Graphical User Interface programming with SQUIRREL

Reference manual of the User Interface Add-Ons

Version 0.71
March 11, 2001

Author: Jean-Louis Villecroze
Email: jlv@kirilla.com
Web site: http://www.kirilla.com

This document was prepared with LaTeX 2_ε.

Contents

List of Tables

List of Figures

About this Document

This document covers the following SQUIRREL Add-Ons:

- GUI (Main Add-on)
- Imaging (Image viewer widget)
- Widgets (Extra set of widgets)

SQUIRREL is a programming language from the Logo family for the Be operating system (BeOS). You may consult the SQUIRREL *Developer's Guide* for a complete coverage of this language. We will assume in this document a good previous knowledge of SQUIRREL . Exposure to the *Be Developer's Guide (a.k.a BeBook)* from the *Be Development Team* could also be very useful.

At this time, neither the GUI Add-On nor this document is perfect. We would appreciate notification of any errors you may find.

This manual is divided into four parts:

Basis of GUI Building introduces all the GUI concepts

The Window discusses the Window in depth

The Widgets lists and describes all the widgets available

Supports lists and describes several useful primitives and font objects

Release notes contains pertinent information on the releases

It should be understood that several features are still to be added in upcoming releases, in particular, more widgets.

We have used in this document several documentation conventions which are :

- All code elements are presented in a distinct font like `print "foo`
- Primitive syntax is usually a mix of code element and italic font. The part in italic is always the input of the primitive.
- Primitive inputs use special kind of symbols :
  - **(word)** indicate that the input is optional
  - **word | number** indicate that the input could be either a word or a number

- **(word)+** indicate that several words could be inputted to the primitive, but at least one is required.
- **(word)*** indicate that several words could be inputted to the primitive, but that one is optional.

A big *Mahalo* to Susan Banh[1] and Ulrich ”scholly” Scholz for reading this document and correcting most of it english mistakes.

Please enjoy reading this manual and building GUI with SQUIRREL !

Jean-Louis, March 11, 2001

[1] and all my love

Chapter 1

Basis of GUI Building

Like most of the recent (or less) scripting languages like Tcl or Python, SQUIRREL disposes of a special tool for building Graphical User Interfaces, in much the same way as Tk has been added to Tcl or Python.

This Add-On to SQUIRREL is a warper to the *Be Interface Kit* and was written for the SQUIRREL programming language. This Add-On makes it easy to create widgets and frameworks to be placed on a window.

This manual concentrates on the features of the GUI Add-On of SQUIRREL rather than on the Be Interface Kit. You may consult the *Be Developer Guide* for a complete coverage of this Kit as well as other Kits.

Graphical User Interfaces are rather difficult to put on static paper as they are dynamic. We encourage you to run each example found in this chapter to get a better feeling for them. Experimenting with this Add-On and SQUIRREL is also a good way to learn GUI programming. Installing other packages other than SQUIRREL is not required since this Add-on is part of the standard SQUIRREL distribution.

Let's start now by a set of small examples to illustrate the very basics of GUI building with SQUIRREL .

1.1 Hello World

Traditionally, the first example on whatever computer language is always this one, with or without GUI. As shown below, it takes four lines of SQUIRREL to produce the code :

Example 1

```
make "hello Window "titled 'Squirrel' [100 100]
make "button Button 'Hello World !'
Glue :hello "top [] :button
$hello~show
```

That's all ... no inheritance and no class. It's that simple. A window is created when the code is executed and it's look like :

Figure 1.1: Hello World

Even a trivial example like this *Hello World* demonstrates a great deal about the common steps in GUI programming :

1. Create a window
2. Create a widget
3. Arrange the widget on a parent (a window here)
4. Brins the window to the screen

Once the window is created, SQUIRREL will wait for it to be destroyed and will process all the user-generated events. When the window appears, the widgets will be displayed only after being glued (placed) first on the window or on a parent widget.

The `Glue` primitive used to place the button on the window is the *geometry manager* which controls how the widgets are arranged in a parent widget (or a window). The first input of this primitive is always the parent followed by the side the child is to be placed. In this particular case, it would be the top side. The third input is the vertical and horizontal padding. It's a space in pixels which will separate the widgets from the side of the parent or the other widgets glued on the parent. It's always defined as a list of two integers. If the list is empty, no padding is used. The last input is the widget that needs to be glued on the parent. It could be more than one widget by adding other objects as input to the primitive.

Widget gluing is how Squirrel arranges widgets in a parent. It's a very easy and popular way of placing the widgets.

1.2 Running GUI Programs

There are different ways to run a SQUIRREL GUI program, like with any other script:

You may choose the method which fits best your needs. The use of *Package files* within a *Shell script* is the most common.

Methods	Descriptions	Command
Program file	Passing the program file as argument to SQUIRREL	`% Squirrel.dr4 myfile.sqi`
Package file	Loading the program file from SQUIRREL	`load myfile.sqi`
Shell script	Adding `#!/path/Squirrel.5` as first line of the program file	`% myfile.sqi`
Interactively	Typing the code in the SQUIRREL console	`@> make "hello Window`

Table 1.1: The 4 different methods to run a SQUIRREL script

1.3 Adding respond to the user actions

The example 1 features a button, which when the user clicks on it, could perform an operation. We didn't use this possibility in the example so we could have simply written our *Hello World* example using a simple Text widget :

Example 2

```
make "hello Window "titled 'Squirrel' [100 100]
make "text Text 'Hello World !'
Glue :hello "top [] :text
$hello~show
```

The result is:

Figure 1.2: Hello World with a Text widget

The window that we have created inherited its size from when the widgets were glued on. In our two examples, the size of the window is not the same as the size of the button and the text are not the same. The way the window was created allows the user to resize it by using the bottom right corner of the border as shown in the next figure:

Figure 1.3: Hello World resized

If we want our Example 1 to perform an action when the button is pressed by the user, we will have to insert a function to the primitive `Hook` a *callback* function (also called : hook) which can then be triggered by the user.

