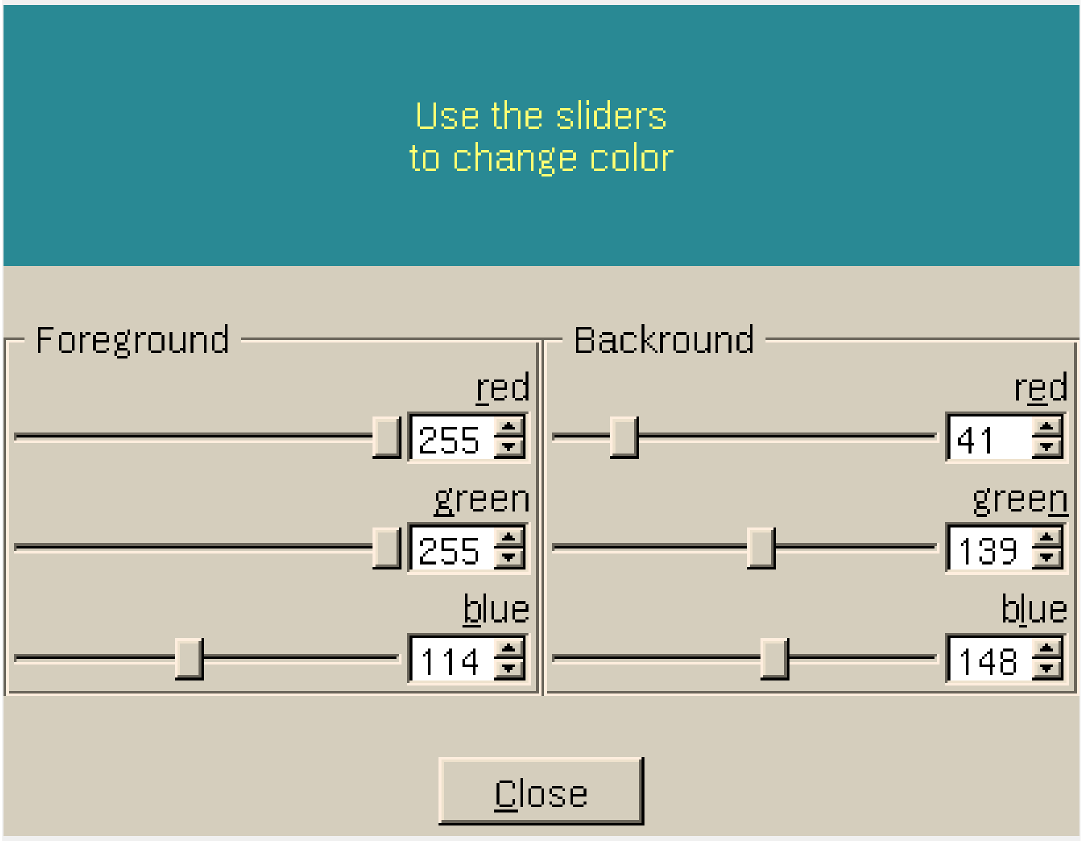
```
      {
    if ( ! UI::HasSpecialWidget(`ColoredLabel) )
    {
        UI::OpenDialog(
                   `VBox(
                         `Label("Error: This UI doesn't support the ColoredLabel widget!"),
                         `PushButton(`opt(`default), "&OK")
                         )
                   );
        UI::UserInput();
        UI::CloseDialog();

        return;
    }

    UI::OpenDialog(
               `VBox(
                     `ColoredLabel( "Hello, blue world!",
                                    `rgb(255, 255, 255 ), // foreground
                                    `rgb(  0,   0, 200 ), // background
                                    30 ),
                     `PushButton(`opt(`default), "&OK")
                     )
               );
    UI::UserInput();
    UI::CloseDialog();
}
```



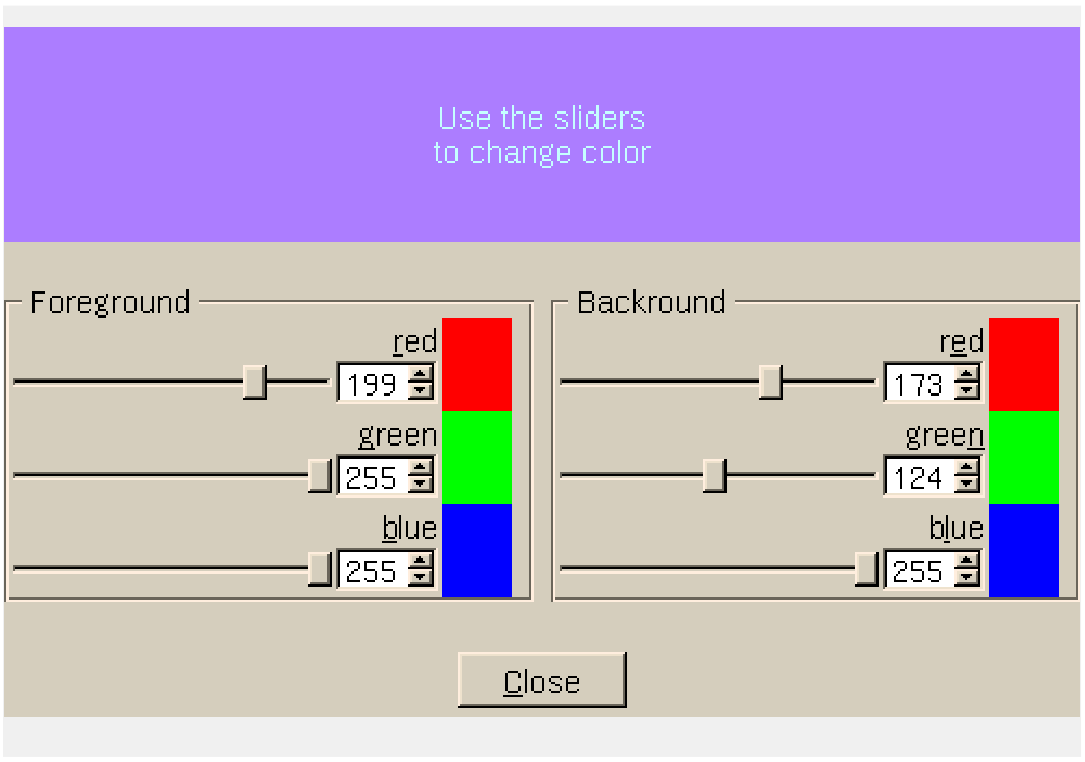
```
          {
    if ( ! UI::HasSpecialWidget(`ColoredLabel) )
    {
        UI::OpenDialog(
                   `VBox(
                         `Label("Error: This UI doesn't support the ColoredLabel widget!"),
                         `PushButton(`opt(`default), "&OK")
                         )
                   );
        UI::UserInput();
        UI::CloseDialog();

        return;
    }

    UI::OpenDialog(
               `VBox(
                     `ColoredLabel( `opt(`hstretch), "red"  , `rgb( 255,  0,    0 ), `rgb( 80, 80, 80 ), 10 ),
                     `ColoredLabel( `opt(`hstretch), "green", `rgb(   0, 255,   0 ), `rgb( 80, 80, 80 ), 10 ),
                     `ColoredLabel( `opt(`hstretch), "blue" , `rgb(   0,   0, 255 ), `rgb( 80, 80, 80 ), 10 ),
                     `VSpacing(),
                     `ColoredLabel( `opt(`hstretch), "red"  , `rgb( 80, 80, 80 ), `rgb( 255,   0,   0 ), 10 ),
                     `ColoredLabel( `opt(`hstretch), "green", `rgb( 80, 80, 80 ), `rgb(   0, 255,   0 ), 10 ),
                     `ColoredLabel( `opt(`hstretch), "blue" , `rgb( 80, 80, 80 ), `rgb(   0,   0, 255 ), 10 ),
                     `VSpacing(),
                     `PushButton(`opt(`default), "&OK")
                     )
               );
    UI::UserInput();
    UI::CloseDialog();
}
```



```
        /**
 * Advanced ColoredLabel example: A sample ColoredLabel widget and sliders
 * for both foreground and background colors.
 **/
{
    if ( ! UI::HasSpecialWidget(`ColoredLabel) || ! UI::HasSpecialWidget(`Slider) )
    {
        UI::OpenDialog(
                   `VBox(
                         `Label("Error: This UI doesn't support the required special widgets!"),
                         `PushButton(`opt(`default), "&OK")
                         )
                   );
        UI::UserInput();
        UI::CloseDialog();

        return;
    }

    define term sample( integer fg_red, integer fg_green, integer fg_blue,
                        integer bg_red, integer bg_green, integer bg_blue ) ``{
        return `ColoredLabel( `opt(`hstretch),
                              "Use the sliders\nto change color",
                              `rgb( fg_red, fg_green, fg_blue ),
                              `rgb( bg_red, bg_green, bg_blue ),
                              30 );
    };

    UI::OpenDialog(
               `VBox(
                     `ReplacePoint(`id(`sample), sample( 200, 0, 0,
                                                         0, 0, 200) ),
                     `VSpacing(),
                     `HBox(
                           `Frame( "Foreground",
                                   `VBox(
                                         `Slider(`id(`fg_red  ),`opt(`notify), "&red",   0, 255, 200 ),
                                         `Slider(`id(`fg_green),`opt(`notify), "&green", 0, 255,   0 ),
                                         `Slider(`id(`fg_blue ),`opt(`notify), "&blue",  0, 255,   0 )
                                         )
                                   ),
                           `Frame( "Backround",
                                   `VBox(
                                         `Slider(`id(`bg_red  ), `opt(`notify), "r&ed",   0, 255,   0 ),
                                         `Slider(`id(`bg_green), `opt(`notify), "gree&n", 0, 255,   0 ),
                                         `Slider(`id(`bg_blue ), `opt(`notify), "b&lue",  0, 255, 200 )
                                         )
                                   )
                            ),
                     `VSpacing(),
                     `PushButton(`id(`close), "&Close")
                     )
                );

    any widget = nil;

    do
    {
        widget = UI::UserInput();

        if ( widget != `close )
        {
            UI::ReplaceWidget(`id(`sample),
                          sample( (integer) UI::QueryWidget(`id(`fg_red  ), `Value ),
                                  (integer) UI::QueryWidget(`id(`fg_green), `Value ),
                                  (integer) UI::QueryWidget(`id(`fg_blue ), `Value ),

                                  (integer) UI::QueryWidget(`id(`bg_red  ), `Value ),
                                  (integer) UI::QueryWidget(`id(`bg_green), `Value ),
                                  (integer) UI::QueryWidget(`id(`bg_blue ), `Value )
                                  )
                          );

        }

    } while ( widget != `close );

    UI::CloseDialog();
}
```



```
          /**
 * Advanced ColoredLabel example: A sample ColoredLabel widget and sliders
 * for both foreground and background colors and a colored label beside each slider.
 **/
{
    if ( ! UI::HasSpecialWidget(`ColoredLabel) || ! UI::HasSpecialWidget(`Slider) )
    {
        UI::OpenDialog(
                   `VBox(
                         `Label("Error: This UI doesn't support the required special widgets!"),
                         `PushButton(`opt(`default), "&OK")
                         )
                   );
        UI::UserInput();
        UI::CloseDialog();

        return;
    }

    define term sample( integer fg_red, integer fg_green, integer fg_blue,
                        integer bg_red, integer bg_green, integer bg_blue ) ``{
        return `ColoredLabel( `opt(`hstretch),
                              "Use the sliders\nto change color",
                              `rgb( fg_red, fg_green, fg_blue ),
                              `rgb( bg_red, bg_green, bg_blue ),
                              30 );
    };

    UI::OpenDialog(
               `VBox(
                     `ReplacePoint(`id(`sample), sample( 200, 0, 0,
                                                         0, 0, 200) ),
                     `VSpacing(),
                     `HBox(
                           `Frame( "Foreground",
                                   `VBox(
                                         `HBox(
                                               `Slider(`id(`fg_red  ),`opt(`notify), "&red",   0, 255, 200 ),
                                               `ColoredLabel("   ", `rgb(255, 0, 0), `rgb(255, 0, 0), 10 ),
                                               `HSpacing()
                                               ),
                                         `HBox(
                                               `Slider(`id(`fg_green),`opt(`notify), "&green", 0, 255,   0 ),
                                               `ColoredLabel("   ", `rgb(0, 255, 0), `rgb(0, 255, 0), 10 ),
                                               `HSpacing()
                                               ),
                                         `HBox(
                                               `Slider(`id(`fg_blue ),`opt(`notify), "&blue",  0, 255,   0 ),
                                               `ColoredLabel("   ", `rgb(0, 0, 255), `rgb(0, 0, 255), 10 ),
                                               `HSpacing()
                                               )
                                         )
                                   ),
                           `HSpacing(),
                           `Frame( "Backround",
                                   `VBox(
                                         `HBox(
                                               `Slider(`id(`bg_red  ),`opt(`notify), "&red",   0, 255, 0 ),
                                               `ColoredLabel("   ", `rgb(255, 0, 0), `rgb(255, 0, 0), 10 ),
                                               `HSpacing()
                                               ),
                                         `HBox(
                                               `Slider(`id(`bg_green),`opt(`notify), "&green", 0, 255,   0 ),
                                               `ColoredLabel("   ", `rgb(0, 255, 0), `rgb(0, 255, 0), 10 ),
                                               `HSpacing()
                                               ),
                                         `HBox(
                                               `Slider(`id(`bg_blue ),`opt(`notify), "&blue",  0, 255, 200 ),
                                               `ColoredLabel("   ", `rgb(0, 0, 255), `rgb(0, 0, 255), 10 ),
                                               `HSpacing()
                                               )
                                         )
                                   )
                           ),
                     `VSpacing(),
                     `PushButton(`id(`close), "&Close")
                     )
               );

    any widget = nil;

    do
    {
        widget = UI::UserInput();

        if ( widget != `close )
        {
            UI::ReplaceWidget(`id(`sample),
                              sample( (integer) UI::QueryWidget(`id(`fg_red  ), `Value ),
                                      (integer) UI::QueryWidget(`id(`fg_green), `Value ),
                                      (integer) UI::QueryWidget(`id(`fg_blue ), `Value ),

                                      (integer) UI::QueryWidget(`id(`bg_red  ), `Value ),
                                      (integer) UI::QueryWidget(`id(`bg_green), `Value ),
                                      (integer) UI::QueryWidget(`id(`bg_blue ), `Value )
                                      )
                              );

        }

    } while ( widget != `close );

    UI::CloseDialog();
}
```


Use the sliders
to change color
Foreground
red
199
green
255
blue
255
Backround
red
173
green
124
blue
255
Close

Name

DownloadProgress -- Self-polling file growth progress indicator (optional widget)

DownloadProgress

```
DownloadProgress (label, filename, expectedSize);
string label ;
string filename ;
integer expectedSize ;
```

Parameters

string *label*	label above the indicator
string *filename*	file name with full path of the file to poll
integer *expectedSize*	expected final size of the file in bytes

Properties

string *Label*	the label above the progress indicator
string *Filename*	file name with full path of the file to poll
integer *ExpectedSize*	expected final size of the file in bytes

Description

This widget automatically displays the progress of a lengthy download operation. The widget itself (i.e. the UI) polls the specified file and automatically updates the display as required even if the download is taking place in the foreground.

Please notice that this will work only if the UI runs on the same machine as the file to download which may not taken for granted (but which is so for most users).

Note

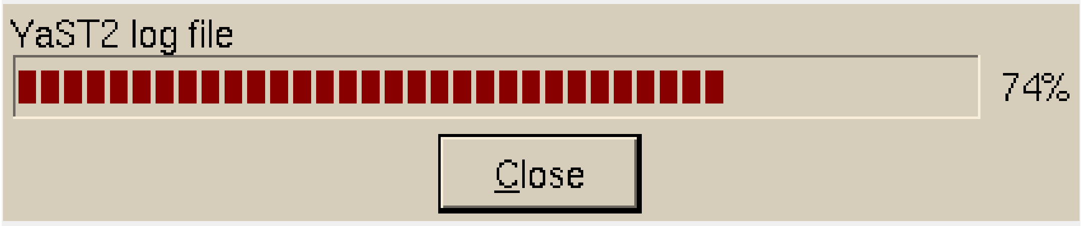
This is a "special" widget, i.e. not all UIs necessarily support it. Check for availability with *HasSpecialWidget(`DownloadProgress)* before using it.

Usage

```
      if ( HasSpecialWidget( `DownloadProgress ) {...
      `DownloadProgress( "Base system ( 230k )", "/tmp/aaa_base.rpm", 230*1024 );
```

Examples

```
      {
  if ( ! UI::HasSpecialWidget(`DownloadProgress) )
  {
        UI::OpenDialog(
                   `VBox(
                         `Label("Error: This UI doesn't support the DownloadProgress widget!"),
                         `PushButton(`opt(`default), "&OK")
                         )
                   );
        UI::UserInput();
        UI::CloseDialog();

        return;
    }

    string  filename = "/suse/sh/.y2log";
    // string  filename = "/var/log/y2log";
    integer expected_size = 20 * 1024;

    UI::OpenDialog(
               `VBox(
                     `DownloadProgress("YaST2 log file", filename, expected_size ),
                     `HSpacing(50), // force width
                     `PushButton(`opt(`default), "&Close")
                     )
               );
    UI::UserInput();
    UI::CloseDialog();
}
```



Name

DumbTab -- Simplistic tab widget that behaves like push buttons

DumbTab

```
DumbTab (tabs, contents);
list tabs ;
term contents ;
```

Parameters

list *tabs*	page headers
term *contents*	page contents - usually a ReplacePoint

Properties

any *CurrentItem*	the currently selected tab

Description

This is a very simplistic approach to tabbed dialogs: The application specifies a number of tab headers and the page contents and takes care of most other things all by itself, in particular page switching. Each tab header behaves very much like a PushButton - as the user activates a tab header, the DumbTab widget simply returns the ID of that tab (or its text if it has no ID). The application should then take care of changing the page contents accordingly - call UI::ReplaceWidget() on the ReplacePoint specified as tab contents or similar actions (it might even just replace data in a Table or RichText widget if this is the tab contents). Hence the name *Dumb*Tab.

The items in the item list can either be simple strings or `item() terms with an optional ID for each individual item (which will be returned upon UI::UserInput() and related when the user selects this tab), a (mandatory) user-visible label and an (optional) flag that indicates that this tab is initially selected. If you specify only a string, UI::UserInput() will return this string.

This is a "special" widget, i.e. not all UIs necessarily support it. Check for availability with *HasSpecialWidget(`DumbTab)* before using it.

Note

Please notice that using this kind of widget more often than not is the result of *poor dialog or workflow design.*

Using tabs only hides complexity, but the complexity remains there. They do little to make problems simpler. This however should be the approach of choice for good user interfaces.

It is very common for tabs to be overlooked by users if there are just two tabs to select from, so in this case better use an "Expert..." or "Details..." button - this gives much more clue to the user that there is more information available while at the same time clearly indicating that those other options are much less commonly used.

If there are very many different views on data or if there are lots and lots of settings, you might consider using a tree for much better navigation. The Qt UI's wizard even has a built-in tree that can be used instead of the help panel.

If you use a tree for navigation, unter all circumstances avoid using tabs at the same time - there is no way for the user to tell which tree nodes have tabs and which have not, making navigation even more difficult. KDE's control center or Mozilla's settings are very good examples how *not* to do that - you become bogged down for sure in all those tree nodes and tabs hidden within so many of them.