Example 3

```
1  to exit :src
2          $hello~quit
3  end
4
5  make "hello Window "titled 'Squirrel' [100 100]
6  make "button Button 'Hello World !'
7  Hook :button "invoked "exit
8  Glue :hello "top [] :button
9  $hello~show
```

The function `exit` will ask the window to quit, which then terminates the application when the button will be pressed. Take note of the inputs of the primitive `Hook` on line 7 : first we have the widget to register a hook for, followed by the name of the event, and then the name of the function to execute. A further in depth discussion will be given on the `Hook` primitive and *callbacks*.

Hook functions can be any kind of function or primitive. However, their inputs must match the number expected by SQUIRREL . In the Example 3 the exit function had one argument which was filled when the hook was called by SQUIRREL to the calling widget. In this example, the calling widget was the button.

1.4 Event Driving programming

SQUIRREL works in much the same way as most GUI programming languages do such as *Tcl* or *Delphi*. It's event driven. The coding of an interface starts with the creation of the widget, followed by the registration of the action to perform, when events trigger them.

Programming a GUI is a combination of event driven programming and sequential programming. Events trigger functions which in turn execute sequential instructions or generate another event, which will triggers events and so on ...

1.5 Widgets as parents

As mentioned earlier, a widget could also be a parent to other widgets. Not all the widgets could assume this role. The widget Frame demonstrates this possibility in the next example:

Example 4

```
make "hello Window "titled 'Squirrel' [100 100]
make "frame Frame
make "text Text 'Hello World !'
Glue :frame "top [] :text
Glue :hello "top [] :frame
$hello~show
```

This example builds a frame widget and glues on it a simple text widget displaying the string `'Hello World !'`. The next figure illustrates this :

Figure 1.4: Hello World with a frame

The window offers pretty much the same feature except it's a bit larger than in the Example 2. As well, the background of the text widget is now gray. All the changes are due to the frame widget, which by default bares a gray color. This is the color inherited by other widgets that get glued on.

Let's now try to glue three widgets on a frame, a text and two buttons:

Example 5

```
make "win Window "titled 'Squirrel' [100 100]
make "f Frame "flatened
make "text Text 'How\'s the weather out there ?'
make "b1 Button 'Good'
make "b2 Button 'Bad'
Glue :f "top [] :text
Glue :f "left [] :b1
Glue :f "right [] :b2
Glue :win "top [] :f
$win~show
```

This example creates three widgets, stores them in the variable `text`, `b1` and `b2` and glues them on the frame. Since we want a special layout for each widget, we need to issue three calls to the primitive `Glue` in order to glue each widget where we want:

Figure 1.5: 3 widgets in a frame

1.6 Gluing widgets

The layout of the widgets in the previous example could appear a bit surprising but it's actually what we have asked for. The *geometry manager* of SQUIRREL is working sequentially in the order given to the widget. In this example the widget `:text` will be glued first, followed by the button `:b1` and then `:b2`. When `:b1` is glued on the frame, the text widget is already there and so the *geometry manager* would place the button on the left of anything already glued:

Figure 1.6: After the second call to `Glue`

If we wanted to have the two buttons side-by-side below the text, we should have put the buttons on a new frame and glued it on the frame with the text widget like that shown in the next example :

Example 6

```
make "win Window "titled 'Squirrel' [100 100]
make "frame Frame
make "buttons Frame
make "text Text 'How\'s the weather out there ?'
make "b1 Button 'Good'
make "b2 Button 'Bad'
Glue :frame "top [] :text
Glue :buttons "left [] :b1
Glue :buttons "right [] :b2
Glue :win "top [] :frame :buttons
$win~show
```

The new frame `:buttons` is holding the two button widgets. This frame is glued in the same `Glue` call than the first frame used (holding the text widget) but following it so glued below the `:frame` frame.

Using frames is the best way to achieve what you want. *Divide and conquer* is the motto of the successful gluing strategy. We could have also given a 3D appearance to our button frame by using one of the Frame widget options:

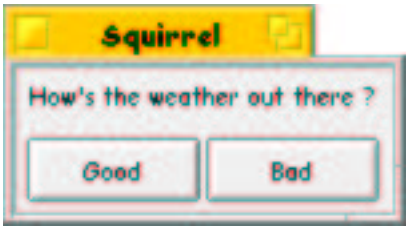


Figure 1.7: With two frames

Example 7

```
make "win Window "titled 'Squirrel' [100 100]
make "frame Frame
make "buttons Frame "raised
make "text Text 'How\'s the weather out there ?'
make "b1 Button 'Good'
make "b2 Button 'Bad'
Glue :frame "top [] :text
Glue :buttons "left [] :b1
Glue :buttons "right [] :b2
Glue :win "top [] :frame :buttons
$win~show
```

The frame widget will be discussed later but it supports several *3D Looks* like the raised look used in this example:

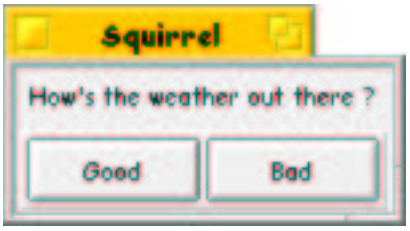


Figure 1.8: Using 3D Looking frame

When the user resizes the window (if he's allowed to), the *geometry manager* will change the position and the size of the widgets according to their configurations and to what is possible. If we try on the Example 7, we will get :

The *geometry manager* is aware of only four positions within a parent : `top bottom left right`. When both frames are glued on top of the window, their positions will not be changed when we resize the parent.

It may appear than in the Figure 1.7, that the two frames are left justified within the window. This is always the default with the *geometry manager*. We will see later in another example how to change this alignment but first let's try to set our frames to always follow the bottom side of the window :

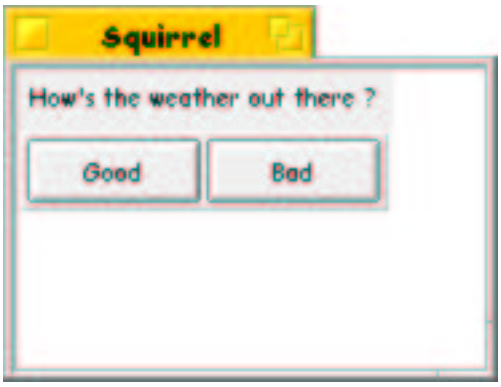


Figure 1.9: After resizing of the window

Example 8

```
1  make "win Window "titled 'Squirrel' [100 100]
2  make "frame Frame
3  make "buttons Frame "raised
4  make "text Text 'How\'s the weather out there ?'
5  make "b1 Button 'Good'
6  make "b2 Button 'Bad'
7  Glue :frame "top [] :text
8  Glue :buttons "left [] :b1
9  Glue :buttons "right [] :b2
10 Glue :win "bottom [] :frame :buttons
11 $win~show
```

The change in Example 7 is located on line 10, within the `Glue` primitive which places the two frames on the window. Instead of the top position we ask for the bottom position and the next Figure show what's happening :

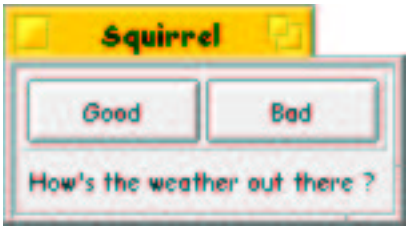


Figure 1.10: Gluing on the bottom

The ordering of the `Glue` primitive in which the widgets are given as input will determine the way the *geometry manager* will place them. When gluing on the bottom of a parent, the first widget will always be the last widget, the closer to the bottom side of the parent. To place the buttons below the text, we need to invert the widgets order as shown in the next example :

Example 9

```
make "win Window "titled 'Squirrel' [100 100]
make "frame Frame
make "buttons Frame "raised
make "text Text 'How\'s the weather out there ?'
make "b1 Button 'Good'
make "b2 Button 'Bad'
Glue :frame "top [] :text
Glue :buttons "left [] :b1
Glue :buttons "right [] :b2
Glue :win "bottom [] :buttons :frame
$win~show
```

The window looks something like the Figure 7 :

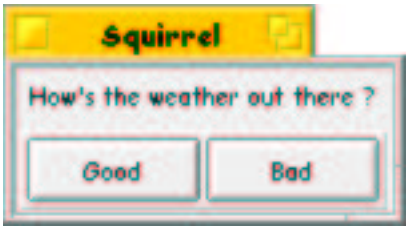


Figure 1.11: Gluing on the bottom after inverting the widgets order

The difference between the two examples is apparent when the user resizes the window:

Figure 1.12: Bottom gluing after resizing of the window

The two frames now follow the bottom side of the parent as expected. We would have obtained similar results if we had glued it on the right :

Figure 1.13: Right gluing after resizing of the window

1.7 Widget alignment within a parent

In our previous example, we were lucky. The size of the text widget was exactly the same size as that of the button's frame, which makes the window appear neat. Let's try another string for our text widget using a different button string:

Example 10

```
make "win Window "titled 'Squirrel' [100 100]
make "frame Frame
make "buttons Frame "raised
make "text Text 'Wanna skydiving now ?'
make "b1 Button 'I rather not'
make "b2 Button 'Let\'s go!'
Glue :frame "top [] :text
Glue :buttons "left [] :b1
Glue :buttons "right [] :b2
Glue :win "bottom [] :buttons :frame
$win~show
```

Figure 1.14: Different size of widgets

Two things don't appear right in this window; the white rectangle on the right and the button size are different. Let's now try to fix one of the two problems. The Button widget description, described later, will show how to make them the same size.

Each type of widget in SQUIRREL disposes some configurable settings after the creation of the widget. The one currently of interest to us is the horizontal and vertical alignment. For our next example, we are going to set the horizontal alignment :

Example 11

```
make "win Window "titled 'Squirrel' [100 100]
make "frame Frame
$frame~config "align.h "set "center
make "buttons Frame "raised
make "text Text 'Wanna skydiving now ?'
make "b1 Button 'I rather not'
make "b2 Button 'Let\'s go!'
Glue :frame "top [] :text
Glue :buttons "left [] :b1
Glue :buttons "right [] :b2
Glue :win "bottom [] :buttons :frame
$win~show
```

The only difference between this example and the previous is the new third line that calls the method `config` on the object frame. This call sets the horizontal alignment to the center, which horizontally centers the frame as shown in the next figure :

Figure 1.15: Center horizontal alignment for a frame widget

What is happening when the user resizes the window ? The *geometry manager* should change the position of each widget according to the gluing rules and the widget configurations. So if the window is wider, our text frame should always be centered like shown in the next figure :

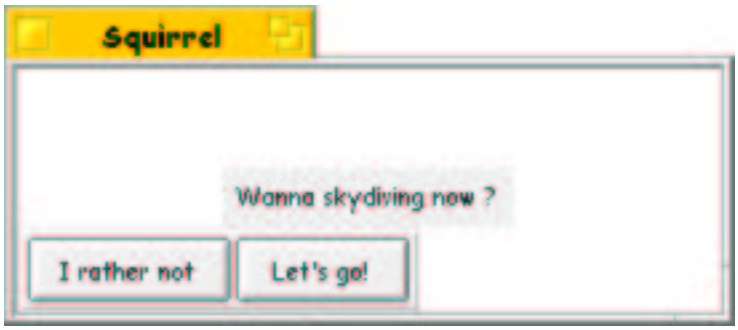


Figure 1.16: Window resized with a horizontal alignment

Example 12

```
make "win Window "titled 'Squirrel' [100 100]
make "frame Frame
$frame~config "align.h "set "center
make "buttons Frame "raised
$buttons~config "align.h "set "center
make "text Text 'Wanna skydiving now ?'
make "b1 Button 'I rather not'
make "b2 Button 'Let\'s go!'
Glue :frame "top [] :text
Glue :buttons "left [] :b1
Glue :buttons "right [] :b2
Glue :win "bottom [] :buttons :frame
$win~show
```