Usage

```
        if ( HasSpecialWidget( `DumbTab) {...
        `DumbTab( [ `item(`id(`page1), "Page &1" ), `item(`id(`page2), "Page &2" ) ], contents; }
```

Examples

```
         // Minimalistic example for tab widget
{
    if ( ! UI::HasSpecialWidget(`DumbTab ) )
    {
        UI::OpenDialog(
                   `VBox(
                         `Label("Error: This UI doesn't support the DumbTab widget!"),
                         `PushButton(`opt(`default), "&OK")
                         )
                   );
        UI::UserInput();
        UI::CloseDialog();

        return;
    }


    UI::OpenDialog(
                   `VBox(
                         `DumbTab(
                                  [ "Page 1", "Page 2", "Page 3" ],
                                  `RichText(`id(`contents), "Contents" )
                                  ),
                         `Right(`PushButton(`id(`close), "&Close" ) )
                         )
                   );

    UI::DumpWidgetTree();

    any input = nil;

    repeat
    {
        input = UI::UserInput();

        if ( is( input, string ) )
        {
            UI::ChangeWidget(`contents, `Value, (string) input );
        }
    } until ( input == `close );


    UI::CloseDialog();
}
```

```
         // Typical usage example for tab widget
{
    term address_page =
        `VBox(
              `Left( `Heading( "Address" ) ),
              `VSpacing(),
              `HCenter(
                       `HSquash(
                                `VBox(
                                      `HSpacing( 50 ),
                                      `TextEntry( "Name" ),
                                      `TextEntry( "E-Mail" ),
                                      `TextEntry( "Phone" ),
                                      `VSpacing(),
                                      `MultiLineEdit( "Comments" ),
                                      `VStretch()
                                      )
                                )
                       )
              );

    term overview_page =
        `VBox(
              `Left( `Heading( "DumbTab Widget Overview" ) ),
              `VSpacing(),
              `Label( "This kind of tab is pretty dumb - hence the name DumbTab.\n"
                      + "You need to do most everything yourself.\n"
                      + "Each tab behaves very much like a push button;\n"
                      + "the YCP application is notified when the user clicks on a tab.\n"
                      + "The application must take care to exchange the tab contents." )
              );

    term style_hints_page =
        `VBox(
              `Left( `Heading( "GUI Style Hints" ) ),
              `VSpacing(),
              `Heading( "Using tabs is usually a result of poor dialog design." ),
              `VSpacing(),
              `Left(
                    `Label( "Tabs hide complexity, they do not resolve it.\n"
                            + "The problem remains just as complex as before,\n"
                            + "only the user can no longer see it."
                            )
                    )
              );



    UI::OpenDialog(`opt(`defaultsize),
                   `VBox(
                         `DumbTab( [
                                    `item(`id(`address  ), "&Address"  ),
                                    `item(`id(`overview ), "&Overview" ),
                                    `item(`id(`style    ), "GUI &Style Hints", true ) // true: selected
                                    ],
                                   `Left(
                                         `Top(
                                              `HVSquash(
                                                        `VBox(
                                                              `VSpacing(0.3),
                                                              `HBox(
                                                                    `HSpacing(1),
                                                                    `ReplacePoint(`id(`tabContents ), style_h
                                                                    )
                                                              )
                                                        )
                                              )
                                         )
                                   ),
                         `Right(`PushButton(`id(`close), "Cl&ose" ) )
                         )
                   );


    while ( true )
    {
        symbol widget = (symbol) UI::UserInput();

        if         ( widget == `close ) break;
        else if ( widget == `address ) UI::ReplaceWidget(`tabContents, address_page );
        else if ( widget == `overview ) UI::ReplaceWidget(`tabContents, overview_page );
        else if ( widget == `style ) UI::ReplaceWidget(`tabContents, style_hints_page );
    }

    UI::CloseDialog();
}
```

Name

VMultiProgressMeter -- Progress bar with multiple segments (optional widget)

VMultiProgressMeter, HMultiProgressMeter

```
VMultiProgressMeter (maxValues);
List<integer> maxValues ;
HMultiProgressMeter (maxValues);
List<integer> maxValues ;
```

Parameters

List<integer> *maxValues*	maximum values

Properties

string *Values*	the current values for all segments

Description

A vertical (VMultiProgressMeter) or horizontal (HMultiProgressMeter) progress display with multiple segments. The numbers passed on widget creation are the maximum numbers of each individual segment. Segments sizes will be displayed proportionally to these numbers.

This widget is intended for applications like showing the progress of installing from multiple CDs while giving the user a hint how much will be installed from each individual CD.

```
 Set actual values later with
UI::ChangeWidget(`id(...), `Values, [ 1, 2, ...] );
```

The widget may choose to reserve a minimum amount of space for each segment even if that means that some segments will be shown slightly out of proportion.

Note

This is a "special" widget, i.e. not all UIs necessarily support it. Check for availability with *HasSpecialWidget(`MultiProgressMeter)* before using it.

Usage

```
        if ( HasSpecialWidget( `MultiProgressMeter ) {...
        `MultiProgressMeter( "Percentage", 1, 100, 50 )
```

Examples

```
          // Simple example for MultiProgressMeter
{
    if ( ! UI::HasSpecialWidget(`HMultiProgressMeter ) )
    {
        UI::OpenDialog(
                       `VBox(
                             `Label("Error: This UI doesn't support the MultiProgressMeter widget!"),
                             `PushButton(`opt(`default), "&OK")
                             )
                       );
        UI::UserInput();
        UI::CloseDialog();

        return;
    }


    UI::OpenDialog(
                   `VBox(
                         `HMultiProgressMeter(`id(`prog), [ 1000, 200, 500, 20, 100 ] ),
                         `PushButton(`opt(`default), "&Ok" )
                         )
                   );

    UI::ChangeWidget(`prog, `Values, [ 1000, 200, 500, 20, 100 ] ); UI::UserInput();
    UI::ChangeWidget(`prog, `Values, [  800, 200, 500, 20, 100 ] ); UI::UserInput();
    UI::ChangeWidget(`prog, `Values, [  500, 200, 500, 20, 100 ] ); UI::UserInput();
    UI::ChangeWidget(`prog, `Values, [  200, 200, 500, 20, 100 ] ); UI::UserInput();
    UI::ChangeWidget(`prog, `Values, [    0, 200, 500, 20, 100 ] ); UI::UserInput();

    UI::ChangeWidget(`prog, `Values, [    0, 100, 500, 20, 100 ] ); UI::UserInput();
    UI::ChangeWidget(`prog, `Values, [    0,  20, 500, 20, 100 ] ); UI::UserInput();
    UI::ChangeWidget(`prog, `Values, [    0,   0, 500, 20, 100 ] ); UI::UserInput();

    UI::ChangeWidget(`prog, `Values, [    0,   0, 400, 20, 100 ] ); UI::UserInput();
    UI::ChangeWidget(`prog, `Values, [    0,   0, 300, 20, 100 ] ); UI::UserInput();
    UI::ChangeWidget(`prog, `Values, [    0,   0, 200, 20, 100 ] ); UI::UserInput();
    UI::ChangeWidget(`prog, `Values, [    0,   0, 100, 20, 100 ] ); UI::UserInput();
}
```

Name

Slider -- Numeric limited range input (optional widget)

Slider

```
Slider (label, minValue, maxValue, initialValue);
string label ;
integer minValue ;
integer maxValue ;
integer initialValue ;
```

Parameters

string *label*	Explanatory label above the slider
integer *minValue*	minimum value
integer *maxValue*	maximum value
integer *initialValue*	initial value

Properties

integer *Value*	the numerical value
string *Label*	the slider label

Description

A horizontal slider with (numeric) input field that allows input of an integer value in a given range. The user can either drag the slider or simply enter a value in the input field.

Remember you can use *`opt( `notify)* in order to get instant response when the user changes the value - if this is desired.

Note

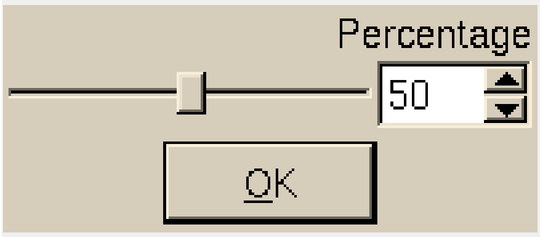
This is a "special" widget, i.e. not all UIs necessarily support it. Check for availability with *HasSpecialWidget(`Slider)* before using it.

Usage

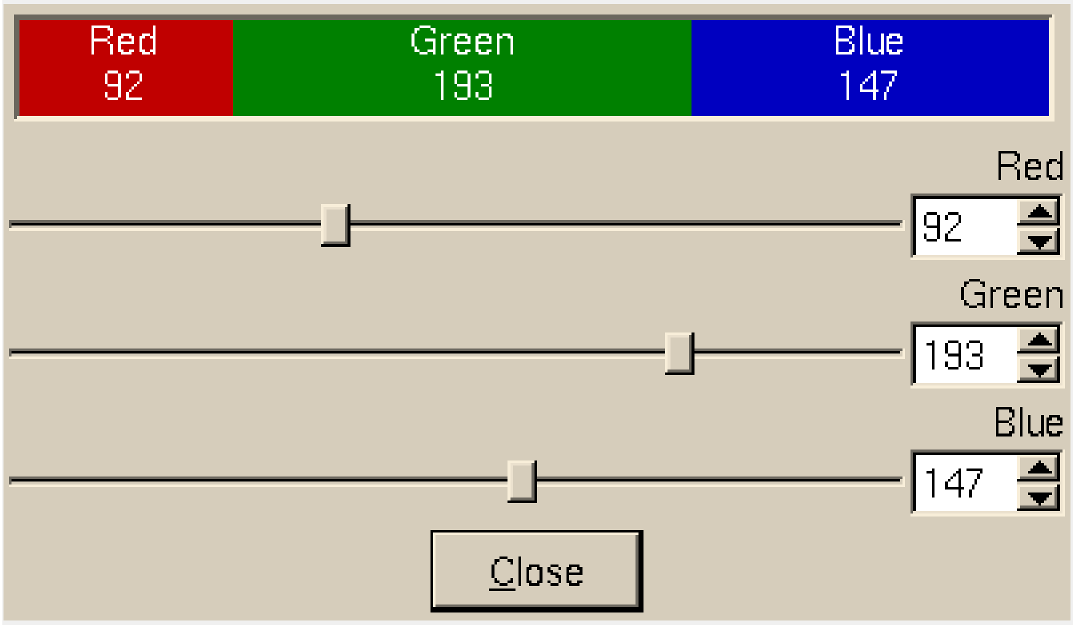
```
        if ( HasSpecialWidget( `Slider ) {...
        `Slider( "Percentage", 1, 100, 50 )
```

Examples

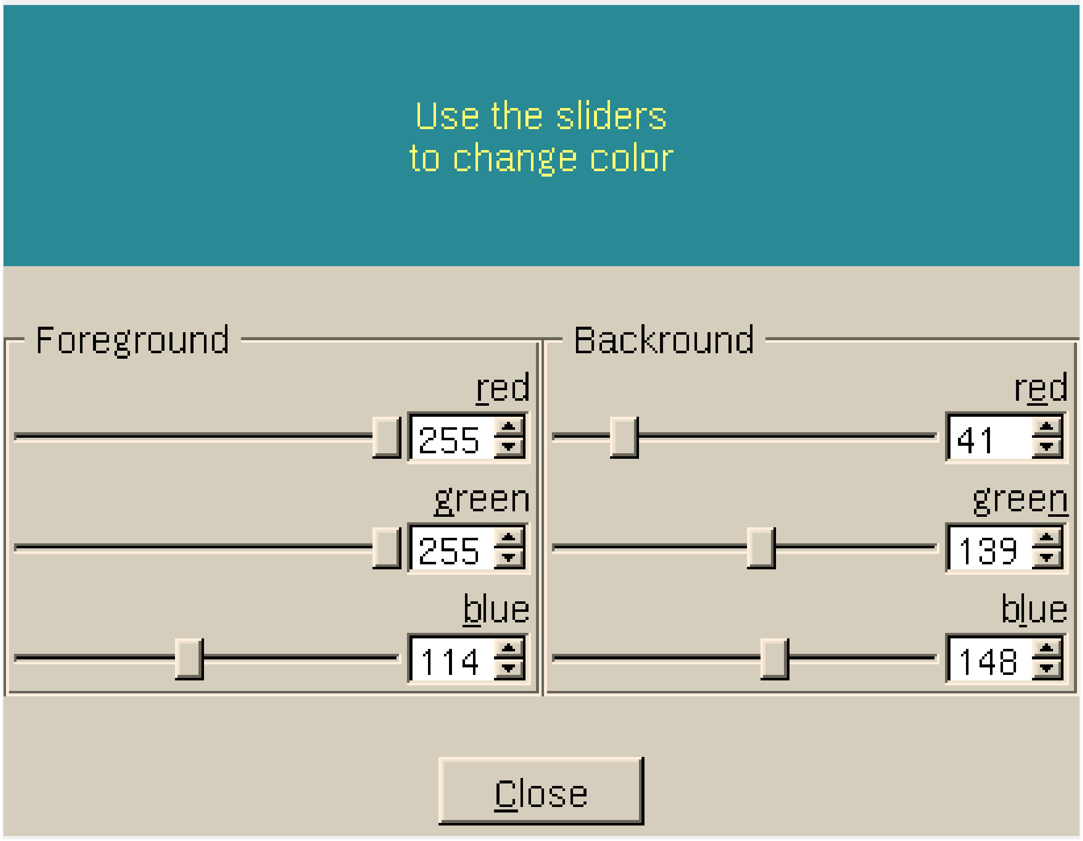
```
         {
    if ( ! UI::HasSpecialWidget(`Slider) )
    {
        UI::OpenDialog(
                   `VBox(
                         `Label("Error: This UI doesn't support the Slider widget!"),
                         `PushButton(`opt(`default), "&OK")
                         )
                   );
        UI::UserInput();
        UI::CloseDialog();

        return;
    }

    UI::OpenDialog(
               `VBox(
                     `Slider( "Percentage", 0, 100, 50),
                     `PushButton(`opt(`default), "&OK")
                     )
               );
    UI::UserInput();
    UI::CloseDialog();
}
```



```
          // Advanced Slider + BarGraph example:
//
// Display a dialog with a bar graph for RGB color percentages
// and 3 sliders for the RGB percentage.
// Update the bar graph while the user adjusts the RGB values.
//
// Unfortunately the colors don't match any more in the BarGraph widget - they
// used to be red, blue and green. You need to use a bit of imagination
// here. ;-)

{
    // Check for availability of required widgets

    if ( ! UI::HasSpecialWidget(`Slider) ||
         ! UI::HasSpecialWidget(`BarGraph ) )
    {
        UI::OpenDialog(
                   `VBox(
                         `Label("Error: This UI doesn't support the required special widgets!"),
                         `PushButton(`opt(`default), "&OK")
                         )
                   );
        UI::UserInput();
        UI::CloseDialog();

        return;
    }


    // Initialize RGB values

    integer red   = 128;
    integer blue  = 128;
    integer green = 128;


    // Create the dialog

    UI::OpenDialog(
               `VBox(
                     `HSpacing(50), // force width
                     `BarGraph( `id(`graph),
                                [ red, green, blue ],
                                [ "Red\n%1", "Green\n%1", "Blue\n%1" ] ),
                     `Slider( `id(`red),   `opt(`notify), "Red",   0, 255, red   ),
                     `Slider( `id(`green), `opt(`notify), "Green", 0, 255, green ),
                     `Slider( `id(`blue),  `opt(`notify), "Blue",  0, 255, blue  ),
                     `PushButton(`id(`close), `opt(`default), "&Close")
                     )
               );


    // Event processing loop - left only via the "close" button
    // or the window manager close button / function.
    any widget = nil;

    do
    {
        widget = UI::UserInput();

        if ( widget == `red   || // any of the sliders?
             widget == `blue  ||
             widget == `green   )
        {
            // Get all slider values

            red   = (integer) UI::QueryWidget(`id(`red),   `Value );
            green = (integer) UI::QueryWidget(`id(`green), `Value );
            blue  = (integer) UI::QueryWidget(`id(`blue),  `Value );


            // Update bar graph

            UI::ChangeWidget(`id(`graph), `Values, [ red, green, blue ] );
        }
    } while ( widget != `close && // the real "Close" button
              widget != `cancel ); // the window manager close function/button

    UI::CloseDialog();
}
```



```
         /**
 * Advanced ColoredLabel example: A sample ColoredLabel widget and sliders
 * for both foreground and background colors.
 **/
{
    if ( ! UI::HasSpecialWidget(`ColoredLabel) || ! UI::HasSpecialWidget(`Slider) )
    {
        UI::OpenDialog(
                   `VBox(
                         `Label("Error: This UI doesn't support the required special widgets!"),
                         `PushButton(`opt(`default), "&OK")
                         )
                   );
        UI::UserInput();
        UI::CloseDialog();

        return;
    }

    define term sample( integer fg_red, integer fg_green, integer fg_blue,
                        integer bg_red, integer bg_green, integer bg_blue ) ``{
        return `ColoredLabel( `opt(`hstretch),
                              "Use the sliders\nto change color",
                              `rgb( fg_red, fg_green, fg_blue ),
                              `rgb( bg_red, bg_green, bg_blue ),
                              30 );
    };
    UI::OpenDialog(
               `VBox(
                     `ReplacePoint(`id(`sample), sample( 200, 0, 0,
                                                         0, 0, 200) ),
                     `VSpacing(),
                     `HBox(
                           `Frame( "Foreground",
                                   `VBox(
                                         `Slider(`id(`fg_red  ),`opt(`notify), "&red",   0, 255, 200 ),
                                         `Slider(`id(`fg_green),`opt(`notify), "&green", 0, 255,   0 ),
                                         `Slider(`id(`fg_blue ),`opt(`notify), "&blue",  0, 255,   0 )
                                         )
                                   ),
                           `Frame( "Backround",
                                   `VBox(
                                         `Slider(`id(`bg_red  ), `opt(`notify), "r&ed",   0, 255,   0 ),
                                         `Slider(`id(`bg_green), `opt(`notify), "gree&n", 0, 255,   0 ),
                                         `Slider(`id(`bg_blue ), `opt(`notify), "b&lue",  0, 255, 200 )
                                         )
                                   )
                           ),
                     `VSpacing(),
                     `PushButton(`id(`close), "&Close")
                     )
               );

    any widget = nil;

    do
    {
        widget = UI::UserInput();

        if ( widget != `close )
        {
            UI::ReplaceWidget(`id(`sample),
                              sample( (integer) UI::QueryWidget(`id(`fg_red  ), `Value ),
                                      (integer) UI::QueryWidget(`id(`fg_green), `Value ),
                                      (integer) UI::QueryWidget(`id(`fg_blue ), `Value ),

                                      (integer) UI::QueryWidget(`id(`bg_red  ), `Value ),
                                      (integer) UI::QueryWidget(`id(`bg_green), `Value ),
                                      (integer) UI::QueryWidget(`id(`bg_blue ), `Value )
                                      )
                              );

        }

    } while ( widget != `close );

    UI::CloseDialog();
}
```


Use the sliders
to change color

Foreground
red 255
green 255
blue 114

Backround
red 41
green 139
blue 148

Close

Name

PartitionSplitter -- Hard disk partition splitter tool (optional widget)

PartitionSplitter

```
PartitionSplitter (usedSize, totalFreeSize, newPartSize, minNew-
PartSize, minFreeSize, usedLabel, freeLabel, newPartLabel,
freeFieldLabel, newPartFieldLabel);
integer usedSize ;
integer totalFreeSize ;
integer newPartSize ;
integer minNewPartSize ;
integer minFreeSize ;
string usedLabel ;
string freeLabel ;
string newPartLabel ;
string freeFieldLabel ;
string newPartFieldLabel ;
```

Parameters

integer *usedSize*	size of the used part of the partition
integer *totalFreeSize*	total size of the free part of the partition (before the split)
integer *newPartSize*	suggested size of the new partition
integer *minNewPartSize*	minimum size of the new partition
integer *minFreeSize*	minimum free size of the old partition
string *usedLabel*	BarGraph label for the used part of the old partition
string *freeLabel*	BarGraph label for the free part of the old partition
string *newPartLabel*	BarGraph label for the new partition
string *freeFieldLabel*	label for the remaining free space field
string *newPartFieldLabel*	label for the new size field

Properties

integer *Value*	the numerical value

Description

A very specialized widget to allow a user to comfortably split an existing hard disk partition in two parts. Shows a bar graph that displays the used space of the partition, the remaining free space (before the split) of the partition and the space of the new partition (as suggested). Below the bar graph is a slider with an input fields to the left and right where the user can either input the desired remaining free space or the desired size of the new partition or drag the slider to do this.

The total size is *usedSize+freeSize*.

The user can resize the new partition between `minNewPartSize` and `totalFreeSize-minFreeSize`.

Note

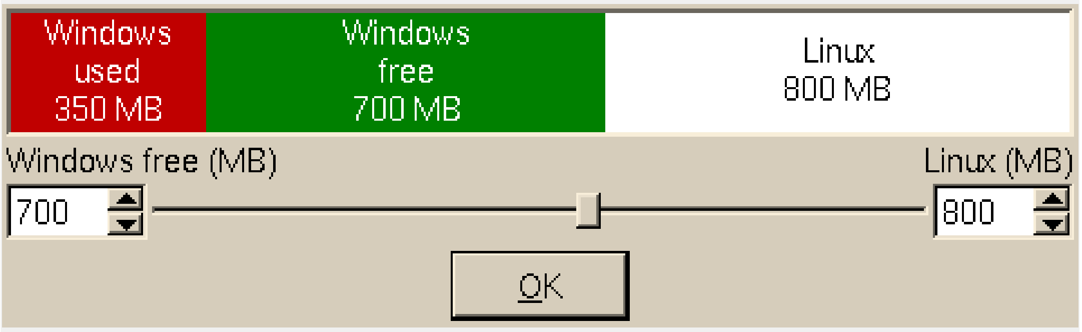
This is a "special" widget, i.e. not all UIs necessarily support it. Check for availability with `HasSpecialWidget( `PartitionSplitter)` before using it.

Usage

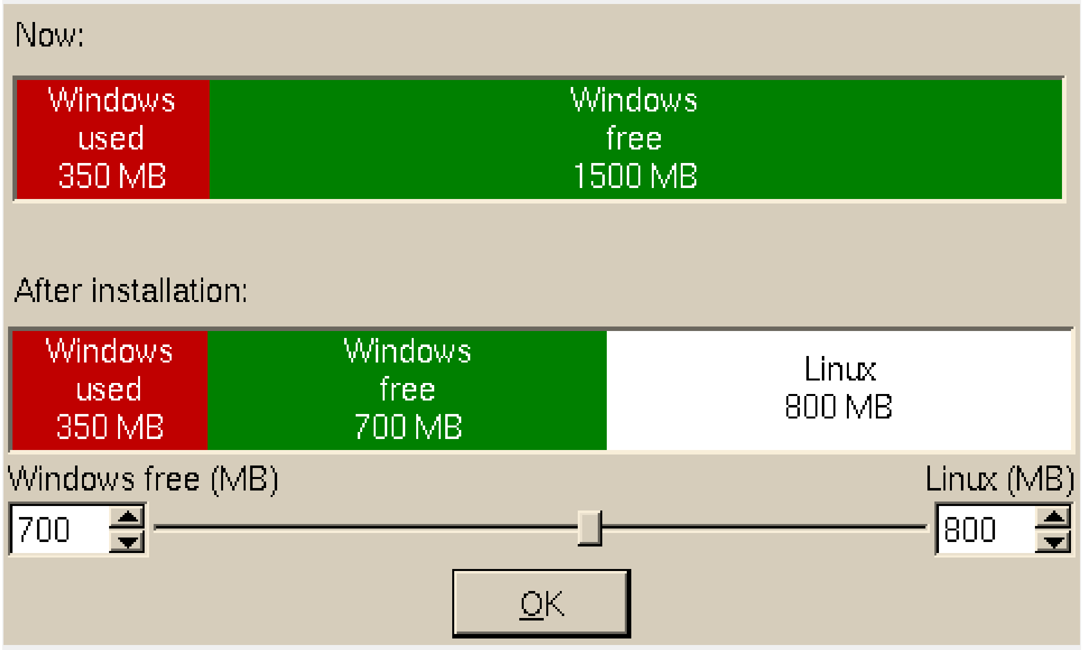
```
        if ( HasSpecialWidget( `PartitionSplitter ) {...
        `PartitionSplitter( 600, 1200, 800, 300, 50,
                            "Windows used\n%1 MB", "Windows used\n%1 MB", "Linux\n%1 MB", "Linux ( MB )" )
```

Examples

```
          {
    if ( ! UI::HasSpecialWidget(`PartitionSplitter) )
    {
        UI::OpenDialog(
                   `VBox(
                         `Label("Error: This UI doesn't support the PartitionSplitter widget!"),
                         `PushButton(`opt(`default), "&OK")
                         )
                   );
        UI::UserInput();
        UI::CloseDialog();

        return;
    }

    string  unit = "MB";
    integer win_used = 350;
    integer total_free = 1500;
    integer min_free = 50;
    integer linux_min = 300;
    integer linux_size = 800;

    UI::OpenDialog(
               `VBox(
                     `HSpacing( 60 ), // wider default size
                     `PartitionSplitter( win_used, total_free,
                                         linux_size, linux_min, min_free,
                                         "Windows\nused\n%1 " + unit,
                                         "Windows\nfree\n%1 "         + unit,
                                         "Linux\n%1 "                 + unit,
                                         "Windows free (" + unit + ")",
                                         "Linux (" + unit + ")"
                                         ),
                     `PushButton(`opt(`default), "&OK")
                     )
               );
    UI::UserInput();
    UI::CloseDialog();
}
```



```
          {
    if ( ! UI::HasSpecialWidget(`Slider) ||
         ! UI::HasSpecialWidget(`BarGraph ) )
    {
        UI::OpenDialog(
                   `VBox(
                         `Label("Error: This UI doesn't support the required special widgets!"),
                         `PushButton(`opt(`default), "&OK")
                         )
                   );
        UI::UserInput();
        UI::CloseDialog();

        return;
    }

    string  unit = "MB";
    integer win_used = 350;
    integer total_free = 1500;
    integer min_free = 50;
    integer linux_min = 300;
    integer linux_size = 800;

    UI::OpenDialog(
               `VBox(
                     `HSpacing( 60 ), // wider default size
                     `Left( `Label( "Now:") ),
                     `BarGraph( `opt(`vstretch),
                                [ win_used, total_free ],
                                [
                                 "Windows\nused\n%1 " + unit,
                                 "Windows\nfree\n%1 " + unit
                                ]
                                ),
                     `VSpacing(1),
                     `Left( `Label( "After installation:" ) ),
                     `PartitionSplitter( win_used, total_free,
                                         linux_size, linux_min, min_free,
                                         "Windows\nused\n%1 " + unit,
                                         "Windows\nfree\n%1 "          + unit,
                                         "Linux\n%1 "                  + unit,
                                         "Windows free (" + unit + ")",
                                         "Linux (" + unit + ")"
                                         ),
                     `PushButton(`opt(`default), "&OK")
                     )
               );
    UI::UserInput();
    UI::CloseDialog();
}
```



Name

Date -- Date input field

Date

```
Date (label);
string label ;
```

Parameters

string *label*

Properties

string *Value*	the date

Description

TODO - contact nashif@suse.de

Usage

```
        `Date( "Date:", "2004-10-12" )
```

Name

Time -- Time input field

Time

```
Time (label);
string label ;
```

Parameters

string *label*

Properties

string *Value*	the date

Description

TODO - contact nashif@suse.de

Usage

```
        `Time( "Time:" , "20:20:20" )
```

Name

Wizard -- Wizard frame - not for general use, use the Wizard:: module instead!

Wizard

```
Wizard (backButtonId, backButtonLabel, abortButtonId, abortButton-
Label, nextButtonId, nextButtonLabel);
any backButtonId ;
string backButtonLabel ;
any abortButtonId ;
string abortButtonLabel ;
any nextButtonId ;
string nextButtonLabel ;
```

Parameters

any *`backButtonId`*	ID to return when the user presses the "Back" button
string *`backButtonLabel`*	Label of the "Back" button
any *`abortButtonId`*	ID to return when the user presses the "Abort" button
string *`abortButtonLabel`*	Label of the "Abort" button
any *`nextButtonId`*	ID to return when the user presses the "Next" button
string *`nextButtonLabel`*	Label of the "Next" button

Options

`stepsEnabled`
: Enable showing wizard steps (use UI::WizardCommand() to set them).

`treeEnabled`
: Enable showing a selection tree in the left panel. Disables stepsEnabled.

Description

This is the UI-specific technical implementation of a wizard dialog's main widget. This is not intended for general use - use the Wizard:: module instead which will use this widget properly.

A wizard widget always has ID `wizard.<p> The ID of the single replace point within the wizard is always `contents.

Note

This is a "special" widget, i.e. not all UIs necessarily support it. Check for availability with *`HasSpecialWidget( `PartitionSplitter)`* before using it.

Usage

Usage

```
        `Wizard(`id(`back), "&Back", `id(`abort), "Ab&ort", `id(`next), "&Next" )
        `Wizard(`back, "&Back", `abort, "Ab&ort", `next, "&Next" )
```

Part III. Frameworks Reference

Table of Contents

Proposal API Reference

Name

MakeProposal -- Make proposal for installation.

MakeProposal

```
map MakeProposal(force_reset, language_changed);
boolean force_reset;
boolean language_changed;
```

Parameters

boolean *force_reset*	If 'true', discard anything that may be cached and start over from scratch. If 'false', use cached values from the last invocation if there are any.
boolean *language_changed*	The installation language has changed since the last call of "MakeProposal". This is important only if there is a language change mechanism in one of the other submodules. If this parameter is "true", any texts the user can see in the proposal need to be retranslated. The internal translator mechanism will take care of this itself if the corresponding strings are once more put through it (the _("...") function). Only very few submodules that translate any strings internally based on internal maps (e.g., keyboard etc.) need to take more action.

Return Values

`MakeProposal()` returns a map containing:

list<string> links

A list of additional hyperlink ids used in summaries returned by this section. All possible values must be included.

Example: ["device_enable", "device_test"]

string preformatted_proposal (optional)

Human readable proposal preformatted in HTML.

Tip

Use the HTML:: module for such formatting.

list *raw_proposal*

(only used if 'preformatted_proposal' is not present in the result map)

Human readable proposal, not formatted yet. The caller will format each list item (string) as a HTML list item ("<li> ... </li>").

The proposal can contain hyperlinks with ids listed in the list *links*.

The caller will make a HTML unsorted list of this, e.g.:

```
<ul>
<li>string from list element #1</li>
<li>string from list element #2</li>
<li>string from list element #3</li>
...
</ul>
```

string *warning* (optional)

Warning in human readable format without HTML tags other than <br>.

The warning will be embedded in appropriate HTML format specifications ac-

	cording to 'warning_level' below.
symbol *`warning_level`* (optional)	Determines the severity and the visual display of the warning. Valid values: • \`notice • \`warning (default) • \`error • \`blocker • \`fatal *\`blocker* will prevent the user from continuing the installation. If any proposal contains a \`blocker warning, the "accept" button in the proposal dialog will be disabled - the user needs to fix that blocker before continuing. *\`fatal* is like \`blocker but also stops building the proposal
boolean *`language_changed`* (optional)	This module just caused a change of the installation language. This is only relevant for the "language" module.
boolean *`mode_changed`* (optional)	This module just caused a change of the installation mode. This is only relevant for the "inst mode" module.
boolean *`rootpart_changed`* (optional)	This module just caused a change of the root partition. This is only relevant for the "root part" module.

Name

AskUser -- Run an interactive workflow

AskUser

```
map AskUser(has_next, chosen_id);
boolean has_next;
string chosen_id;
```

Parameters

boolean *has_next*	Use a "next" button even if the module by itself has only one step, thus would normally have an "OK" button - or would rename the "next" button to something like "finish" or "accept".
string *chosen_id*	If a section proposal contains hyperlinks and user clicks on one of them, this defines the id of the hyperlink. All hyperlink IDs must be set in the map retuned by `Description`. If a user didn't click on a proposal hyperlink, this parameter is not defined.

Description

Run an interactive workflow - let user decide upon values he might want to change. May contain one single dialog, a sequence of dialogs or one master dialog with one or more "expert" dialogs. The module is responsible for controlling the workflow sequence (i.e., "next", "back" buttons etc.).

Submodules don't provide an "abort" button to abort the entire installation. If the user wishes to do that, he can always go back to the main dialog (the installation proposal) and choose "abort" there.

Return Values

`AskUser()` returns a map containing:

symbol workflow_sequence	`next (default)	Everything OK - continue with the next step in workflow sequence.
	`back	User requested to go back in the workflow sequence.
	`again	Call this submodule again (i.e., re-initialize the submodule)
	`auto	Continue with the workflow sequence in the current direction - forward if the last submodule returned `next, backward otherwise.
	`finish	Finish the installation. This is specific to "inst_mode.ycp" when the user selected "boot system" there.
boolean language_changed (optional)	This module just caused a change of the installation language. This is only relevant for the "language" module.	

Name

Description -- Return human readable titles both for the RichText (HTML) widget and for menu-entries.

Description

```
map Description();
```

Return Values

Returns a map containing:

string rich_text _title	(Translated) human readable title for this section in the RichText widget without any HTML formatting. This will be embedded in `<h3><a href="???"> ... </a></h3>` so make sure not to add any additional HTML formatting. Keyboard shortcuts are not (yet?) supported, so don't include any '&' characters. Example: "Input devices"
string menu_tit le	(Translated) human readable menuentry for this section. Must contain a keyboard short-cut ('&'). Should NOT contain trailing periods ('...') - the caller will add them. Example: "&Input devices"
string id	Programmer readable unique identifier for this section. This is not auto-generated to keep the log file readable. Example: "input_dev"

This map may be empty. In this case, this proposal section will silently be ignored. Proposals modules may use this if there is no useful proposal at all. Use with caution - this may be confusing for the user.

Note: In this case, all other proposal functions must return a useful success value so they can be called without problems.

Name

Write -- Write the proposed (and probably modified) settings to the system.

Write

```
map Write();
```

Description

Write the proposed (and probably modified) settings to the system. It is up to the proposal dispatcher how it remembers the settings. The errors must be reported using the Report:: module to have the possibility to control the behaviour from the main program.

This Write() function is optional. The dispatcher module is required to allow this function to be called without returning an error value if it isn't there.

Return Values

Returns a map containing:

boolean success	Returns true, if the settings were written successfully.

Part IV. YaST2 Library Reference

Table of Contents

Address manipulation routines

Name

Address::Check -- Check syntax of a network address (IP address or hostname)

Address::Check

Import Address;

```
boolean Address::Check (address);
string address ;
```

Parameters

string *address*	an address

Return Value

boolean	true if correct

Name

Address::Check4 -- Check syntax of a network address (ip4 or name)

Address::Check4

Import Address;

```
boolean Address::Check4 (address);
string address ;
```

Parameters

string *address*	an address

Return Value

boolean	true if correct

Name

Address::Check6 -- Check syntax of a network address (ip6 or name)

Address::Check6

Import Address;

```
boolean Address::Check6 (address);
string address ;
```

Parameters

string *address*	an address

Return Value

boolean	true if correct

Name

Address::Valid4 -- Return a description of a valid address (ip4 or name)

Address::Valid4

Import Address;

`string **Address::Valid4** ();`

Return Value

string	description

Confirmation routines

Name

Confirm::Delete -- Opens a popup yes/no confirmation.

Confirm::Delete

Import Confirm;

```
boolean Confirm::Delete (delete);
string delete ;
```

Parameters

string *delete*

Return Value

boolean	delete selected entry

Description

If users confirms deleting of named entry/file/etc., return true, else return false

Name

Confirm::DeleteSelected -- Opens a popup yes/no confirmation.

Confirm::DeleteSelected

Import Confirm;

```
boolean Confirm::DeleteSelected ();
```

Return Value

boolean	delete selected entry

Description

If users confirms deleting, return true, else return false

Name

Confirm::Detection -- Confirm hardware detection (only in manual installation)

Confirm::Detection

Import Confirm;

```
boolean Confirm::Detection (class);
string class ;
```

Parameters

string *class*	hardware class (network cards)

Return Value

boolean	true on continue

Name

Confirm::MustBeRoot -- If we are running as root, return true. Otherwise ask the user whether we should continue even though things might not work

Confirm::MustBeRoot

Import Confirm;

```
boolean Confirm::MustBeRoot ();
```

Return Value

boolean	true if running as root

Hostname manipulation routines

Name

Hostname::Check -- Check syntax of hostname entry (that is a domain name component, unqualified, without dots)

Hostname::Check

Import Hostname;

```
boolean Hostname::Check (host);
string host ;
```

Parameters

string `host`	hostname

Return Value

boolean	true if correct

See

- rfc1123, rfc2396 and obsoleted rfc1034

Name

Hostname::CheckDomain -- Check syntax of domain entry

Hostname::CheckDomain

Import Hostname;

```
boolean Hostname::CheckDomain (domain);
string domain ;
```

Parameters

string *domain*	domain name

Return Value

boolean	true if correct

Name

Hostname::CheckFQ -- Check syntax of fully qualified hostname

Hostname::CheckFQ

Import Hostname;

```
boolean Hostname::CheckFQ (host);
string host ;
```

Parameters

string *host*	hostname

Return Value

boolean	true if correct

Name

Hostname::MergeFQ -- Merge short hostname and domain to full-qualified host name

Hostname::MergeFQ

Import Hostname;

```
string Hostname::MergeFQ (hostname, domain);
string hostname ;
string domain ;
```

Parameters

string *hostname*	short host name
string *domain*	domain name

Return Value

string	FQ hostname

Name

Hostname::SplitFQ -- Split FQ hostname to hostname and domain name

Hostname::SplitFQ

Import Hostname;

```
list<string> Hostname::SplitFQ (fqhostname);
string fqhostname ;
```

Parameters

string *fqhostname*	FQ hostname

Return Value

list<string>	of hostname and domain name

Examples

```
Hostname::SplitFQ("ftp.suse.cz") -> ["ftp", "suse.cz"]
Hostname::SplitFQ("ftp") -> ["ftp"]
```

Name

Hostname::ValidDomain -- describe a valid domain name

Hostname::ValidDomain

Import Hostname;

```
string Hostname::ValidDomain ();
```

Return Value

string	describtion

Name

Hostname::ValidFQ -- describe a valid FQ host name

Hostname::ValidFQ

Import Hostname;

```
string Hostname::ValidFQ ();
```

Return Value

string	describe a valid FQ host name

Generic HTML formatting

Name

HTML::Bold -- Make a piece of HTML code bold

HTML::Bold

Import HTML;

```
string HTML::Bold (text);
string text ;
```

Parameters

string *text*	text to make bold

Return Value

string	HTML code

Description

i.e. embed it into [b]...[/b]

You still need to embed that into a paragraph or heading etc.!