The same method `config` with the same input is used here. The result becomes nicer:

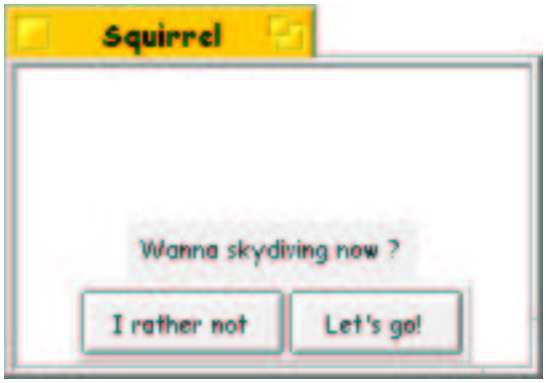


Figure 1.17: Both widgets horizontally aligned and the window is resized

What about also centering the two frames vertically also ? It should definitely be nicer ? The next example implements this solution :

Example 13

```
1  make "win Window "titled 'Squirrel' [100 100]
2  make "frame Frame
3  $frame~config "align.h "set "center
4  $frame~config "align.v "set "center
5  make "buttons Frame "raised
6  $buttons~config "align.h "set "center
7  $buttons~config "align.v "set "center
8  make "text Text 'Wanna skydiving now ?'
9  make "b1 Button 'I rather not'
10 make "b2 Button 'Let\'s go!'
11 Glue :frame "top [] :text
12 Glue :buttons "left [] :b1
13 Glue :buttons "right [] :b2
14 Glue :win "top [] :frame :buttons
15 $win~show
```

Two lines have been added (line 4 and 7) to set the configuration of the two frame widgets for the vertical alignment to be centered. We have also changed the gluing position of the frames on the window so that only the top position will be accepted for vertical alignment.

Let's now see what's happening to the widget when the user is resizing the window :

Figure 1.18: Both frame aligned vertically and horizontally when the window is resized

It's not quite what we were expecting. Although both widgets are horizontally centered, the vertical alignment is not correct. It's actually due to a limitation of the current version of the *geometry manager*. This problem will be fixed in future releases. What we were expecting was to have the two frames side by side in the middle of the window.

A working implementation would be to create a frame containing both frames, and then to align this frame in the center, vertically and then horizontally. The next example demonstrates this possibility :

Example 14

```
1  make "win Window "titled 'Squirrel' [100 100]
```

```
make "container Frame
$container~config "align.h "set "center
$container~config "align.v "set "center
make "frame Frame
make "buttons Frame "raised
make "text Text 'Wanna skydiving now ?'
make "b1 Button 'I rather not'
make "b2 Button 'Let\'s go!'
Glue :frame "top [] :text
Glue :buttons "left [] :b1
Glue :buttons "right [] :b2
Glue :container "top [] :frame :buttons
Glue :win "top [] :container
$win~show
```

A new frame widget has been created and stored in the variable `container`, and its configuration has been set to be always centered both vertically and horizontally. The two widget frames used previously has been glued on this new frame.

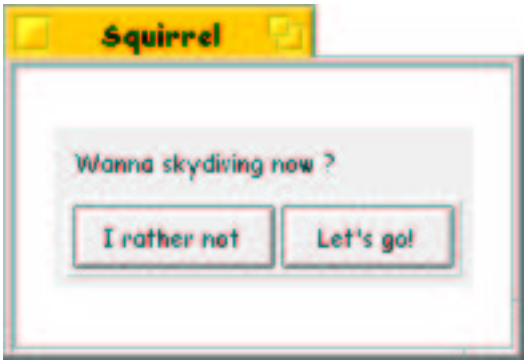


Figure 1.19: A frame centered vertically and horizontally

The result is now what we were expecting earlier, although we could have also set the text widget to be centered within its parent. We would have then obtained this nicer window :

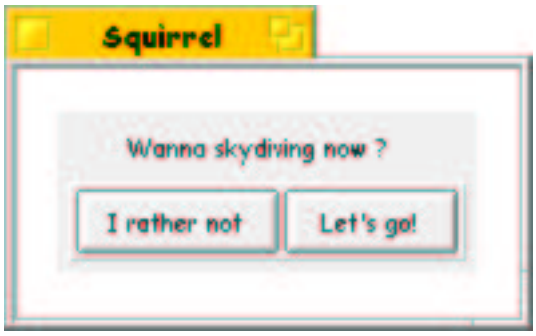


Figure 1.20: All frames centered vertically and horizontally

1.8 Expanding widgets

Recall the Example 10 which was building the window:

Figure 1.21: Different size of widgets

We could have the text frame expand itself to cover the white rectangle just by using an option from the frame:

Example 15

```
make "win Window "titled 'Squirrel' [100 100]
make "frame Frame
$frame~config "expand.x "set true
make "buttons Frame "raised
make "text Text 'Wanna skydiving now ?'
make "b1 Button 'I rather not'
make "b2 Button 'Let\'s go!'
Glue :frame "top [] :text
Glue :buttons "left [] :b1
Glue :buttons "right [] :b2
Glue :win "bottom [] :buttons :frame
$win~show
```

The difference between Example 10 and this example is the third line that we have added which set the horizontal expanding mode of the frame to true. This means that the frame could expand itself. We could check the next figure to see if the result is correct:

Figure 1.22: Frame expanded horizontally

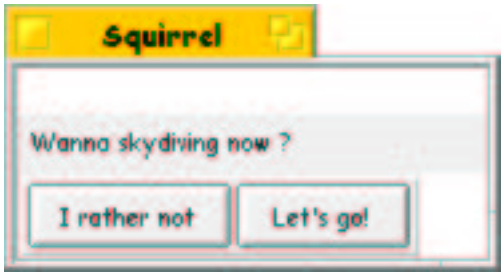


Figure 1.23: Frame expanded horizontally and window resized

What's happening now when the user resizes the window ?