Name

HTML::ColoredList -- Make a HTML (unsorted) colored list from a list of strings

HTML::ColoredList

Import HTML;

```
string HTML::ColoredList (items, color);
list<string> items ;
string color ;
```

Parameters

list<string> *items*	list of strings for items
string *color*	item color

Return Value

string	HTML code

Description

[ul] [li][font color="..."]...[/font][/li] [li][font color="..."]...[/font][/li] ... [/ul]

Name

HTML::Colorize -- Colorize a piece of HTML code

HTML::Colorize

Import HTML;

```
string HTML::Colorize (text, color);
string text ;
string color ;
```

Parameters

string *text*	text to colorize
string *color*	item color

Return Value

string	HTML code

Description

i.e. embed it into [font color="..."]...[/font]

You still need to embed that into a paragraph or heading etc.!

Name

HTML::Heading -- Make a HTML heading from a text

HTML::Heading

Import HTML;

```
string HTML::Heading (text);
string text ;
```

Parameters

string *text*	plain text or HTML fragment

Return Value

string	HTML code

Description

i.e. embed a text into [h3]...[/h3]

Note: There is only one heading level here since we don't have any more fonts anyway.

Name

HTML::Link -- Make a HTML link

HTML::Link

Import HTML;

```
string HTML::Link (text, link_id);
string text ;
string link_id ;
```

Parameters

string *text*	(translated) text the user will see
string *link_id*	internal ID of that link returned by UserInput()

Return Value

string	HTML code

Description

For example [a href="..."]...[/a]

You still need to embed that into a paragraph or heading etc.!

Name

HTML::List -- Make a HTML (unsorted) list from a list of strings

HTML::List

Import HTML;

```
string HTML::List (items);
list<string> items ;
```

Parameters

list<string> *items*	list of strings for items

Return Value

string	HTML code

Description

[ul] [li]...[/li] [li]...[/li] ... [/ul]

Name

HTML::ListEnd -- End a HTML (unsorted) list

HTML::ListEnd

Import HTML;

```
string HTML::ListEnd ();
```

Return Value

string	HTML code

Description

For example [/ul]

You might consider using HTML::list() instead which takes a list of items and does all the rest by itself.

Name

HTML::ListItem -- Make a HTML list item

HTML::ListItem

Import HTML;

```
string HTML::ListItem (text);
string text ;
```

Parameters

string *text*	plain text or HTML fragment

Return Value

string	HTML code

Description

For example embed a text into [li][p]...[/p][/li]

You might consider using HTML::list() instead which takes a list of items and does all the rest by itself.

Name

HTML::ListStart -- Start a HTML (unsorted) list

HTML::ListStart

Import HTML;

`string` **`HTML::ListStart`** `();`

Return Value

string	HTML code

Description

For example [ul]

You might consider using HTML::list() instead which takes a list of items and does all the rest by itself.

Name

HTML::Newline -- Make a forced HTML line break

HTML::Newline

Import HTML;

```
string HTML::Newline ();
```

Return Value

string	HTML code

Name

HTML::Newlines -- Make a number of forced HTML line breaks

HTML::Newlines

Import HTML;

```
string HTML::Newlines (count);
integer count ;
```

Parameters

integer *count*	how many of them

Return Value

string	HTML code

Name

HTML::Para -- Make a HTML paragraph from a text

HTML::Para

Import HTML;

```
string HTML::Para (text);
string text ;
```

Parameters

string *text*	plain text or HTML fragment

Return Value

string	HTML code

Description

i.e. embed a text into * [p]...[/p]

IP manipulation routines

Name

IP::Check -- Check syntax of IP address

IP::Check

Import IP;

```
boolean IP::Check (ip);
string ip ;
```

Parameters

string *ip*	IP address

Return Value

boolean	true if correct

Name

IP::Check4 -- Check syntax of IPv4 address

IP::Check4

Import IP;

```
boolean IP::Check4 (ip);
string ip ;
```

Parameters

string *ip*	IPv4 address

Return Value

boolean	true if correct

Name

IP::Check6 -- Check syntax of IPv6 address

IP::Check6

Import IP;

```
boolean IP::Check6 (ip);
string ip ;
```

Parameters

string *ip*	IPv6 address

Return Value

boolean	true if correct

Name

IP::ComputeBroadcast -- Compute IPv4 broadcast address from ip4 address and network mask.

IP::ComputeBroadcast

Import IP;

```
string IP::ComputeBroadcast (ip, mask);
string ip ;
string mask ;
```

Parameters

string *ip*	IPv4 address
string *mask*	netmask

Return Value

string	computed broadcast

Name

IP::ComputeNetwork -- Compute IPv4 network address from ip4 address and network mask.

IP::ComputeNetwork

Import IP;

```
string IP::ComputeNetwork (ip, mask);
string ip ;
string mask ;
```

Parameters

string *ip*	IPv4 address
string *mask*	netmask

Return Value

string	computed subnet

Name

IP::ToHex -- Converts IPv4 address from string to hex format

IP::ToHex

Import IP;

```
string IP::ToHex (ip);
string ip ;
```

Parameters

string *ip*	IPv4 address as string in "ipv4" format

Return Value

string	representing IP in Hex

Examples

```
IP::ToHex("192.168.1.1") -> "0xC0A80101"
IP::ToHex("10.10.0.1") -> "0x0A0A0001"
```

Name

IP::ToInteger -- Convert IPv4 address from string to integer

IP::ToInteger

Import IP;

```
integer IP::ToInteger (ip);
string ip ;
```

Parameters

string *ip*	IPv4 address

Return Value

integer	ip address as integer

Name

IP::ToString -- Convert IPv4 address from integer to string

IP::ToString

Import IP;

```
string IP::ToString (ip);
integer ip ;
```

Parameters

integer *ip*	IPv4 address

Return Value

string	ip address as string

Name

IP::Valid4 -- Describe a valid IPv4 address

IP::Valid4

Import IP;

`string **IP::Valid4** ();`

Return Value

string	describtion a valid IPv4 address

Map manipulation routines

Name

Map::CheckKeys -- Check if a map contains all needed keys

Map::CheckKeys

Import Map;

```
boolean Map::CheckKeys (m, keys);
map m ;
list keys ;
```

Parameters

map *m*	map to be checked
list *keys*	needed keys

Return Value

boolean	true if map kontains all keys

Name

Map::FromString -- Convert string "var=val ..." to map $[val:var, ...]

Map::FromString

Import Map;

```
map **Map::FromString** (s);
string s ;
```

Parameters

string *s*	string to be converted

Return Value

map	converted string

Name

Map::Keys -- Return all keys from the map

Map::Keys

Import Map;

```
list Map::Keys (m);
map m ;
```

Parameters

map *m*	the map

Return Value

list	a list of all keys from the map

Name

Map::KeysToLower -- Switch map keys to lower case

Map::KeysToLower

Import Map;

```
map Map::KeysToLower (m);
map<string, any> m ;
```

Parameters

map<string, any> *m*	input map

Return Value

map	with keys converted to lower case

Name

Map::KeysToUpper -- Switch map keys to upper case

Map::KeysToUpper

Import Map;

```
map Map::KeysToUpper (m);
map<string, any> m ;
```

Parameters

map<string, any> *m*	input map

Return Value

map	with keys converted to lower case

Name

Map::ToString -- Convert options map $[var:val, ...] to string "var=val ..."

Map::ToString

Import Map;

```
string Map::ToString (m);
map m ;
```

Parameters

map *m*	map to be converted

Return Value

string	converted map

Name

Map::Values -- Return all values from the map

Map::Values

Import Map;

```
list Map::Values (m);
map m ;
```

Parameters

map *m*	the map

Return Value

list	a list of all values from the map

Netmask manipulation routines

Name

Netmask::Check -- Check the netmask

Netmask::Check

Import Netmask;

```
boolean Netmask::Check (netmask);
string netmask ;
```

Parameters

string *netmask*	network mask

Return Value

boolean	true if correct

Name

Netmask::Check4 -- Check the IPv4 netmask Note that 0.0.0.0 is not a correct netmask.

Netmask::Check4

Import Netmask;

```
boolean Netmask::Check4 (netmask);
string netmask ;
```

Parameters

string *netmask*	network mask

Return Value

boolean	true if correct

Name

Netmask::Check6 -- Check the IPv6 netmask

Netmask::Check6

Import Netmask;

```
boolean Netmask::Check6 (netmask);
string netmask ;
```

Parameters

string *netmask*	network mask

Return Value

boolean	true if correct

Name

Netmask::FromBits -- Convert netmask in bits form (24) to netmask string (255.255.240.0)

Netmask::FromBits

Import Netmask;

```
string Netmask::FromBits (bits);
integer bits ;
```

Parameters

integer *bits*	number of bits in netmask

Return Value

string	netmask string

Name

Netmask::ToBits -- Convert IPv4 netmask as string (255.255.240.0) to bits form (24)

Netmask::ToBits

Import Netmask;

```
integer Netmask::ToBits (netmask);
string netmask ;
```

Parameters

string *netmask*	netmask as string

Return Value

integer	number of bits in netmask

Popup dialogs for browsing the local network

Name

NetworkPopup::ChooseItem -- Let the user choose one of a list of items

NetworkPopup::ChooseItem

Import NetworkPopup;

```
string NetworkPopup::ChooseItem (title, items, selected);
string title ;
list<string> items ;
string selected ;
```

Parameters

string *title*	selectionbox title
list<string> *items*	a list of items
string *selected*	preselected a value in the list

Return Value

string	one item or nil

Name

NetworkPopup::HostName -- Give me one host name on the local network

NetworkPopup::HostName

Import NetworkPopup;

```
string NetworkPopup::HostName (selected);
string selected ;
```

Parameters

string *selected*	preselect a value in the list

Return Value

string	a hostname or nil if "Cancel" was pressed

Name

NetworkPopup::NFSExport -- Give me export path of selected server

NetworkPopup::NFSExport

Import NetworkPopup;

```
string NetworkPopup::NFSExport (server, selected);
string server ;
string selected ;
```

Parameters

string *server*	a NFS server name
string *selected*	preselected a value in the list

Return Value

string	an export or nil if "Cancel" was pressed

Name

NetworkPopup::NFSServer -- Give me NFS server name on the local network

NetworkPopup::NFSServer

Import NetworkPopup;

```
string NetworkPopup::NFSServer (selected);
string selected ;
```

Parameters

string *selected*	preselected a value in the list

Return Value

string	a hostname or nil if "Cancel" was pressed

Packages manipulation

Name

Package::DoInstall -- Install list of packages

Package::DoInstall

Import Package;

```
boolean Package::DoInstall (packages);
list<string> packages ;
```

Parameters

list<string> *packages*	list of packages to be installed

Return Value

boolean	True on success

Name

Package::DoInstallAndRemove -- Install and Remove list of packages in one go

Package::DoInstallAndRemove

Import Package;

```
boolean Package::DoInstallAndRemove (toinstall, toremove);
list<string> toinstall ;
list<string> toremove ;
```

Parameters

list<string> *toinstall*	list of packages to be installed
list<string> *toremove*	list of packages to be removed

Return Value

boolean	True on success

Name

Package::DoRemove -- Remove list of packages

Package::DoRemove

Import Package;

```
boolean Package::DoRemove (packages);
list<string> packages ;
```

Parameters

list<string> *packages*	list of packages to be removed

Return Value

boolean	True on success

Name

Package::AvailableAll -- Are all of these packages available?

Package::AvailableAll

Import Package;

```
boolean Package::AvailableAll (packages);
list<string> packages ;
```

Parameters

list<string> *packages*	list of packages

Return Value

boolean	true if yes

Name

Package::AvailableAny -- Is any of these packages available?

Package::AvailableAny

Import Package;

```
boolean Package::AvailableAny (packages);
list<string> packages ;
```

Parameters

list<string> *packages*	list of packages

Return Value

boolean	true if yes

Name

Package::InstallAllMsg -- Install list of packages with a custom text message

Package::InstallAllMsg

Import Package;

```
boolean **Package::InstallAllMsg** (packages, message);
list<string> packages ;
string message ;
```

Parameters

list<string> *packages*	The list packages to be installed
string *message*	custom text message

Return Value

boolean	True on success

Name

Package::InstallMsg -- Install a package with a custom text message

Package::InstallMsg

Import Package;

```
boolean **Package::InstallMsg** (package, message);
string package ;
string message ;
```

Parameters

string *package*	to be installed
string *message*	custom text message

Return Value

boolean	True on success

Name

Package::InstalledAll -- Are all of these packages installed?

Package::InstalledAll

Import Package;

```
boolean Package::InstalledAll (packages);
list<string> packages ;
```

Parameters

list<string> *packages*	list of packages

Return Value

boolean	true if yes

Name

Package::InstalledAny -- Is any of these packages installed?

Package::InstalledAny

Import Package;

```
boolean Package::InstalledAny (packages);
list<string> packages ;
```

Parameters

list<string> *packages*	list of packages

Return Value

boolean	true if yes

Name

Package::LastOperationCanceled -- Return result of the last operation Use immediately after calling any Package*:: function

Package::LastOperationCanceled

Import Package;

`boolean` **`Package::LastOperationCanceled`** `();`

Return Value

boolean	true if it last operation was canceled

Name

Package::PackageDialog -- Main package installatio|removal dialog

Package::PackageDialog

Import Package;

```
boolean Package::PackageDialog (packages, install, message);
list<string> packages ;
boolean install ;
string message ;
```

Parameters

list<string> *packages*	list of packages
boolean *install*	true if install, false if remove
string *message*	optional installation|removal text (nil -> standard will be used)

Return Value

boolean	true on success

Name

Package::RemoveAllMsg -- Remove a list of packages with a custom text message

Package::RemoveAllMsg

Import Package;

```
boolean Package::RemoveAllMsg (packages, message);
list<string> packages ;
string message ;
```

Parameters

list<string> *packages*	The list of packages to be removed
string *message*	custom text message

Return Value

boolean	True on success

Name

Package::RemoveMsg -- Remove a package with a custom text message

Package::RemoveMsg

Import Package;

```
boolean Package::RemoveMsg (package, message);
string package ;
string message ;
```

Parameters

string *package*	package to be removed
string *message*	custom text message

Return Value

boolean	True on success

Name

Package::RunSUSEconfig -- Run SUSEconfig, create new wizard dialog before it, close after it is finished

Package::RunSUSEconfig

Import Package;

`void` **`Package::RunSUSEconfig`** `();`

Return Value

void

Progress bar

Name

Progress::CloseSuperior -- Replaces stages of superior progress by an empty help text.

Progress::CloseSuperior

Import Progress;

`void` **`Progress::CloseSuperior`** `();`

Return Value

void

Name

Progress::Finish -- Moves progress bar to the end and marks all stages as completed.

Progress::Finish

Import Progress;

```
void Progress::Finish ();
```

Return Value

void

Name

Progress::New -- New complex progress bar with stages.

Progress::New

Import Progress;

```
void Progress::New (window_title, progress_title, length, stg,
tits, help_text);
string window_title ;
string progress_title ;
integer length ;
list<string> stg ;
list tits ;
string help_text ;
```

Parameters

string *window_title*	title of the window
string *progress_title*	title of the progress bar. Pass at least " " (one space) if you want some progress bar title.
integer *length*	number of steps. If 0, no progress bar is created, there are only stages and bottom title. THIS IS NOT NUMBER OF STAGES!
list<string> *stg*	list of strings - stage names. If it is nil, then there are no stages.
list *tits*	Titles corresponding to stages. When stage changes, progress bar title changes to one of these titles. May be nil/empty.
string *help_text*	help text

Return Value

void

Name

Progress::NextStage -- Advance stage, advance step by 1 and set progress bar caption to that defined in New.

Progress::NextStage

Import Progress;

```
void Progress::NextStage ();
```

Return Value

void

Name

Progress::NextStageStep -- Jumps to the next stage and sets step to st.

Progress::NextStageStep

Import Progress;

```
void Progress::NextStageStep (st);
integer st ;
```

Parameters

integer *st*	new progress bar value

Return Value

void

Name

Progress::NextStep -- Some people say it is the best operating system ever. But now to the function. Advances progress bar value by 1.

Progress::NextStep

Import Progress;

```
void Progress::NextStep ();
```

Return Value

void

Name

Progress::OpenSuperior -- Creates a higher-level progress bar made of stages. Currently it is placed instead of help text.

Progress::OpenSuperior

Import Progress;

```
void Progress::OpenSuperior (title, stages);
string title ;
list<string> stages ;
```

Parameters

string *title*	title of the progress...
list<string> *stages*	list of stage descriptions

Return Value

void

Name

Progress::Simple -- Create simple progress bar with no stages, only with progress bar.

Progress::Simple

Import Progress;

```
void Progress::Simple (window_title, progress_title, length, help_text);
string window_title ;
string progress_title ;
integer length ;
string help_text ;
```

Parameters

string *window_title*	Title of the window.
string *progress_title*	Title of the progress bar.
integer *length*	Number of steps.
string *help_text*	Help text.

Return Value

void

Name

Progress::Stage -- Go to stage st, change progress bar title to title and set progress bar step to step.

Progress::Stage

Import Progress;

```
void Progress::Stage (st, title, step);
integer st ;
string title ;
integer step ;
```

Parameters

integer *st*	New stage.
string *title*	New title for progress bar. If nil, title specified in New is used.
integer *step*	New step or -1 if step should not change.

Return Value

void

Name

Progress::Step -- Changes progress bar value to st.

Progress::Step

Import Progress;

```
void Progress::Step (st);
integer st ;
```

Parameters

integer *st*	new value

Return Value

void

Name

Progress::StepSuperior -- Make one step in a superior progress bar.

Progress::StepSuperior

Import Progress;

```
void Progress::StepSuperior ();
```

Return Value

void

Name

Progress::Title -- Change progress bar title.

Progress::Title

Import Progress;

```
void Progress::Title (t);
string t ;
```

Parameters

string *t*	new title. Use ""(empty string) if you want an empty progress bar.

Return Value

void

Name

Progress::off -- Turns progress bar off. All Progress:: calls return immediatelly.

Progress::off

Import Progress;

```
void Progress::off ();
```

Return Value

void

Name

Progress::on -- Turns progress bar on after calling Progress::off.

Progress::on

Import Progress;

```
void Progress::on ();
```

Return Value

void

Messages handling

Name

Report::AnyQuestion -- Question with headline and Yes/No Buttons

Report::AnyQuestion

Import Report;

```
boolean Report::AnyQuestion (headline, message, yes_button_message,
no_button_message, focus);
string headline ;
string message ;
string yes_button_message ;
string no_button_message ;
symbol focus ;
```

Parameters

string *headline*	Popup Headline
string *message*	Popup Message
string *yes_button_message*	Yes Button Message
string *no_button_message*	No Button Message
symbol *focus*	Which Button has the focus

Return Value

boolean	True if Yes is pressed, otherwise false

Name

Report::ClearAll -- Clear all stored messages (errors, messages and warnings)

Report::ClearAll

Import Report;

```
void Report::ClearAll ();
```

Return Value

void

Name

Report::ClearErrors -- Clear stored errors

Report::ClearErrors

Import Report;

```
void Report::ClearErrors ();
```

Return Value

void

Name

Report::ClearMessages -- Clear stored messages

Report::ClearMessages

Import Report;

```
void Report::ClearMessages ();
```

Return Value

void

Name

Report::ClearWarnings -- Clear stored warnings

Report::ClearWarnings

Import Report;

```
void Report::ClearWarnings ();
```

Return Value

void

Name

Report::ClearYesNoMessages -- Clear stored yes/no messages

Report::ClearYesNoMessages

Import Report;

```
void Report::ClearYesNoMessages ();
```

Return Value

void

Name

Report::DisplayErrors -- Error popup dialog can displayed immediately when new error is stored.