The frame has correctly expanded its width with respect to the new window width. We could have also configure the text frame to expand its size vertically as the widget is glued on the bottom of the window:

Example 16

```
1  make "win Window "titled 'Squirrel' [100 100]
2  make "frame Frame
3  $frame~config "expand.x "set true
4  $frame~config "expand.y "set true
5  make "buttons Frame "raised
6  make "text Text 'Wanna skydiving now ?'
7  make "b1 Button 'I rather not'
8  make "b2 Button 'Let\'s go!'
9  Glue :frame "top [] :text
10 Glue :buttons "left [] :b1
11 Glue :buttons "right [] :b2
12 Glue :win "bottom [] :buttons :frame
13 $win~show
```

We add a new line (line 4) setting to the value true to configure the frame and give it an expanding possibility in the vertical direction(`expand.y`) The window looks like this when the user resizes it:

We could now also expand the button frame for a better look :

Example 17

Figure 1.24: Frame expanded vertically and horizontally

```
make "win Window "titled 'Squirrel' [100 100]
make "frame Frame
$frame~config "expand.x "set true
$frame~config "expand.y "set true
make "buttons Frame "raised
$buttons~config "expand.x "set true
make "text Text 'Wanna skydiving now ?'
make "b1 Button 'I rather not'
make "b2 Button 'Let\'s go!'
Glue :frame "top [] :text
Glue :buttons "left [] :b1
Glue :buttons "right [] :b2
Glue :win "bottom [] :buttons :frame
$win~show
```

It is not necessary to set the `expand.y` property of the button's frame for this frame is always between the text frame and the bottom side of the window. When resized by the user, the window will appear as:

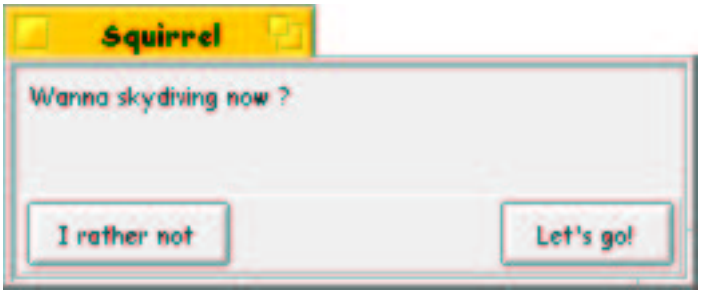


Figure 1.25: Both frame expanded

Notice the position of the button. One is on the left and the other is on the right due to the gluing configuration for the buttons.

To complete this set of simple examples of SQUIRREL GUI capabilities, let's simply set the text widget to be always centered within its parent frame:

Example 18

```
make "win Window "titled 'Squirrel' [100 100]
make "frame Frame
$frame~config "expand.x "set true
$frame~config "expand.y "set true
make "buttons Frame "raised
$buttons~config "expand.x "set true
make "text Text 'Wanna skydiving now ?'
$text~config "align.v "set "center
$text~config "align.h "set "center
make "b1 Button 'I rather not'
make "b2 Button 'Let\'s go!'
Glue :frame "top [] :text
Glue :buttons "left [] :b1
Glue :buttons "right [] :b2
Glue :win "bottom [] :buttons :frame
$win~show
```

Figure 1.26: Text widget centered in an expanded frame

Chapter 2

The Window

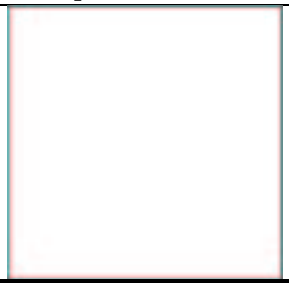
The top-level container in SQUIRREL is the Window. This widget is derived from the BWindow class from the *Be Interface Kit*. This chapter provides a complete reference to this object.

2.1 Types of Window

The Window in BeOS could bare several looks and feels, with each one giving the window a different behavior. When creating a window, it's possible to specify the type of window, look or feel. The type is actually a shortcut for a certain look and feel. According to the *BeBook* (BeOS development guide) we have devised the following looks and feels.

2.1.1 Window Look (or Type)

The following table shows and describes the difference between all the possible looks :

Look	Description	Example
`"bordered`	The window have a border, no title table and can't be moved, resized or closed by the user	

Continue on next page

Look	Description	Example
Continuing ...		
`"document`	The window have a border and a title tab with a zoom and iconify buttons. The window may be resized by the user by using the right-bottom corner tab (also called: resize corner). The gray tab around the window frame could be used to move the window.	
`"floating`	The window has a thinner border than the previous look and a smaller title. This look is usually used as a member window of an application. The resize corner (bottom-right) has been replaced by a more simple corner allowing the user to resize the window	
`"modal`	The window has a thick border and a simple resize corner but no title tab. This window disables access to any other window of the application when shown on the screen. The window can be closed by the user	
Continue on next page		

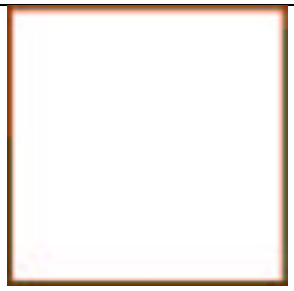
Look	Description	Example
Continuing ...		
`"no.bordered`	The window has no title tab, no border, and no resize corner. The user can close the window. (The red border in the picture is in fact part of the screen background image; it has been left to show the window which is all white)	
`"titled`	The window has a border, a title tab with a zoom and iconify button. The window may be resized by the user by using a simple resize corner. The gray tab around the window frame could be used to move the window	

Table 2.1: Window's Look

2.1.2 Window Feel

The feel of a window determines a window's behavior relative to other windows of the same application.

Name	Description
`"floating.all`	The window will float on top of any other window of the application of its subset.
`"floating.app`	The window will float on top of any other window of the application of its subset. The window will only be visible when one of the windows in the application is active.
`"floating.subset`	The window will float on top of any other window of the application of its subset. The window will only be visible when one of the windows in the subset is active.
`"modal.all`	When on screen, the window will block the activity of all other window of the applications and will be present on every screen.
`"modal.app`	When on screen, the window will block the activity of all other window of the applications and will be present on every screen. The window will be visible only if one window of the application is visible.
`"modal.subset`	When on screen, the window will block the activity of all other window of the applications and will be present on every screen. The window will be visible only if another window of the application or subset is visible.
`"normal`	The window will not float or be modal. It's the default feel of the window of type : `"titled` , `"document`, `"no.bordered` and `"bordered`.

Table 2.2: Window's Feel

2.1.3 Subset & Application

The look and feel of a window introduces the notion of *subsets* and *applications*. By default, every window created in an application is part of this application. But it's possible to create within this application several subsets of a window. Being part of a subset will only affect the modal and floating windows.

2.2 Creating a window

As mentioned in the first chapter, creating a window in SQUIRREL is done by calling the `Window` primitive. Although using this primitive is simple, several options could be inputs to the primitive in order to change the behavior of the window.

The syntax of the Window primitive is :

`Window` *word* | *list string list (word)**

The first input could be either a word or a list. It describes the look and feel of the window. When a word is given, it will be taken as the type (defined mix of Look and Feel) of the window. A list will be seen as the Look and Feel of the window and so must have two words as elements.

The second input is a string (or a word) which will be used as the title of the window. Window must have a title even if there's no title tab.

The third input is a list of two numbers which supplies the position on the screen where the window must be displayed. The first element of the list is the x-axis and the second element of the list is the y-axis.

The other input to the primitive will be seen (if they exist) as a flag to the window, describing what the user will be allowed to do on the window, such as resizing or moving. The next table describes all the flags:

Name	Description
`"accept.first.click`	The window will receive a mouse click when the window is not the active window, otherwise the window will be activated when the user clicks on it. The click will not be received by the widget which the user has clicked.
`"not.closable`	The window will not be closable by the user. The title tab will not display the usual closing button.
`"not.h.resizable`	The window will not be horizontally resizable by the user.
`"not.minimizable`	The window will not be minimizable by the user (put in the DeskBar). Double clicking on the title tab will not minimize the window.
`"not.movable`	The user will not be able to move the window around. Although, it will be possible to put the window on another screen.
`"not.resizable`	The window will not be resizable at all by the user. Neither vertically nor horizontally. Note that the two gray lines on the right bottom corner which indicate where to drag the window border for resizing.
`"not.v.resizable`	The window will not be horizontally resizable by the user.
`"not.zoomable`	The user will not be able to zoom (maximize) the window.

Table 2.3: Window's Flags

Let's now look at some examples of window creation :

Example 1

```
make "win Window "titled "Test [200 100] "not.closable
$win~show
```

The window is created with the type *titled* and will not be closable by the user as shown on the next figure :

Example 2

Figure 2.1: Not closable window

```
make "win Window "modal "Question [200 100] "not.resizable
$win~show
```

The modal window will have no title table and will block all other windows of the application.

Figure 2.2: Modal window not resizable

Example 3

```
make "win Window "document "Question [200 100] "not.zoomable
$win~show
```

2.3 Methods

When using the ability of any SQUIRREL object to call methods, a window has several primitives which are accessible only to the window.

activate

```
$window~activate
```

Figure 2.3: Simple window without zoom button

Make the window the active window.

add.to.subset

```
$window~add.to.subset window
```

Add a window given as input to the window's subset.

bounds

```
$window~bounds
```

Output the bounds of the window as a list of four numbers (left-top right-bottom).

center

```
$window~center word (list)
```

Set or get the center of the window. When the first input is the word `"get`, the method output a list that contain the coordinates on the screen of the center of the window. If the input is `"set`, the method need a second input that must be a list of two numbers. The window then move on the screen to center itself on those coordinates.

close

```
$window~close
```

Close the window. If the window is the last window of the application, the application will be terminated. This method has the same effect than the `Quit` method.

deactivate

```
$window~deactivate
```

If the window was the active window, the window will lose its active status.

enable

`$window~enable boolean`

If the input is `true`, all the widgets will be enabled to the user. If the input is `false`, all the widgets will be disabled and the user will not be able to interact with them.

frame

`$window~frame`

Output the frame of the window as a list of four numbers.

hide

`$window~hide`

The window is removed from the screen but not destroyed. It is hidden from the user.

is.active

`$window~is.active`

Output `true` if the window is the active window, `false` if not.

is.front

`$window~is.front`

Output `true` if the window is the front most window on the screen, `false` if not.

is.hidden

`$window~is.hidden`

Output `true` if the window is hidden, `false` if not.

minimize

`$window~minimize`

The window is removed from the screen and placed on the DeskBar.

move.by

```
$window~move.by [horizontal vertical]
```

Shift the position of the window by the value given horizontally and vertically. A positive value is given for a shift right or a shift to the top. A negative value is given for a shift left or a shift to the bottom.

move.to

```
$window~move.to [x y]
```

The window is moved to a new position on the screen as given by the coordinates of the new left upper corner of the window.

quit

```
$window~quit
```

Close the window. If the window is the last window of the application, the application will be terminated. Using the keyboard shortcut `COMMAND-Q` will have the same effect than calling this method. If a menu use the same shortcut, the menu callback will NOT be executed.

reglue

```
$window~reglue
```

Restart the gluing of all the widgets of the window. Usually done when a widget had been removed or resized within its parent.

rem.from.subset

```
$window~rem.from.subset window
```

Remove a window given as input from the window's subset.

resize.by

```
$window~resize.by number number
```

Resize the window by the value given as inputs (width, height).

resize.to

```
$window~resize.to number number
```

Resize the window to the value given as inputs (width, height).

show

`$window~show`

The window is displayed on the screen. This method is used after the window has been hidden or when the window has been created.

unmimimize

`$window~unminimize`

The window is ”unmimimized” from the desk bar and displayed on the screen. The method is the reverse of minimize.

widgets

`$window~widgets`

Output a list of all the widgets glued on the window.