Report::DisplayErrors

Import Report;

```
void Report::DisplayErrors (display, timeout);
boolean display ;
integer timeout ;
```

Parameters

boolean *`display`*	if true then display error popups immediately
integer *`timeout`*	dialog is automatically closed after timeout seconds. Value 0 means no time out, dialog will be closed only by user.

Return Value

void

Description

This function enables or diables popuping of dialogs.

Name

Report::DisplayMessages -- Message popup dialog can be displayed immediately when a new message is stored.

Report::DisplayMessages

Import Report;

```
void Report::DisplayMessages (display, timeout);
boolean display ;
integer timeout ;
```

Parameters

boolean *display*	if true then display message popups immediately
integer *timeout*	dialog is automatically closed after timeout seconds. Value 0 means no time out, dialog will be closed only by user.

Return Value

void

Description

This function enables or diables popuping of dialogs.

Name

Report::DisplayWarnings -- Warning popup dialog can displayed immediately when new warningr is stored.

Report::DisplayWarnings

Import Report;

```
void Report::DisplayWarnings (display, timeout);
boolean display ;
integer timeout ;
```

Parameters

boolean *display*	if true then display warning popups immediately
integer *timeout*	dialog is automatically closed after timeout seconds. Value 0 means no time out, dialog will be closed only by user.

Return Value

void

Description

This function enables or diables popuping of dialogs.

Name

Report::DisplayYesNoMessages -- Yes/No Message popup dialog can be displayed immediately when a new message is stored.

Report::DisplayYesNoMessages

Import Report;

```
void Report::DisplayYesNoMessages (display, timeout);
boolean display ;
integer timeout ;
```

Parameters

boolean *display*	if true then display message popups immediately
integer *timeout*	dialog is automatically closed after timeout seconds. Value 0 means no time out, dialog will be closed only by user.

Return Value

void

Description

This function enables or diables popuping of dialogs.

Name

Report::Error -- Store new error text

Report::Error

Import Report;

```
void Report::Error (error_string);
string error_string ;
```

Parameters

string *error_string*	error text, it can contain new line characters ("\n")

Return Value

void

Name

Report::Export -- Dump the Report settings to a map, for autoinstallation use.

Report::Export

Import Report;

`map` **`Report::Export`** `();`

Return Value

map	Map with settings

Name

Report::GetMessages -- Create rich text string from stored warning, message or error messages.

Report::GetMessages

Import Report;

```
string Report::GetMessages (w, e, m, ynm);
boolean w ;
boolean e ;
boolean m ;
boolean ynm ;
```

Parameters

boolean *w*	include warnings in returned string
boolean *e*	include errors in returned string
boolean *m*	include messages in returned string
boolean *ynm*	include Yes/No messages in returned string

Return Value

string	rich text string

Description

Every new line character "\n" is replaced by string "[BR]".

Name

Report::GetModified -- Functions which returns if the settings were modified

Report::GetModified

Import Report;

`boolean` **`Report::GetModified`** `();`

Return Value

boolean	settings were modified

Name

Report::Import -- Get all the Report configuration from a map.

Report::Import

Import Report;

```
boolean **Report::Import** (settings);
map settings ;
```

Parameters

map *settings*	Map with settings (keys: "messages", "errors", "warnings"; values: map

Return Value

boolean	success

Description

the map may be empty.

Name

Report::LogErrors -- Set warnings logging to .y2log file

Report::LogErrors

Import Report;

```
void Report::LogErrors (log);
boolean log ;
```

Parameters

boolean *log*	if log is true then warning messages will be logged

Return Value

void

Name

Report::LogMessages -- Set messages logging to .y2log file

Report::LogMessages

Import Report;

```
void Report::LogMessages (log);
boolean log ;
```

Parameters

boolean *log*	if log is true then messages will be logged

Return Value

void

Name

Report::LogWarnings -- Set warnings logging to .y2log file

Report::LogWarnings

Import Report;

```
void Report::LogWarnings (log);
boolean log ;
```

Parameters

boolean *log*	if log is true then warning messages will be logged

Return Value

void

Name

Report::LogYesNoMessages -- Set yes/no messages logging to .y2log file

Report::LogYesNoMessages

Import Report;

```
void Report::LogYesNoMessages (log);
boolean log ;
```

Parameters

boolean *log*	if log is true then messages will be logged

Return Value

void

Name

Report::Message -- Store new message text

Report::Message

Import Report;

```
void Report::Message (message_string);
string message_string ;
```

Parameters

string *message_string*	message text, it can contain new line characters ("\n")

Return Value

void

Name

Report::NumErrors -- Return number of stored errors

Report::NumErrors

Import Report;

```
integer Report::NumErrors ();
```

Return Value

integer	number of errors

Name

Report::NumMessages -- Return number of stored messages

Report::NumMessages

Import Report;

```
integer Report::NumMessages ();
```

Return Value

integer	number of messages

Name

Report::NumWarnings -- Return number of stored warnings

Report::NumWarnings

Import Report;

```
integer Report::NumWarnings ();
```

Return Value

integer	number of warnings

Name

Report::NumYesNoMessages -- Return number of stored yes/no messages

Report::NumYesNoMessages

Import Report;

```
integer Report::NumYesNoMessages ();
```

Return Value

integer	number of messages

Name

Report::SetModified -- Function sets internal variable, which indicates, that any settings were modified, to "true"

Report::SetModified

Import Report;

`void` **`Report::SetModified`** `();`

Return Value

void

Name

Report::ShowText -- Store new message text

Report::ShowText

Import Report;

```
void Report::ShowText (headline_string, message_string);
string headline_string ;
string message_string ;
```

Parameters

string *headline_string*	
string *message_string*	message text, it can contain new line characters ("\n")

Return Value

void

Name

Report::Summary -- Summary of current settings

Report::Summary

Import Report;

```
string Report::Summary ();
```

Return Value

string	Html formatted configuration summary

Name

Report::Warning -- Store new warning text

Report::Warning

Import Report;

```
void Report::Warning (warning_string);
string warning_string ;
```

Parameters

string *warning_string*	warning text, it can contain new line characters ("\n")

Return Value

void

Rich text manipulation routines

Name

RichText::Rich2Plain -- Convert a richtext string into a formatted plain text.

RichText::Rich2Plain

Import RichText;

```
string RichText::Rich2Plain (richtext);
string richtext ;
```

Parameters

string *richtext*	the text to be converted

Return Value

string	the converted text

Wizard Sequencer

Name

Sequencer::Run -- The Wizard Sequencer

Sequencer::Run

Import Sequencer;

```
symbol Sequencer::Run (aliases, sequence);
map aliases ;
map sequence ;
```

Parameters

map *aliases*	the map of aliases
map *sequence*	the sequence of dialogs

Return Value

symbol	final symbol or nil, if error (see the y2log)

Name

Sequencer::WS_alias -- Find an aliases in the aliases map

Sequencer::WS_alias

Import Sequencer;

```
any Sequencer::WS_alias (aliases, alias);
map aliases ;
string alias ;
```

Parameters

map *aliases*	map of aliases
string *alias*	given alias

Return Value

any	belonging to the given alias or nil, if error

Name

Sequencer::WS_check -- Check correct types in maps and alias presence for sequence.

Sequencer::WS_check

Import Sequencer;

```
boolean Sequencer::WS_check (aliases, sequence);
map aliases ;
map sequence ;
```

Parameters

map *aliases*	the map of aliases
map *sequence*	the sequence of dialogs

Return Value

boolean	check passed?

Name

Sequencer::WS_error -- Report error and return nil

Sequencer::WS_error

Import Sequencer;

```
any Sequencer::WS_error (error);
string error ;
```

Parameters

string *error*	the error message text

Return Value

any	always nil

See

- bug #6474 [http://bugzilla.suse.de/6474]

Name

Sequencer::WS_next -- Find a next item in the sequence

Sequencer::WS_next

Import Sequencer;

```
any Sequencer::WS_next (sequence, current, ret);
map sequence ;
string current ;
symbol ret ;
```

Parameters

map *sequence*	sequence of dialogs
string *current*	current dialog
symbol *ret*	returned value (determines the next dialog)

Return Value

any	next dialog (symbol), WS action (string) or nil, if error (current or next not found)

Name

Sequencer::WS_pop -- Pop one item from the stack (remove an item and return the stack top item)

Sequencer::WS_pop

Import Sequencer;

```
list Sequencer::WS_pop (stack);
list stack ;
```

Parameters

list *stack*	stack of previously run dialogsk

Return Value

list	[new stack, poped value] or nil if the stack is empty or nil

Name

Sequencer::WS_push -- Push one item to the stack

Sequencer::WS_push

Import Sequencer;

```
list Sequencer::WS_push (stack, item);
list stack ;
any item ;
```

Parameters

list *stack*	stack of previously run dialogs
any *item*	item to be pushed

Return Value

list	the new stack or nil, if the stack is nil

Name

Sequencer::WS_run -- Run a function from the aliases map

Sequencer::WS_run

Import Sequencer;

```
symbol Sequencer::WS_run (aliases, id);
map aliases ;
string id ;
```

Parameters

map *aliases*	map of aliases
string *id*	function to run

Return Value

symbol	returned value from function or nil, if function is nil or returned something else than symbol

Name

Sequencer::WS_special -- Decide if an alias is special

Sequencer::WS_special

Import Sequencer;

```
boolean Sequencer::WS_special (aliases, alias);
map aliases ;
string alias ;
```

Parameters

map *aliases*	map of aliases
string *alias*	given alias

Return Value

boolean	true if the given alias is special or nil, if not found

Name

Sequencer::WS_testall -- Test (run) all dialogs in the aliases map

Sequencer::WS_testall

Import Sequencer;

```
list Sequencer::WS_testall (aliases);
map aliases ;
```

Parameters

map *aliases*	the map of aliases

Return Value

list	returned values of tested dialogs

See

- WS documentation for the format of aliases map

Service manipulation

Name

Service::Adjust -- Adjust runlevels in which the service runs.

Service::Adjust

Import Service;

```
boolean Service::Adjust (name, action);
string name ;
string action ;
```

Parameters

string *name*	service name
string *action*	"disable" -- remove links, "enable" -- if there are no links, set default, otherwise do nothing, "default" -- set defaults.

Return Value

boolean	success state

Name

Service::Disable -- Disable service

Service::Disable

Import Service;

```
boolean Service::Disable (service);
string service ;
```

Parameters

string *service*	service to be disabled

Return Value

boolean	true if operation is successful

Name

Service::Enable -- Enable service

Service::Enable

Import Service;

```
boolean Service::Enable (service);
string service ;
```

Parameters

string *service*	service to be enabled

Return Value

boolean	true if operation is successful

Name

Service::Enabled -- Check if service is enabled

Service::Enabled

Import Service;

```
boolean Service::Enabled (name);
string name ;
```

Parameters

string *name*	service name

Return Value

boolean	true if service is set to run in any runlevel

Description

Returns true if any link in /etc/init.d/rc?.d/ exists for this script. If service does not exist, logs an error.

Name

Service::Error -- Error Message

Service::Error

Import Service;

`string` **`Service::Error`** `();`

Return Value

string	error message from the last operation

Name

Service::Finetune -- Set service to run in selected runlevels.

Service::Finetune

Import Service;

```
boolean Service::Finetune (name, rl);
string name ;
list rl ;
```

Parameters

string *name*	name of service to adjust
list *rl*	list of runlevels in which service should start

Return Value

boolean	success state

Name

Service::FullInfo -- Get service info and find out whether service is running.

Service::FullInfo

Import Service;

```
map Service::FullInfo (name);
string name ;
```

Parameters

string *name*	name of the service

Return Value

map	service map or empty map ($[])

Name

Service::Info -- Get service info without peeking if service runs.

Service::Info

Import Service;

```
map Service::Info (name);
string name ;
```

Parameters

string *name*	name of the service

Return Value

map	Service information or empty map ($[])

Name

Service::Reload -- Reload service

Service::Reload

Import Service;

```
boolean Service::Reload (service);
string service ;
```

Parameters

string *service*	service to be reloaded

Return Value

boolean	true if operation is successful

Name

Service::Restart -- Restart service

Service::Restart

Import Service;

```
boolean Service::Restart (service);
string service ;
```

Parameters

string *service*	service to be restarted

Return Value

boolean	true if operation is successful

Name

Service::RunInitScript -- Run init script.

Service::RunInitScript

Import Service;

```
integer Service::RunInitScript (name, param);
string name ;
string param ;
```

Parameters

string *name*	init service name
string *param*	init script argument

Return Value

integer	exit value

Name

Service::RunInitScriptOutput -- Run init script and return output

Service::RunInitScriptOutput

Import Service;

```
map Service::RunInitScriptOutput (name, param);
string name ;
string param ;
```

Parameters

string *name*	init service name
string *param*	init script argument

Return Value

map	A map of $["stdout" : "...", "stderr" : "...", "exit" : int]

Name

Service::Start -- Start service

Service::Start

Import Service;

```
boolean Service::Start (service);
string service ;
```

Parameters

string *service*	service to be started

Return Value

boolean	true if operation is successful

Name

Service::Status -- Get service status.

Service::Status

Import Service;

```
integer Service::Status (name);
string name ;
```

Parameters

string *name*	name of the service

Return Value

integer	init script exit status or -1 if it does not exist

Name

Service::Stop -- Stop service

Service::Stop

Import Service;

```
boolean **Service::Stop** (service);
string service ;
```

Parameters

string *service*	service to be stopped

Return Value

boolean	true if operation is successful

String manipulation routines

Name

String::CAlnum --

String::CAlnum

Import String;

`string` **`String::CAlnum`** `();`

Return Value

string	the 52 upper and lowercase ASCII letters and digits

Name

String::CAlpha --

String::CAlpha

Import String;

```
string String::CAlpha ();
```

Return Value

string	the 52 upper and lowercase ASCII letters

Name

String::CDigit --

String::CDigit

Import String;

```
string String::CDigit ();
```

Return Value

string	0123456789

Name

String::CGraph --

String::CGraph

Import String;

```
string String::CGraph ();
```

Return Value

string	printable ASCII charcters except whitespace

Name

String::CLower --

String::CLower

Import String;

```
string String::CLower ();
```

Return Value

string	the 26 lowercase ASCII letters

Name

String::CPrint --

String::CPrint

Import String;

```
string String::CPrint ();
```

Return Value

string	printable ASCII charcters including whitespace

Name

String::CPunct --

String::CPunct

Import String;

```
string String::CPunct ();
```

Return Value

string	the ASCII printable non-blank non-alphanumeric characters

Name

String::CSpace --

String::CSpace

Import String;

```
string String::CSpace ();
```

Return Value

string	ASCII whitespace

Name

String::CUpper --

String::CUpper

Import String;

```
string String::CUpper ();
```

Return Value

string	the 26 uppercase ASCII letters

Name

String::CXdigit --

String::CXdigit

Import String;

```
string String::CXdigit ();
```

Return Value

string	hexadecimal digits: 0123456789ABCDEFabcdef

Name

String::CutBlanks -- Remove blanks at begin and end of input string.

String::CutBlanks

Import String;

```
string String::CutBlanks (input);
string input ;
```

Parameters

string *input*	string to be stripped

Return Value

string	stripped string

Examples

```
CutBlanks("  any  input     ") -> "any  input"
```

Name

String::CutRegexMatch -- Remove first or every match of given regular expression from a string

String::CutRegexMatch

Import String;

```
string String::CutRegexMatch (input, regex, glob);
string input ;
string regex ;
boolean glob ;
```

Parameters

string *input*	string that might occur regex
string *regex*	regular expression to search for, must not contain brackets
boolean *glob*	flag if only first or every occuring match should be removed

Return Value

string	that has matches removed

Description

(e.g. CutRegexMatch("abcdef12ef34gh000", "[0-9]+", true) -> "abcdefefgh", CutRegexMatch("abcdef12ef34gh000", "[0-9]+", false) -> "abcdefef34gh000")

Name

String::CutZeros -- Remove any leading zeros

String::CutZeros

Import String;

```
string String::CutZeros (input);
string input ;
```

Parameters

string *input*	number that might contain leadig zero

Return Value

string	that has leading zeros removed

Description

Remove any leading zeros that make tointeger inadvertently assume an octal number (e.g. "09" -> "9", "0001" -> "1", but "0" -> "0")

Name

String::EscapeTags -- Function for escaping (replacing) (HTML|XML...) tags with their (HTML|XML...) meaning.

String::EscapeTags

Import String;

```
string String::EscapeTags (text);
string text ;
```

Parameters

string *text*

Return Value

string	escaped text

Description

Usable to present text "as is" in RichText.

Name

String::FirstChunk -- Shorthand for select (splitstring (s, separators), 0, "") Useful now that the above produces a deprecation warning.

String::FirstChunk

Import String;

```
string String::FirstChunk (s, separators);
string s ;
string separators ;
```

Parameters

string *s*	string to be split
string *separators*	characters which delimit components

Return Value

string	first component or ""

Name

String::FormatSize -- Return a pretty description of a byte count

String::FormatSize

Import String;

```
string String::FormatSize (bytes);
integer bytes ;
```

Parameters

integer *bytes*	size (e.g. free diskspace) in Bytes

Return Value

string	formatted string

Description

Return a pretty description of a byte count, with two fraction digits and using KB, MB or GB as unit as appropriate.

Examples

```
FormatSize(23456767890) -> "223.70 MB"
```

Name

String::FormatSizeWithPrecision -- Return a pretty description of a byte count

String::FormatSizeWithPrecision

Import String;

```
string String::FormatSizeWithPrecision (bytes, precision, omit_zeroes);
integer bytes ;
integer precision ;
boolean omit_zeroes ;
```

Parameters

integer *bytes*	size (e.g. free diskspace, memory size) in Bytes
integer *precision*	number of fraction digits in output
boolean *omit_zeroes*	if true then do not add zeroes (usefull for memory size - 128 MB RAM looks better than 128.00 MB RAM)

Return Value

string	formatted string

Description

Return a pretty description of a byte count with required precision and using KB, MB or GB as unit as appropriate.

Examples

```
FormatSizeWithPrecision(4096, 2, true) -> "4 KB"
FormatSizeWithPrecision(4096, 2, false) -> "4.00 KB"
```

Name

String::Pad -- Add spaces after the text to make it long enough

String::Pad

Import String;

```
string String::Pad (text, length);
string text ;
integer length ;
```

Parameters

string *text*	text to be padded
integer *length*	requested length

Return Value

string	padded text

Description

Add spaces after the text to make it long enough. If the text is longer than requested, no changes are made.

Name

String::PadZeros -- Add zeros before the text to make it long enough.

String::PadZeros

Import String;

```
string String::PadZeros (text, length);
string text ;
integer length ;
```

Parameters

string *text*	text to be padded
integer *length*	requested length

Return Value

string	padded text

Description

Add zeros before the text to make it long enough. If the text is longer than requested, no changes are made.