2.4 Configuration

One of the window’s methods allows one to set or get the window configuration. This method is `config` and follows the syntax :

`$window config "get` *word*

or

`$window config "set` *word thing*

Using `"get` as a first argument will retrieve the value for the specified configuration, given as the second input. `"set` will set the configuration to the value of the third input. The configuration of a Window may be changed at anytime during the application’s lifetime.

One word about `constraint` and `limit`: By default, the `constraint` is set to `"auto`. The window will in this case not allow the user to resize the window smaller than the size that fit perfectly the window contents. If this config is set to `"none`, the user will be allowed to resize the window however he want. In `"manual`, the maximun and minimum size can be set by the script with the config `"limit`. This config take as third input the word `"max` or `"min` wheter you want to set the minimum size or the maximum.

Item	Description	value
`"constraint`	Window size constraint	A valid word (`"auto` `"none` `"manual)`
`"defaultbutton`	Button by default of the window. When the user hits the Enter key of the keyboard, the window is active and this button will be invoked.	A button object is needed as input to set
`"feel`	Feel of the window	A valid word
`"focus`	Widget of the window having the focus	A widget object glued on the window
`"font`	Default font used by the widgets of the window	A font object
`"limit`	Size limit of the window	A list of 2 integers
`"look`	Look of the window	A valid word
`"pulserate`	How often the widget of the window will receive the pulse event (in ms)	An integer
`"title`	Title of the window	A string or word
`"zoom`	Maximum size the window could take when the user zoom it	A list of 2 integers (width and height)

Table 2.4: Window's Configuration

2.5 Hooks

Like widgets, functions could be defined in SQUIRREL to serve as *callbacks* for events generated by the user on the window. Those hooks could be used to perform several tasks according to the application's need.

The following table describes all the possible hooks.Note that the name of the callback function could be anything.

Name	Description	Function prototype

enter	The mouse pointer enters the window frame	`to enter :win`
		`; win is the window object`
		`end`

leave	The mouse pointer leaves the window frame	`to leave :win`
		`; win is the window object`
		`end`

maximize	The window has been unminimized	`to maximize :win`
		`; win is the window object`
		`end`

minimize	The window has been mimimized (put in the Desk Bar)	`to minimize :win`
		`; win is the window object`
		`end`

Continue on next page

Name	Description	Function prototype
Continuing ...		

move	The window has been moved within the screen	`to move :win ::x :y`
		`/* win is the window object`
		`x and y are the new coordinates`
		`of the left-top corner of the win-`
		`dow */`
		`end`

quit	The window has been asked to quit. The function should return `true` if the window must quit, `false` else	`to quit :win`
		`; win is the window object`
		`end`

resize	The window has been resized by the user	`to resize :win :w :h`
		`; win is the window object`
		`; w is the new width (integer)`
		`; h is the new height (integer)`
		`end`

workspaceactivate	The workspace where the window is, has become the active workspace or has lost this status	`to wsactivate :win :ws :status`
		`; win is the window object`
		`; ws are the workspace number (in-`
		`teger)`
		`; status is true when the workspace`
		`is active, false else.`
		`end`

workspacechange	The window has been moved to another workspace	`to wschange :win :old :new`
		`; win is the window object`
		`; old is the previous workspace`
		`number`
		`; new is the new workspace number`
		`end`

zoom	The user has zoomed the window	`to zoom :win :x :y :w :h`
		`; win is the window object`
		`; x and y are the new left-top cor-`
		`ner coordinate of the window`
		`; w and h and the new width and`
		`height`
		`end`

Table 2.5: Window's Hooks

Chapter 3

The Widgets

All the graphical elements of SQUIRREL called *widgets* are even with their differences of the same type and share a number of common methods, configurations and hooks.

This version of the GUI Add-on contains twelve basic widgets. This number will increase with every release of this Add-on.

3.1 The widget

Like a window, any widget will have a set of methods, configurations and hooks. This section describes what's common to all widgets. We will evaluate the flags specified during the creation of a widget, which allows us to invoke more behavior.

3.1.1 Methods

config

```
$widget~config word word (thing)
```

Get or set the element of the configuration of the widget. The first input is a word which indicates the configuration to access. The second input must be the word `"get` or `"set`. When setting a configuration, a third input is requested, otherwise the method will output the current value of the configuration.

enable

```
$widget~enable boolean
```

Set if the widget must be enabled or disabled to the user's action.

invalidate

```
$widget~invalidate
```

Force the redraw of the widget.

is.enable

```
$widget~is.enable
```

Output `true` if the widget is enabled, `false` if not.

is.focus

```
$widget~is.focus
```

Output `true` if the widget has the keyboard focus, `false` if not.

3.1.2 Configuration

Like for the window, using the method `config` allows us to change the configuration of the widget.

For the vertical alignment of a widget, the valid words are : `"top "center "bottom` and for the horizontal alignment there is : `"left "center` and `"right`.