Name

String::ParseOptions -- Parse string of values

String::ParseOptions

Import String;

```
list<string> String::ParseOptions (options, parameters);
string options ;
map parameters ;
```

Parameters

string *options*	Input string
map *parameters*	Parmeter used at parsing - map with keys: "separator":<string> - value separator (default: " \t"), "unique":<boolean> - result will not contain any duplicates, first occurance of the string is stored into output (default: false), "interpret_backslash":<boolean> - convert backslash sequence into one character (e.g. "\\n" => "\n") (default: true) "remove_whitespace":<boolean> - remove white spaces around values (default: true),

Return Value

list<string>	List of strings

Name

String::Quote -- Quote a string with 's

String::Quote

Import String;

```
string String::Quote (var);
string var ;
```

Parameters

string *var*	unquoted string

Return Value

string	quoted string

Examples

```
quote("a'b") -> "a'\''b"
```

Name

String::UnQuote -- Unquote a string with 's (quoted with quote)

String::UnQuote

Import String;

```
string String::UnQuote (var);
string var ;
```

Parameters

string *var*	quoted string

Return Value

string	unquoted string

See

- String::quote

Name

String::ValidCharsFilename -- Characters valid in a filename (not pathname). Naturally "/" is disallowed. Otherwise, the graphical ASCII characters are allowed.

String::ValidCharsFilename

Import String;

```
string String::ValidCharsFilename ();
```

Return Value

string	for ValidChars

Support for summaries of the configured devices

Name

Summary::AddHeader -- Add a RichText section header to an existing summary.

Summary::AddHeader

Import Summary;

```
string **Summary::AddHeader** (summary, header);
string summary ;
string header ;
```

Parameters

string *summary*	previous RichText (HTML) summary to add to
string *header*	header to add (plain text, no HTML)

Return Value

string	the new summary including the new header

Name

Summary::AddLine -- Add a line to an existing summary.

Summary::AddLine

Import Summary;

```
string Summary::AddLine (summary, line);
string summary ;
string line ;
```

Parameters

string *summary*	previous RichText (HTML) summary to add to
string *line*	line to add (plain text, no HTML)

Return Value

string	the new summary including the new line

Name

Summary::AddListItem -- Add a list item to an existing summary. Requires a previous call to 'summaryOpenList()'.

Summary::AddListItem

Import Summary;

```
string Summary::AddListItem (summary, item);
string summary ;
string item ;
```

Parameters

string *summary*	previous RichText (HTML) summary to add to
string *item*	item to add (plain text, no HTML)

Return Value

string	the new summary including the new line

Name

Summary::AddNewLine -- Add a newline to an existing summary.

Summary::AddNewLine

Import Summary;

```
string Summary::AddNewLine (summary);
string summary ;
```

Parameters

string *summary*	previous RichText (HTML) summary to add to

Return Value

string	the new summary

Name

Summary::AddSimpleSection -- Add a simple section to an existing summary, consisting of a header and one single item.

Summary::AddSimpleSection

Import Summary;

```
string Summary::AddSimpleSection (summary, header, item);
string summary ;
string header ;
string item ;
```

Parameters

string *summary*	previous RichText (HTML) summary to add to
string *header*	section header (plain text, no HTML)
string *item*	section item (plain text, no HTML)

Return Value

string	the new summary including the new line

Name

Summary::CloseList -- End a list within a summary.

Summary::CloseList

Import Summary;

```
string Summary::CloseList (summary);
string summary ;
```

Parameters

string *summary*	previous RichText (HTML) summary to add to

Return Value

string	the new summary

Name

Summary::Device -- Function that creates description of one device.

Summary::Device

Import Summary;

```
string Summary::Device (name, description);
string name ;
string description ;
```

Parameters

string *name*	The name of the device given by probing
string *description*	Additional description (how it was confgured or so)

Return Value

string	String with the item.

Name

Summary::DevicesList -- Function that creates the whole final product. "Not detected" will be returned if the list is empty.

Summary::DevicesList

Import Summary;

```
string Summary::DevicesList (devices);
list<string> devices ;
```

Parameters

list<string> *devices*	A list of output of the summaryDevice() calls

Return Value

string	The resulting text.

Name

Summary::NotConfigured -- Function that creates a 'Not configured.' message.

Summary::NotConfigured

Import Summary;

```
string Summary::NotConfigured ();
```

Return Value

string	String with the message.

Name

Summary::OpenList -- Start a list within a summary.

Summary::OpenList

Import Summary;

```
string Summary::OpenList (summary);
string summary ;
```

Parameters

string *summary*	previous RichText (HTML) summary to add to

Return Value

string	the new summary

Type repository for validation of user-defined types

Name

TypeRepository::TypeRepository -- Constructor, defines the known types.

TypeRepository::TypeRepository

Import TypeRepository;

`void` **`TypeRepository::TypeRepository`** `();`

Return Value

void

Name

TypeRepository::enum_validator -- Generic enumerated type validator.

TypeRepository::enum_validator

Import TypeRepository;

```
boolean TypeRepository::enum_validator (values, value);
list values ;
string value ;
```

Parameters

list *values*	a list of possible values
string *value*	the value to be matched

Return Value

boolean	true if successful

Name

TypeRepository::is_a -- Validate, that the given value is of given type.

TypeRepository::is_a

Import TypeRepository;

```
boolean TypeRepository::is_a (value, type);
any value ;
string type ;
```

Parameters

any *value*	value to be validated
string *type*	type against which to validate

Return Value

boolean	true, if the value can be considered to be of a given type

Name

TypeRepository::regex_validator -- Generic regular expression validator.

TypeRepository::regex_validator

Import TypeRepository;

```
boolean TypeRepository::regex_validator (regex, value);
string regex ;
string value ;
```

Parameters

string *regex*	the regular expression to be matched
string *value*	the value to be matched

Return Value

boolean	true if successful

Manipulate and Parse URLs

Name

URL::Build -- Build URL from tokens as parsed with Parse

URL::Build

Import URL;

```
string URL::Build (tokens);
map tokens ;
```

Parameters

map *tokens*

Return Value

string	url, empty string if invalid data is used to build the url.

See

- RFC 2396 (updated by RFC 2732)
- also perl-URI: URI(3)

Name

URL::Check -- Check URL

URL::Check

Import URL;

```
boolean URL::Check (url);
string url ;
```

Parameters

string *url*	URL to be checked

Return Value

boolean	true if correct

See

- RFC 2396 (updated by RFC 2732)
- also perl-URI: URI(3)

Name

URL::Parse -- Tokenize URL

URL::Parse

Import URL;

```
map URL::Parse (url);
string url ;
```

Parameters

string *url*	URL to be parsed

Return Value

map	URL split to tokens

Examples

```
Parse("http://name:pass@www.suse.cz:80/path/index.html?question#part") ->
   $[
       "scheme"  : "http",
       "host"    : "www.suse.cz"
       "port"    : "80",
       "path"    : /path/index.html",
       "user"    : "name",
       "pass"    : "pass",
       "query"   : "question",
       "fragment": "part"
   ]
```

Wizard dialogs for hardware configuration

Name

Wizard_hw::ConfiguredContent -- Create the contents of screen with configured items.

Wizard_hw::ConfiguredContent

Import Wizard_hw;

```
term Wizard_hw::ConfiguredContent (table_header, table_contents,
above_table, below_table, below_buttons, buttons);
term table_header ;
list table_contents ;
term above_table ;
term below_table ;
term below_buttons ;
term buttons ;
```

Parameters

term *table_header*	Table header as defined in UI.
list *table_contents*	Table items.
term *above_table*	Content to place above table. There is no need to place caption here, because the dialog has its caption. Set it to nil if you do not want to place anything here.
term *below_table*	Contents to place between table and buttons. Set it to nil if you do not want to place anything here.
term *below_buttons*	Content to place below bottons. Set it to nil if you do not want to place anything here.
term *buttons*	Content to place rights from buttons. Usually an additional button, e.g. Set as default. Set it to nil if you do not want to place anything here.

Return Value

term	Content for the SetWizardContent[Buttons]() UI elements ids:<table> <tr><td>Table</td><td>`table</td></tr> <tr><td>Button add</td><td>`add_button</td></tr> <tr><td>Button edit</td><td>`edit_button</td></tr> <tr><td>Button delete</td><td>`delete_button</td></tr> </table>

Description

It contains table and buttons Add, Edit, Delete. User may specify table header and content, content that will be placed above table, between table and buttons, below buttons and rights from buttons (usually another button).

Name

Wizard_hw::DetectedContent -- Create the content of the screen with the detected devices.

Wizard_hw::DetectedContent

Import Wizard_hw;

```
term Wizard_hw::DetectedContent (frame_label, detected,
has_restart, already_conf);
string frame_label ;
list detected ;
boolean has_restart ;
string already_conf ;
```

Parameters

string *frame_label*	The frame around the detected devices
list *detected*	A list of the detected devices for the SelectionBox. Must contain "Other (not detected)" as the last item.
boolean *has_restart*	A button for restarting the detection.
string *already_conf*	A content of the RichText. Describes the already configured devices.

Return Value

term	Content for the SetWizardContent[Buttons]()

Description

It is used in the SetWizardContent() or SetWizardContentButtons() functions.

Name

Wizard_hw::SizeAtLeast -- Encloses the content into VBoxes and HBoxes

Wizard_hw::SizeAtLeast

Import Wizard_hw;

```
term Wizard_hw::SizeAtLeast (content, xsize, ysize);
term content ;
float xsize ;
float ysize ;
```

Parameters

term *content*	Content of the dialog
float *xsize*	Minimal size of content in the X direction
float *ysize*	Minimal size of content in the Y direction

Return Value

term	Contents sized at least xsize&nbsp;x&nbsp;ysize.

Name

Wizard_hw::SpacingAround -- Encloses the content into VBoxes and HBoxes with the appropriate spacings around it.

Wizard_hw::SpacingAround

Import Wizard_hw;

```
term Wizard_hw::SpacingAround (content, left, right, top, bottom);
term content ;
float left ;
float right ;
float top ;
float bottom ;
```

Parameters

term *content*	The term we are adding spacing to.
float *left*	Spacing on the left.
float *right*	Spacing on the right.
float *top*	Spacing on the top.
float *bottom*	Spacing on the bottom.

Return Value

term	Content with spacings around it.

Wizard dialog

Name

Wizard::AbortAcceptButtonBox -- Returns a button box with buttons "Abort", "Accept"

Wizard::AbortAcceptButtonBox

Import Wizard;

```
term Wizard::AbortAcceptButtonBox ();
```

Return Value

term	a widget tree

Name

Wizard::AbortApplyFinishButtonBox -- Returns a button box with buttons "Abort", "Apply", "Finish"

Wizard::AbortApplyFinishButtonBox

Import Wizard;

`term` **`Wizard::AbortApplyFinishButtonBox`** `();`

Return Value

term	a widget tree

Name

Wizard::AbortInstallationAcceptButtonBox -- Returns a button box with buttons "Abort Installation", "Accept"

Wizard::AbortInstallationAcceptButtonBox

Import Wizard;

`term` **`Wizard::AbortInstallationAcceptButtonBox`** `();`

Return Value

term	a widget tree

Name

Wizard::AcceptDialog -- Returns a standard wizard dialog with buttons "Cancel", "Accept"

Wizard::AcceptDialog

Import Wizard;

```
term Wizard::AcceptDialog ();
```

Return Value

term	describing the dialog.

Name

Wizard::AddMenu -- Add Menu

Wizard::AddMenu

Import Wizard;

```
list<map> Wizard::AddMenu (Menu, title, id);
list<map> Menu ;
string title ;
string id ;
```

Parameters

list<map> *Menu*	Menu data
string *title*	Menu Title
string *id*	Menu ID

Return Value

list<map>	Updated Menu Data

Name

Wizard::AddMenuEntry -- Add Menu Entry

Wizard::AddMenuEntry

Import Wizard;

```
list<map> Wizard::AddMenuEntry (Menu, parent_id, title, id);
list<map> Menu ;
string parent_id ;
string title ;
string id ;
```

Parameters

list<map> *Menu*	Menu data
string *parent_id*	
string *title*	Menu Title
string *id*	Menu ID

Return Value

list<map>	Updated Menu Data

Name

Wizard::AddSubMenu -- Add Sub Menu

Wizard::AddSubMenu

Import Wizard;

```
list<map> Wizard::AddSubMenu (Menu, parent_id, title, id);
list<map> Menu ;
string parent_id ;
string title ;
string id ;
```

Parameters

list<map> *Menu*	Menu data
string *parent_id*	
string *title*	Menu Title
string *id*	Menu ID

Return Value

list<map>	Updated Menu Data

Name

Wizard::AddTreeItem -- Add Tree Item to tree enabled Wizard

Wizard::AddTreeItem

Import Wizard;

```
list<map> Wizard::AddTreeItem (Tree, parent, title, id);
list<map> Tree ;
string parent ;
string title ;
string id ;
```

Parameters

list<map> *Tree*	Tree Data
string *parent*	Parent of this item
string *title*	Item Title
string *id*	Item ID

Return Value

list<map>	Updated Tree Data

Name

Wizard::BackAbortInstallationNextButtonBox -- Returns a button box with buttons "Back", "Abort Installation", "Next"

Wizard::BackAbortInstallationNextButtonBox

Import Wizard;

```
term Wizard::BackAbortInstallationNextButtonBox ();
```

Return Value

term	a widget tree

Name

Wizard::BackAbortNextButtonBox -- Returns a button box with buttons "Back", "Abort", "Next"

Wizard::BackAbortNextButtonBox

Import Wizard;

`term` **`Wizard::BackAbortNextButtonBox`** `();`

Return Value

term	a widget tree

Name

Wizard::BackNextButtonBox -- Returns a button box with buttons "Back", "Next"

Wizard::BackNextButtonBox

Import Wizard;

`term` **`Wizard::BackNextButtonBox`** `();`

Return Value

term	a widget tree

Name

Wizard::CancelAcceptButtonBox -- Returns a button box with buttons "Cancel", "Accept"

Wizard::CancelAcceptButtonBox

Import Wizard;

```
term Wizard::CancelAcceptButtonBox ();
```

Return Value

term	a widget tree

Name

Wizard::ClearContents -- Clear the wizard contents.

Wizard::ClearContents

Import Wizard;

```
void Wizard::ClearContents ();
```

Return Value

void

Name

Wizard::ClearTitleIcon -- Clear the wizard 'title' icon, i.e. replace it with nothing

Wizard::ClearTitleIcon

Import Wizard;

```
void Wizard::ClearTitleIcon ();
```

Return Value

void

Name

Wizard::CloseDialog -- Close a wizard dialog.

Wizard::CloseDialog

Import Wizard;

```
void Wizard::CloseDialog ();
```

Return Value

void

Name

Wizard::CreateDialog -- Create and open a typical installation wizard dialog.

Wizard::CreateDialog

Import Wizard;

`void` **`Wizard::CreateDialog`** `();`

Return Value

void

Name

Wizard::CreateMenu -- Create the menu in the dialog

Wizard::CreateMenu

Import Wizard;

```
void **Wizard::CreateMenu** (Menu);
list<map> Menu ;
```

Parameters

list<map> *Menu*	Menu data

Return Value

void

Name

Wizard::CreateTree -- Create the tree in the dialog, replaces helpspace with new tree widget

Wizard::CreateTree

Import Wizard;

```
void Wizard::CreateTree (Tree, title);
list<map> Tree ;
string title ;
```

Parameters

list<map> *Tree*	Tree data
string *title*	Tree title

Return Value

void

Name

Wizard::CreateTreeDialog -- Create and open a Tree wizard dialog.

Wizard::CreateTreeDialog

Import Wizard;

`void` **`Wizard::CreateTreeDialog`** `();`

Return Value

void

Name

Wizard::DeleteMenus -- Delete Menu items

Wizard::DeleteMenus

Import Wizard;

`void` **`Wizard::DeleteMenus`** `();`

Return Value

void

Name

Wizard::DeleteTreeItems -- Delete Tree items

Wizard::DeleteTreeItems

Import Wizard;

```
void Wizard::DeleteTreeItems ();
```

Return Value

void

Name

Wizard::DisableAbortButton -- Disable the wizard's "Abort" button.

Wizard::DisableAbortButton

Import Wizard;

`void` **`Wizard::DisableAbortButton`** `();`

Return Value

void

Name

Wizard::DisableBackButton -- Disable the wizard's "Back" button.

Wizard::DisableBackButton

Import Wizard;

```
void Wizard::DisableBackButton ();
```

Return Value

void

Name

Wizard::DisableCancelButton -- Disable the wizard's "Cancel" button.

Wizard::DisableCancelButton

Import Wizard;

```
void Wizard::DisableCancelButton ();
```

Return Value

void

Name

Wizard::DisableNextButton -- Disable the wizard's "Next" (or "Accept") button.

Wizard::DisableNextButton

Import Wizard;

`void` **`Wizard::DisableNextButton`** `();`

Return Value

void

Name

Wizard::EnableAbortButton -- Enable the wizard's "Abort" button.

Wizard::EnableAbortButton

Import Wizard;

```
void Wizard::EnableAbortButton ();
```

Return Value

void

Name

Wizard::EnableBackButton -- Enable the wizard's "Back" button.

Wizard::EnableBackButton

Import Wizard;

```
void Wizard::EnableBackButton ();
```

Return Value

void

Name

Wizard::EnableCancelButton -- Enable the wizard's "Cancel" button.

Wizard::EnableCancelButton

Import Wizard;

`void` **`Wizard::EnableCancelButton`** `();`

Return Value

void

Name

Wizard::EnableNextButton -- Enable the wizard's "Next" (or "Accept") button.

Wizard::EnableNextButton

Import Wizard;

```
void Wizard::EnableNextButton ();
```

Return Value

void

Name

Wizard::GenericDialog -- Create a Generic Dialog

Wizard::GenericDialog

Import Wizard;

```
term Wizard::GenericDialog (button_box);
term button_box ;
```

Parameters

term *button_box*	term that contains a `HBox() with buttons in it

Return Value

term	term describing the dialog.

Description

Returns a term describing a generic wizard dialog with a configurable button box.

Name

Wizard::HideAbortButton -- Hide the Wizard's "Abort" button. Restore it later with RestoreAbort-Button():

Wizard::HideAbortButton

Import Wizard;

```
void Wizard::HideAbortButton ();
```

Return Value

void

Name

Wizard::HideBackButton -- Hide the Wizard's "Back" button. Restore it later with RestoreBackButton():

Wizard::HideBackButton

Import Wizard;

```
void Wizard::HideBackButton ();
```

Return Value

void

Name

Wizard::HideNextButton -- Hide the Wizard's "Next" button. Restore it later with RestoreNextButton():

Wizard::HideNextButton

Import Wizard;

```
void Wizard::HideNextButton ();
```

Return Value

void

Name

Wizard::IsWizardDialog -- Check if the topmost dialog is a wizard dialog (i.e. has a widget with `id(`WizardDialog))

Wizard::IsWizardDialog

Import Wizard;

```
boolean Wizard::IsWizardDialog ();
```

Return Value

boolean	True if topmost dialog is a wizard dialog, false otherwise

Name

Wizard::NextBackDialog -- Returns a standard wizard dialog with buttons "Next", "Back", "Abort".

Wizard::NextBackDialog

Import Wizard;

```
term Wizard::NextBackDialog ();
```

Return Value

term	describing the dialog.

Name

Wizard::OpenAbortApplyFinishDialog -- Open a dialog with "Accept", "Cancel" and set the keyboard focus to "Accept".

Wizard::OpenAbortApplyFinishDialog

Import Wizard;

`void` **`Wizard::OpenAbortApplyFinishDialog`** `();`

Return Value

void

Name

Wizard::OpenAcceptAbortStepsDialog -- Open a dialog with "Accept", "Cancel" that will also accept workflow steps.

Wizard::OpenAcceptAbortStepsDialog

Import Wizard;

```
void Wizard::OpenAcceptAbortStepsDialog ();
```

Return Value

void

Name

Wizard::OpenAcceptDialog -- Open a dialog with "Accept", "Cancel" and set the keyboard focus to "Accept".

Wizard::OpenAcceptDialog

Import Wizard;

```
void Wizard::OpenAcceptDialog ();
```

Return Value

void

Name

Wizard::OpenAcceptStepsDialog -- Open a dialog with "Accept", "Cancel" that will also accept workflow steps.

Wizard::OpenAcceptStepsDialog

Import Wizard;

`void` **`Wizard::OpenAcceptStepsDialog`** `();`

Return Value

void

Name

Wizard::OpenCustomDialog -- Open a wizard dialog with simple layout

Wizard::OpenCustomDialog

Import Wizard;

```
void Wizard::OpenCustomDialog (help_space_contents, button_box);
term help_space_contents ;
term button_box ;
```

Parameters

term *help_space_contents*	Help space contents
term *button_box*	Buttom Box

Return Value

void

Description

no graphics, no steps, only a help widget buttons (by default "Back", "Abort", "Next").

This is the only type of wizard dialog which still allows replacing the help space - either already upon opening it or later with Wizard::ReplaceCustomHelp().

Name

Wizard::OpenDialog -- Open any wizard dialog.

Wizard::OpenDialog

Import Wizard;

```
void Wizard::OpenDialog (dialog);
term dialog ;
```

Parameters

term *dialog*	a wizard dialog, e.g. Wizard::GenericDialog()

Return Value

void

Name

Wizard::OpenNextBackDialog -- Open a dialog with buttons "Next", "Back", "Abort" and set the keyboard focus to "Next".

Wizard::OpenNextBackDialog

Import Wizard;

```
void **Wizard::OpenNextBackDialog** ();
```

Return Value

void

Name

Wizard::OpenNextBackStepsDialog -- Open a dialog with "Back", "Next", "Abort" that will also accept workflow steps.

Wizard::OpenNextBackStepsDialog

Import Wizard;

```
void Wizard::OpenNextBackStepsDialog ();
```

Return Value

void

Name

Wizard::OpenTreeNextBackDialog -- Open a Tree dialog with buttons "Next", "Back", "Abort" and set the keyboard focus to "Next".

Wizard::OpenTreeNextBackDialog

Import Wizard;

```
void Wizard::OpenTreeNextBackDialog ();
```

Return Value

void

Name

Wizard::QueryTreeItem -- Query Tree Item

Wizard::QueryTreeItem

Import Wizard;

```
string Wizard::QueryTreeItem ();
```

Return Value

string	Tree Item

Name

Wizard::ReplaceAbortButton -- Replace the wizard 'abort' button with a custom widget. THIS FUNCTION IS DEPRECATED!

Wizard::ReplaceAbortButton

Import Wizard;

```
void Wizard::ReplaceAbortButton (contents);
term contents ;
```

Parameters

term *contents*	a term describing the new contents

Return Value

void

Name

Wizard::ReplaceBackButton -- Replace the wizard 'back' button with a custom widget. THIS FUNCTION IS DEPRECATED!

Wizard::ReplaceBackButton

Import Wizard;

```
void Wizard::ReplaceBackButton (contents);
term contents ;
```

Parameters

term *contents*	a term describing the new contents

Return Value

void

Name

Wizard::ReplaceCustomHelp -- Replace the help widget for dialogs opened with Wizard::OpenCustomDialog().

Wizard::ReplaceCustomHelp

Import Wizard;

```
void Wizard::ReplaceCustomHelp (contents);
term contents ;
```

Parameters

term *contents*	Replace custom help with supplied contents

Return Value

void

Name

Wizard::ReplaceHelp -- Replace the wizard help subwindow with a custom widget.

Wizard::ReplaceHelp

Import Wizard;

```
void Wizard::ReplaceHelp (contents);
term contents ;
```

Parameters

term *contents*	Replace Help with contents

Return Value

void

Deprecated

This function is deprecated.

Name

Wizard::ReplaceNextButton -- Replace the wizard 'next' button with a custom widget. THIS FUNCTION IS DEPRECATED!

Wizard::ReplaceNextButton

Import Wizard;

```
void Wizard::ReplaceNextButton (contents);
term contents ;
```

Parameters

term *contents*	a term describing the new contents

Return Value

void

Name

Wizard::RestoreAbortButton -- Restore the wizard 'abort' button.

Wizard::RestoreAbortButton

Import Wizard;

```
void Wizard::RestoreAbortButton ();
```

Return Value

void

Name

Wizard::RestoreBackButton -- Restore the wizard 'back' button.

Wizard::RestoreBackButton

Import Wizard;

`void` **`Wizard::RestoreBackButton`** `();`

Return Value

void

Name

Wizard::RestoreHelp -- Restore the wizard help subwindow.

Wizard::RestoreHelp

Import Wizard;

```
void Wizard::RestoreHelp (help_text);
string help_text ;
```

Parameters

string *help_text*	Help text

Return Value

void

Name

Wizard::RestoreNextButton -- Restore the wizard 'next' button.

Wizard::RestoreNextButton

Import Wizard;

`void` **`Wizard::RestoreNextButton`** `();`

Return Value

void

Name

Wizard::RestoreScreenShotName -- Restore the screenshot name.

Wizard::RestoreScreenShotName

Import Wizard;

`void` **`Wizard::RestoreScreenShotName`** `();`

Return Value

void

Name

Wizard::RetranslateButtons -- Retranslate the wizard buttons.

Wizard::RetranslateButtons

Import Wizard;

```
void Wizard::RetranslateButtons ();
```

Return Value

void

Name

Wizard::SelectTreeItem -- Select Tree item

Wizard::SelectTreeItem

Import Wizard;

```
void Wizard::SelectTreeItem (tree_item);
string tree_item ;
```

Parameters

string *tree_item*	tree item

Return Value

void

Name

Wizard::SetAbortButton -- Set the dialog's "Abort" button with a new label and a new ID

Wizard::SetAbortButton

Import Wizard;

```
void **Wizard::SetAbortButton** (id, label);
any id ;
string label ;
```

Parameters

any *id*	Button ID
string *label*	Button Label

Return Value

void

Name

Wizard::SetBackButton -- Set the dialog's "Back" button with a new label and a new ID

Wizard::SetBackButton

Import Wizard;

```
void Wizard::SetBackButton (id, label);
any id ;
string label ;
```

Parameters

any *id*	Button ID
string *label*	Button Label

Return Value

void

Name

Wizard::SetContents -- Set the contents of a wizard dialog

Wizard::SetContents

Import Wizard;

```
void Wizard::SetContents (title, contents, help_text, has_back, has_next);
string title ;
term contents ;
string help_text ;
boolean has_back ;
boolean has_next ;
```

Parameters

string *title*	Dialog Title
term *contents*	The Dialog contents
string *help_text*	Help text
boolean *has_back*	Is the Back button enabled?
boolean *has_next*	Is the Next button enabled?

Return Value

void

Description

How the general framework for the installation wizard should look like. This function creates and shows a dialog.

Examples

```
         /*
 * Trivial wizard dialog example
 *
 * Author: Stefan Hundhammer <sh@suse.de>
 *
 * $Id: wizard1.ycp,v 1.2 2004/09/15 19:11:14 nashif Exp $
 */
{
    import "Wizard";


    Wizard::CreateDialog();

    term   contents  = `Label( "Wizard contents" );
    string headline  = "Trivial Wizard Example";
    string help_text = "<p>Help text</p>";

    Wizard::SetContents( headline, contents, help_text,
                         false,         // have back button
                         true ); // have next button

    Wizard::UserInput();
    UI::CloseDialog();
}
```

Screenshot

Name

Wizard::SetContentsButtons -- Set contents and Buttons of wizard dialog

Wizard::SetContentsButtons

Import Wizard;

```
void Wizard::SetContentsButtons (title, contents, help_text,
back_label, next_label);
string title ;
term contents ;
string help_text ;
string back_label ;
string next_label ;
```

Parameters

string *title*	title of window
term *contents*	contents of dialog
string *help_text*	help text
string *back_label*	label of back button
string *next_label*	label of next button

Return Value

void

Name

Wizard::SetDesktopIcon -- Set Desktop Icon

Wizard::SetDesktopIcon

Import Wizard;

```
boolean Wizard::SetDesktopIcon (file);
string file ;
```

Parameters

string *file*

Return Value

boolean	true on success

Name

Wizard::SetFocusToBackButton -- Set the keyboard focus to the wizard's "Back" (or "Cancel") button.

Wizard::SetFocusToBackButton

Import Wizard;

```
void Wizard::SetFocusToBackButton ();
```

Return Value

void

Name

Wizard::SetFocusToNextButton -- Set the keyboard focus to the wizard's "Next" (or "Accept") button.

Wizard::SetFocusToNextButton

Import Wizard;

void **Wizard::SetFocusToNextButton** ();

Return Value

void

Name

Wizard::SetHelpText -- Set a new help text.

Wizard::SetHelpText

Import Wizard;

```
void Wizard::SetHelpText (help_text);
string help_text ;
```

Parameters

string *help_text*	Help text

Return Value

void

Examples

```
<pre>  Wizard::SetHelpText("This is a help Text");</pre>
```

Name

Wizard::SetNextButton -- Set the dialog's "Next" button with a new label and a new ID

Wizard::SetNextButton

Import Wizard;

```
void Wizard::SetNextButton (id, label);
any id ;
string label ;
```

Parameters

any *id*	Button ID
string *label*	Button Label

Return Value

void

Name

Wizard::SetProductName -- Set the product name for UI

Wizard::SetProductName

Import Wizard;

```
void Wizard::SetProductName (name);
string name ;
```

Parameters

string *name*	the product name

Return Value

void

Name

Wizard::SetScreenShotName -- Declare a name for the current dialog

Wizard::SetScreenShotName

Import Wizard;

```
void Wizard::SetScreenShotName (s);
string s ;
```

Parameters

string *s*	eg. "mail-1-conntype"

Return Value

void

See

- Wizard::RestoreScreenShotName

Name

Wizard::SetTitleIcon -- Set the wizard 'title' icon

Wizard::SetTitleIcon

Import Wizard;

```
void Wizard::SetTitleIcon (icon_name);
string icon_name ;
```

Parameters

string *icon_name*	name (without path) of the new icon

Return Value

void

Description

Set the wizard 'title' icon to the specified icon from the standard icon directory.

Name

Wizard::ShowHelp -- Open a popup dialog that displays a help text (rich text format).

Wizard::ShowHelp

Import Wizard;

```
void Wizard::ShowHelp (help_text);
string help_text ;
```

Parameters

string *help_text*	the text to display

Return Value

void

Name

Wizard::UserInput -- Substitute for UI::UserInput

Wizard::UserInput

Import Wizard;

`any` **`Wizard::UserInput`** `();`

Return Value

any	(maybe normalized) widget ID

Description

This function transparently handles different variations of the wizard layout. Returns `next if `next or `accept were clicked, `back if `back or `cancel were clicked. Simply replace ret = UI::UserInput() with ret = Wizard::UserInput()

Commonly used popup dialogs

Name

Popup::AnyMessage -- Generic message popup

Popup::AnyMessage

Import Popup;

```
void Popup::AnyMessage (headline, message);
string headline ;
string message ;
```

Parameters

string *headline*	optional headline or Popup::NoHeadline()
string *message*	the message (maybe multi-line) to display.

Return Value

void

Description

Show a message with optional headline above and wait until user clicked "OK".

See

- Popup::Message
- Popup::Notify
- Popup::Warning
- Popup::Error

Name

Popup::AnyQuestion -- Generic question popup with two buttons.

Popup::AnyQuestion

Import Popup;

```
boolean Popup::AnyQuestion (headline, message, yes_button_message,
no_button_message, focus);
string headline ;
string message ;
string yes_button_message ;
string no_button_message ;
symbol focus ;
```

Parameters

string *headline*	headline or Popup::NoHeadline()
string *message*	message string
string *yes_button_message*	label on affirmative buttons (on left side)
string *no_button_message*	label on negating button (on right side)
symbol *focus*	`focus_yes (first button) or `focus_no (second button)

Return Value

boolean	true: first button has been clicked false: second button has been clicked

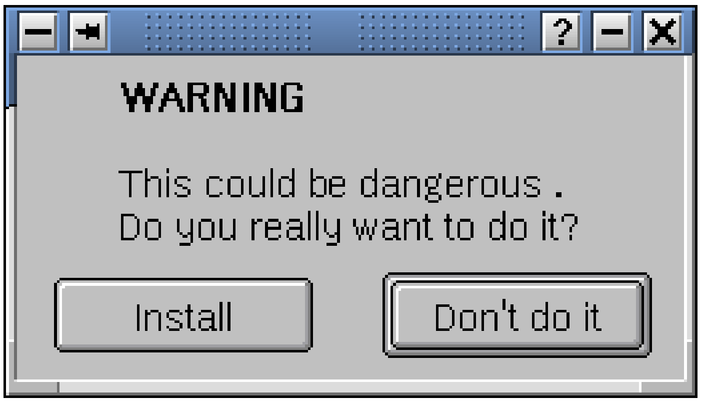
Description

Style guide hint: The first button has to have the semantics of "yes", "OK", "continue" etc., the second its opposite ("no", "cancel", ...). NEVER use this generic question popup to simply exchange the order of yes/no, continue/cancel or ok/cancel buttons!

Examples

```
Popup::AnyQuestion( Label::WarningMsg(), "Do really want to ...?", "Install", "Don't do it", `focus_no );
```

Screenshot


WARNING

This could be dangerous .
Do you really want to do it?

See

- Popup::YesNo
- Popup::ContinueCancel

Name

Popup::AnyQuestion3 -- Generic question popup with three buttons.

Popup::AnyQuestion3

Import Popup;

```
symbol Popup::AnyQuestion3 (headline, message, yes_button_message,
no_button_message, retry_button_message, focus);
string headline ;
string message ;
string yes_button_message ;
string no_button_message ;
string retry_button_message ;
symbol focus ;
```

Parameters

string *headline*	headline or Popup::NoHeadline()
string *message*	message string
string *yes_button_message*	label on affirmative button (on left side)
string *no_button_message*	label on negating button (middle)
string *retry_button_message*	label on retry button (on right side)
symbol *focus*	`focus_yes (first button), `focus_no (second button) or `focus_retry (third button)

Return Value

symbol	- `yes: first button has been clicked - `no: second button has been clicked - `retry: third button has been clicked

Examples

```
Popup::AnyQuestion3( Label::WarningMsg(), _("... failed"), _("Continue"), _("Cancel"), _("Retry"), `focus_y
```

See

- Popup::AnyQuestion

Name

Popup::AnyQuestionRichText -- Show a question that might need scrolling.

Popup::AnyQuestionRichText

Import Popup;

```
boolean Popup::AnyQuestionRichText (headline, richtext, hdim, vdim,
yes_button_message, no_button_message, focus);
string headline ;
string richtext ;
integer hdim ;
integer vdim ;
string yes_button_message ;
string no_button_message ;
symbol focus ;
```

Parameters

string *headline*	short headline
string *richtext*	text input as a rich text
integer *hdim*	initial horizontal dimension of the popup
integer *vdim*	initial vertical dimension of the popup
string *yes_button_message*	message on the left/true button
string *no_button_message*	message on the right/false button
symbol *focus*	`focus_yes, `focus_no, `focus_none

Return Value

boolean	left button pressed?

Name

Popup::AnyTimedMessage -- Generic message popup

Popup::AnyTimedMessage

Import Popup;

```
void Popup::AnyTimedMessage (headline, message, timeout);
string headline ;
string message ;
integer timeout ;
```

Parameters

string *headline*	optional headline or Popup::NoHeadline()
string *message*	the message (maybe multi-line) to display.
integer *timeout*	After timeout seconds dialog will be automatically closed

Return Value

void

Description

Show a message with optional headline above and wait until user clicked "OK".

Name

Popup::ClearFeedback -- Clear feedback message

Popup::ClearFeedback

Import Popup;

```
void Popup::ClearFeedback ();
```

Return Value

void

Name

Popup::ConfirmAbort -- Confirmation for "Abort" button during installation.

Popup::ConfirmAbort

Import Popup;

```
boolean Popup::ConfirmAbort (severity);
symbol severity ;
```

Parameters

symbol *severity*	\`painless, \`incomplete, \`unusable

Return Value

boolean

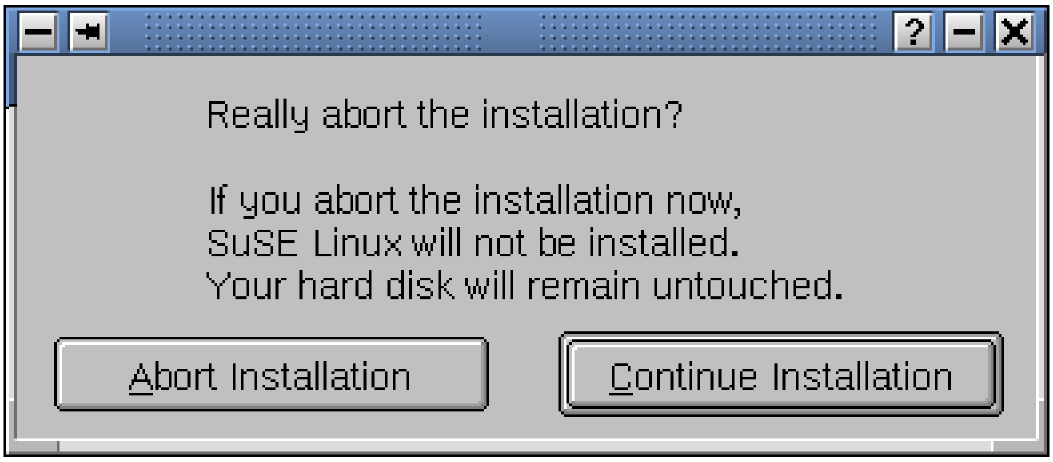
Description

According to the "severity" parameter an appropriate text will be displayed indicating what the user has to expect when he really aborts now.

Examples

```
Popup::ConfirmAbort ( `painless );
```

Screenshot


Really abort the installation?

If you abort the installation now,
SuSE Linux will not be installed.
Your hard disk will remain untouched.

Name

Popup::ContinueCancel -- Dialog which displays the "message" and has a <b>Continue</b> and a <b>Cancel</b> button.

Popup::ContinueCancel

Import Popup;

```
boolean Popup::ContinueCancel (message);
string message ;
```

Parameters

string *message*	message string

Return Value

boolean

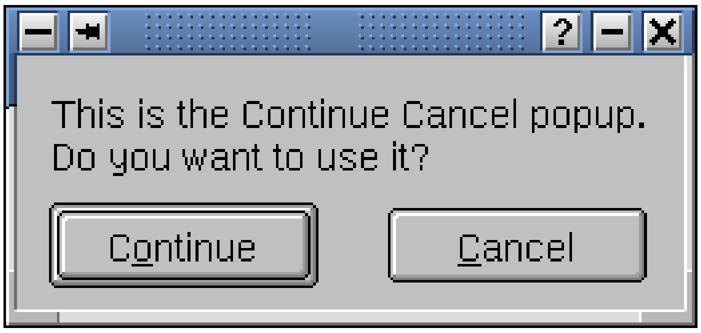
Description

This popup should be used to confirm possibly dangerous actions. The default button is Continue. Returns true if Continue is clicked.

Examples

```
Popup::ContinueCancel ( "Please insert required CD-ROM." );
```

Screenshot



This is the Continue Cancel popup.
Do you want to use it?

See

- Popup::AnyQuestion

Name

Popup::ContinueCancelHeadline -- Dialog which displays the "message" and has a <b>Continue</b> and a <b>Cancel</b> button.

Popup::ContinueCancelHeadline

Import Popup;

```
boolean Popup::ContinueCancelHeadline (headline, message);
string headline ;
string message ;
```

Parameters

string *headline*	short headline or Popup::NoHeadline()
string *message*	message string

Return Value

boolean

Description

This popup should be used to confirm possibly dangerous actions and if it's useful to display a short headline (without headline Popup::ContinueCancel() can be used). The default button is Continue.

Returns true if Continue is clicked.

Examples

```
Popup::ContinueCancelHeadline ( "Short Header", "Going on with action....?" );
```

Screenshot

See

- Popup::ContinueCancel
- Popup::YesNo
- Popup::AnyQuestion

Name

Popup::Error -- Show an error message and wait until user clicked "OK".

Popup::Error

Import Popup;

```
void Popup::Error (message);
string message ;
```

Parameters

string *message*	error message string

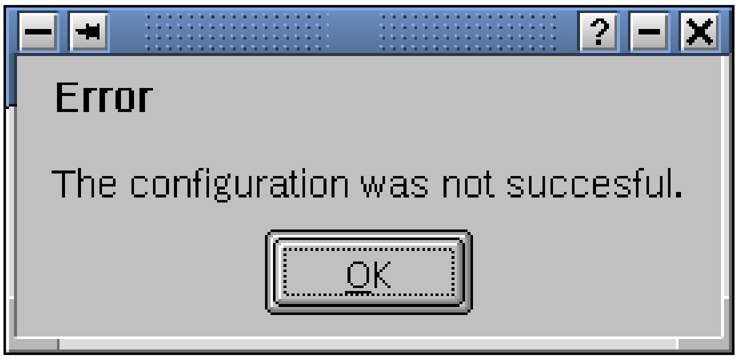
Return Value

void

Examples

```
Popup::Error("The configuration was not succesful." );
```

Screenshot



Error

The configuration was not succesful.

See

- Popup::Message
- Popup::Notify
- Popup::Warning
- Popup::AnyMessage

Name

Popup::LongText -- Show a long text that might need scrolling.

Popup::LongText

Import Popup;

```
void Popup::LongText (headline, richtext, hdim, vdim);
string headline ;
term richtext ;
integer hdim ;
integer vdim ;
```

Parameters

string *headline*	short headline
term *richtext*	text input is `Richtext()
integer *hdim*	initial horizontal dimension of the popup
integer *vdim*	initial vertical dimension of the popup

Return Value

void

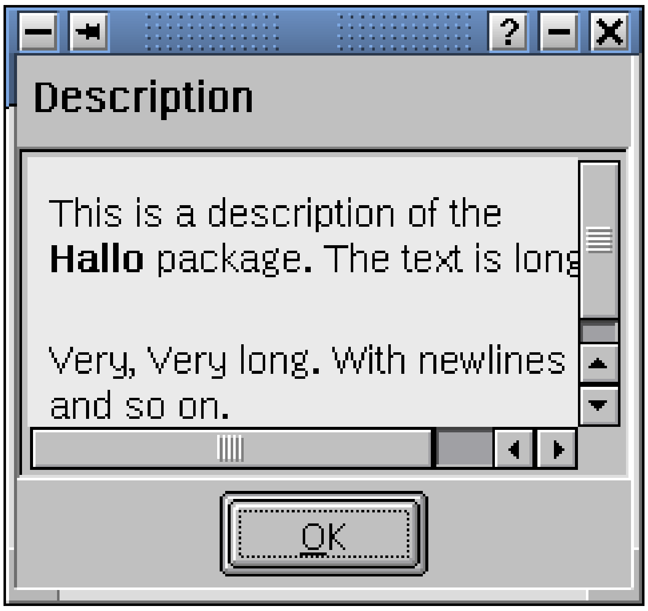
Description

Pass a RichText widget with the parameters appropriate for your text - i.e. without special parameters for HTML-like text or with `opt(`plainText) for plain ASCII text without HTML tags.

Examples

```
Popup::LongText ( "Package description", `Richtext("<p>Hello, this is a long description .....</p>"), 50, 2
```

Screenshot



Description

This is a description of the
Hallo package. The text is long

Very, Very long. With newlines
and so on.

Name

Popup::Message -- Show a simple message and wait until user clicked "OK".

Popup::Message

Import Popup;

```
void **Popup::Message** (message);
string message ;
```

Parameters

string *message*	message string

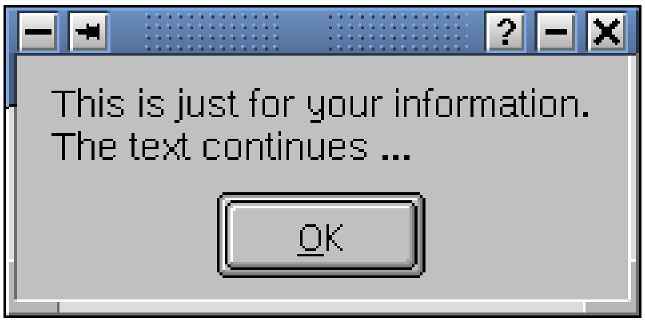
Return Value

void

Examples

```
  Popup::Message("This is an information about ... ." );
```

Screenshot


This is just for your information.
The text continues ...

See

- Popup::AnyMessage
- Popup::Notify
- Popup::Warning
- Popup::Error

Name

Popup::ModuleError -- Special error popup for YCP modules that don't work.

Popup::ModuleError

Import Popup;

```
symbol Popup::ModuleError (text);
string text ;
```

Parameters

string *text*	string

Return Value

symbol	\`back, \`again, \`cancel, \`next

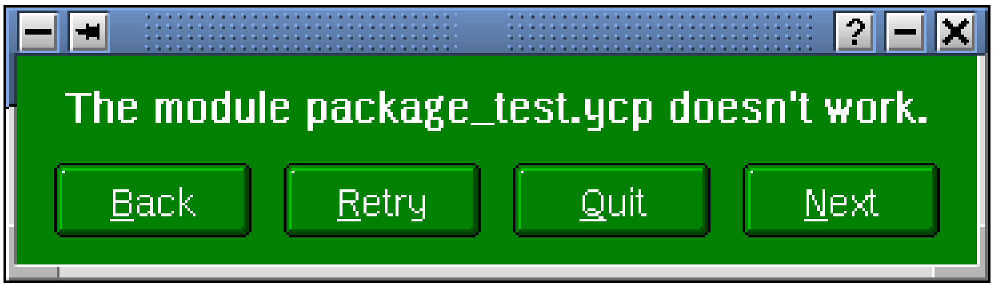
Description

The user can choose one of: - "back" - go back to the previous module - "next" - skip this faulty module and directly go to the next one - "again" - try it again (after fixing something in the code, of course) - "cancel" - exit program

Examples

```
Popup::ModuleError( "The module " + symbolof(argterm) + " does not work." );
```

Screenshot


The module package_test.ycp doesn't work.

Name

Popup::NoHeadline -- Indicator for empty headline for popups that can optionally have one

Popup::NoHeadline

Import Popup;

```
string Popup::NoHeadline ();
```

Return Value

string	empty string ("")

Description

This is really just an alias for the empty string "", but it is slightly better readable.

Name

Popup::Notify -- Show a notification message and wait until user clicked "OK".

Popup::Notify

Import Popup;

```
void Popup::Notify (message);
string message ;
```

Parameters

string *message*	notify message string

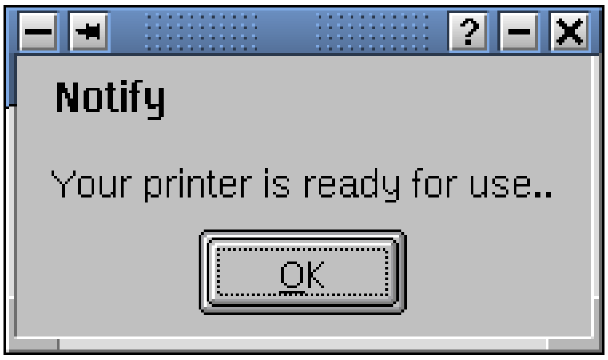
Return Value

void

Examples

```
  Popup::Notify("Your printer is ready for use." );
```

Screenshot


Notify

Your printer is ready for use..

See

- Popup::Message
- Popup::AnyMessage

Name

Popup::ReallyAbort -- Confirmation popup when user clicked "Abort".

Popup::ReallyAbort

Import Popup;

```
boolean Popup::ReallyAbort (have_changes);
boolean have_changes ;
```

Parameters

boolean *have_changes*	true: There are changes that will be lost false: No changes

Return Value

boolean	true: "abort" confirmed; false: don't abort

Description

Set "have changes" to "true" when there are changes that will be lost. Note: If there are none, it is good policy to ask for confirmation anyway, but of course with "have_changes" set to "false" so the user isn't warned about changes that might be lost.

Name

Popup::ShowFeedback -- Show popup with a headline and a message for feedback

Popup::ShowFeedback

Import Popup;

```
void Popup::ShowFeedback (headline, message);
string headline ;
string message ;
```

Parameters

string *headline*	headline of Feedback popup
string *message*	the feedback message

Return Value

void

Name

Popup::ShowFile -- Show the contents of an entire file in a popup.

Popup::ShowFile

Import Popup;

```
void Popup::ShowFile (headline, filename);
string headline ;
string filename ;
```

Parameters

string *headline*	headline text
string *filename*	filename with path of the file to show

Return Value

void

Description

Notice: This is a WFM function, NOT an UI function!

Examples

```
Popup::ShowFile ("Boot Messages", "/var/log/boot.msg");
```

Name

Popup::ShowText -- Show the contents of an entire file in a popup.

Popup::ShowText

Import Popup;

```
void Popup::ShowText (headline, text);
string headline ;
string text ;
```

Parameters

string *headline*	headline text
string *text*	text to show

Return Value

void

Examples

```
Popup::ShowText ("Boot Messages", "kernel panic");
```

Name

Popup::ShowTextTimed -- Show the contents of an entire file in a popup.

Popup::ShowTextTimed

Import Popup;

```
void Popup::ShowTextTimed (headline, text, timeout);
string headline ;
string text ;
integer timeout ;
```

Parameters

string *headline*	headline text
string *text*	text to show
integer *timeout*	text to show

Return Value

void

Examples

```
Popup::ShowText ("Boot Messages", "kernel panic");
```

Name

Popup::TimedAnyQuestion -- Timed question popup with two buttons and time display

Popup::TimedAnyQuestion

Import Popup;

```
boolean Popup::TimedAnyQuestion (headline, message,
yes_button_message, no_button_message, focus, timeout_seconds);
string headline ;
string message ;
string yes_button_message ;
string no_button_message ;
symbol focus ;
integer timeout_seconds ;
```

Parameters

string *headline*	headline or Popup::NoHeadline()
string *message*	message string
string *yes_button_message*	label on affirmative buttons (on left side)
string *no_button_message*	label on negating button (on right side)
symbol *focus*	`focus_yes (first button) or `focus_no (second button)
integer *timeout_seconds*	timeout, if 0, normal behaviour

Return Value

boolean	True if Yes, False if no

See

- Popup::AnyQuestion

Name

Popup::TimedError -- Show an error message and wait specified amount of time or until user clicked "OK".

Popup::TimedError

Import Popup;

```
void Popup::TimedError (message, timeout_seconds);
string message ;
integer timeout_seconds ;
```

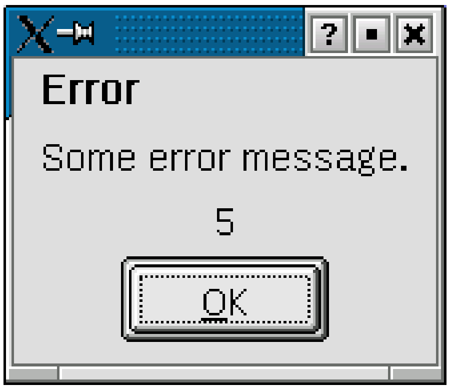
Parameters

string *message*	error message string
integer *timeout_seconds*	time out in seconds

Return Value

void

Screenshot



Error

Some error message.

5

See

- Popup::Error

Name

Popup::TimedMessage -- Display a message with a timeout and return when the user clicks "OK" or when the timeout expires.

Popup::TimedMessage

Import Popup;

```
void Popup::TimedMessage (message, timeout_seconds);
string message ;
integer timeout_seconds ;
```

Parameters

string *message*	message to display
integer *timeout_seconds*	the timeout in seconds

Return Value

void

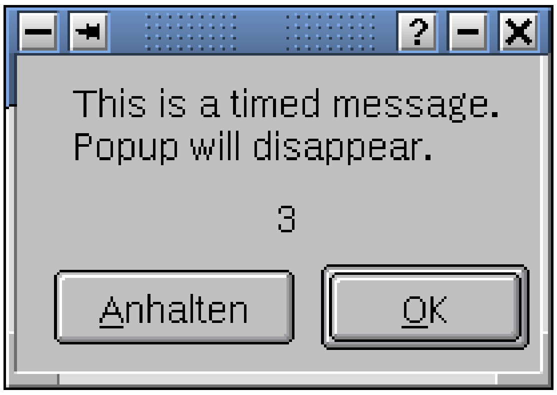
Description

There is also a "stop" button that will stop the countdown. If the user clicks that, the popup will wait forever (or until "OK" is clicked, of course).

Examples

```
Popup::TimedMessage("This is a timed message", 2 );
```

Screenshot



This is a timed message.
Popup will disappear.

3

Name

Popup::TimedOKCancel -- Display a message with a timeout

Popup::TimedOKCancel

Import Popup;

```
boolean Popup::TimedOKCancel (message, timeout_seconds);
string message ;
integer timeout_seconds ;
```

Parameters

string *message*	message to display
integer *timeout_seconds*	the timeout in seconds

Return Value

boolean	true --> "OK" or timer expired false --> "Cancel"

Description

Display a message with a timeout and return when the user clicks "OK", "Cancel" or when the timeout expires ("OK" is assumed then).

There is also a "stop" button that will stop the countdown. If the user clicks that, the popup will wait forever (or until "OK" or "Cancel" is clicked, of course).

Examples

```
boolean ret = Popup::TimedOKCancel("This is a timed message", 2 );
```

Name

Popup::TimedWarning -- Show a warning message and wait specified amount of time or until user clicked "OK".

Popup::TimedWarning

Import Popup;

```
void Popup::TimedWarning (message, timeout_seconds);
string message ;
integer timeout_seconds ;
```

Parameters

string *message*	warning message string
integer *timeout_seconds*	time out in seconds

Return Value

void

Screenshot

See

- Popup::Warning

Name

Popup::Warning -- Show a warning message and wait until user clicked "OK".

Popup::Warning

Import Popup;

```
void Popup::Warning (message);
string message ;
```

Parameters

string *message*	warning message string

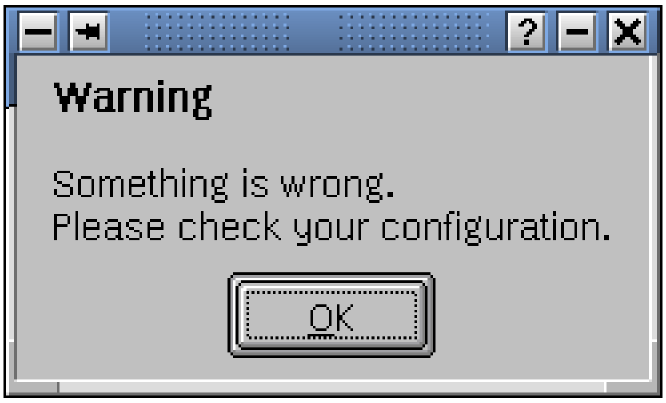
Return Value

void

Examples

```
Popup::Warning("Something is wrong. Please check your configuration." );
```

Screenshot



Warning

Something is wrong.
Please check your configuration.

See

- Popup::Message
- Popup::Notify
- Popup::Error
- Popup::AnyMessage

Name

Popup::YesNo -- Display a yes/no question and wait for answer.

Popup::YesNo

Import Popup;

```
boolean Popup::YesNo (message);
string message ;
```

Parameters

string *message*	message string

Return Value

boolean

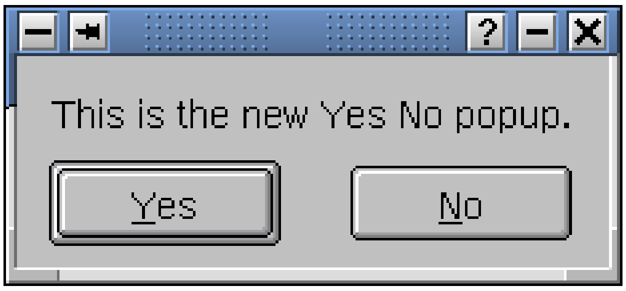
Description

Should be used for decisions about two about equivalent paths, not for simple confirmation - use "Popup::ContinueCancel()" for those. The default button is Yes. Returns true if <b>Yes</b> is clicked.

Examples

```
  Popup::YesNo ( "Create a backup of the config files?" );
```

Screenshot


This is the new Yes No popup.

See

- Popup::YesNoHeadline
- Popup::ContinueCancel
- Popup::AnyQuestion

Name

Popup::YesNoHeadline -- This dialog displays "message" (a question) and has a <b>Yes</b> and a <b>No</b> button.

Popup::YesNoHeadline

Import Popup;

```
boolean Popup::YesNoHeadline (headline, message);
string headline ;
string message ;
```

Parameters

string *headline*	short headline or Popup::NoHeadline()
string *message*	message string

Return Value

boolean

Description

It should be used for decisions about two about equivalent paths, not for simple confirmation - use "Popup::ContinueCancel()" for those. A short headline can be displayed (without headline you can use Popup::YesNo()).

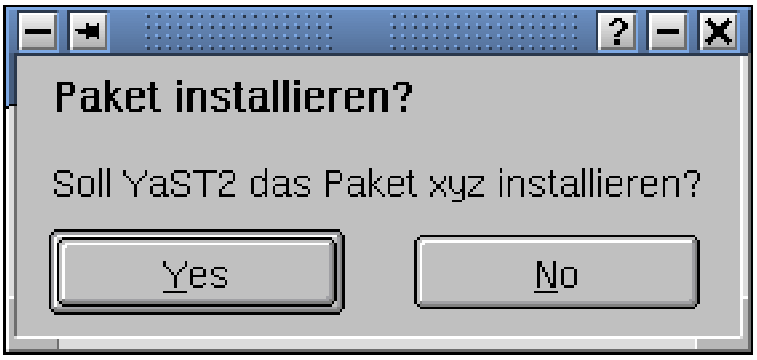
The default button is Yes.

Returns true if <b>Yes</b> is clicked.

Examples

```
Popup::YesNoHeadline ( "Resize Windows Partition?", "... explanation of dangers ..." );
```

Screenshot


Paket installieren?

Soll YaST2 das Paket xyz installieren?

See

- Popup::YesNo
- Popup::AnyQuestion