Configuration	Purpose	Value
`"align`	Vertical and Horizontal Alignment of the widget within its parent	two words describing the horizontal and the vertical alignment. The method will output a list of the alignment when it gets the configuration
`"align.v`	Vertical alignment of the widget within its parent	a valid word
`"align.h`	Horizontal alignment of the widget within its parent	a valid word
`"bgcolor`	Background color of the widget	a list describing a color
`"cursor`	Cursor to use for the widget	a word
`"expand`	Widget will expand its size both vertically and horizontally	two booleans describing the horizontal and vertical expanding. The method will output a list of the expansion when it gets the configuration
`"expand.x`	Widget will expand its size horizontally	a boolean
`"expand.y`	Widget will expand its size vertically	a boolean

Continue on next page

Configuration	Purpose	Value
Continuing ...		
`"font`	Font of the widget	a font object
`"low.color`	Low color of the widget	a list describing a color
`"high.color`	High color of the widget	a list describing a color
`"pad`	Horizontal and vertical padding of the widget	two numbers describing the horizontal and vertical padding. The method will output a list of padding when it gets the configuration
`"pad.x`	Horizontal padding of the widget	a number
`"pad.y`	Vertical padding of the widget	a number

Table 3.1: Widget's common configuration

The widget cursor can be any of the following :

Name	Cursor
`"arrow`	
`"cross`	
`"cut`	
`"downarrow`	
`"hand`	
`"hcross`	
`"hourglass`	
`"ibeam`	
`"leftarrow`	
`"linkhand`	
`"macwatch`	
`"pencil`	
`"rightarrow`	
`"timer`	
`"uparrow`	
`"zoom`	

Table 3.2: Widget's cursors

3.1.3 Hooks

All the widgets in SQUIRREL could be set to handle events by using the `Hook` primitive. The following table summarizes all the common hooks :

Name	Description	Function prototype

`"activated`	The window containing the widget has been activated by the user	`to activate :src`
		`; src is the widget object`
		`end`

Continue on next page

Name	Description	Function prototype
Continuing ...		
"attached	The widget has been attached to a window (glued)	to attached :src :win ; src is the widget object ; win is the window to which the widget is attached end
"detached	The widget has been detached from a window (unglued)	to detached :src :win ; src is the widget object ; win is the window to which the widget was attached end
"draw	The widget is drawn on-screen	to draw :src :rect ; src is the widget object ; rect is a list of 4 numbers which define the updated rectangle of the widget end
"drop	The widget is the target of a drag and drop action	to drop :src :type :msg ; src is the widget object ; type is the type of data dropped in the widget : "simple "mime "archive ; msg is the message that hold the data end
"entered	The mouse cursor has entered the widget	to entered :src :x :y ; src is the widget object ; x and y is the coordinate of the position where the cursor has entered (in the widget coordinate system) end
"exited	The mouse cursor has exited the widget	to exited :src :x :y ; src is the widget object ; x and y is the coordinate of the position where the cursor has left (in the widget coordinate system) end
"focused	The widget has became the widget having the keyboard focus on the window, or lost this status	to focus :src :focus ; src is the widget object ; focus is true if the widget has the focus, false if it has lost it end

Continue on next page

Name	Description	Function prototype
Continuing ...		
"keydown	A key of the keyboard has been pushed	to keydown :src :mod :key :code ; src is the widget object ; mod is a word (or a list) describing the modifiers used by the user ; key is a string version of the key ; code is a serial code number for the pressed key end
"keyup	A key of the keyboard has been released	to keyup :src :mod :key :code ; src is the widget object ; mod is a word (or a list) describing the modifiers used by the user ; key is a string version of the key ; code is a serial code number for the released key end
"mousedown	A button of the mouse has been clicked over the widget.	to mousedown :src :mod :x :y :b ; src is the widget object ; mod is a word (or a list) describing the modifiers used by the user ; x and y is the coordinate where the mouse was when the button was hit ; b is the button number (left = 1 ...) end
"mouseup	A button of the mouse has been released over the widget.	to mouseup :src :mod :x :y :b ; src is the widget object ; mod is a word (or a list) describing the modifiers used by the user ; x and y is the coordinate where the mouse was when the button was released ; b is the button number (left = 1 ...) end
"moved	The widget has been moved within its parent	to moved :src :x :y ; src is the widget object ; x and y is the new coordinate of the left top corner of the widget end
"pulse	The widget received a pulse from the window	to pulse :src ; src is the widget object end

Continue on next page

Name	Description	Function prototype
Continuing ...		

`"resized`	The widget has been resized within its parent.	`to resized :src :w :h`
		`; src is the widget object`
		`; w is the new width of the widget`
		`; h is the new height of the widget`
		`end`

Table 3.3: Widget's Hooks

For the hook `keydown`, `keyup` and `mousedown`, the modifers used when the event occurs are the words :

- `"left_shift`
- `"right_shift`
- `"left_control`
- `"right_control`
- `"left_option`
- `"right_option`
- `"left_alt`
- `"right_alt`

Note than the following hooks:

- `"entered`
- `"exited`
- `"focused`
- `"invoked`
- `"selected`
- `"changed`

runs on their own thread when executed.

3.1.4 Flags

All widgets accept as a last input several words which describe some behavior. The following table lists them all :

Name	Purpose
`"navigable`	The widget can become the focus widget of its window for a keyboard event. (already the default by some widget)
`"navigable.jump`	Pressing `Control-Tab` on the widget will jump the focus to another group of widgets set with the same flags
`"pulsed`	The widget should receive the pulse event from its window

Table 3.4: Widget's Flags

3.2 The widget Banner

A ”Banner” is a simple widget displaying a text linked to a variable. Since all the widgets are linked to a variable, the variable modification will update the widget.

3.2.1 Construction

The primitive `Banner` is used to build a new Banner widget. Its syntax is :

`Banner` *word word (list (words))*

The first input word is the variable name given to the widget. If the variable doesn’t already exist, it will be created. The second word is the justification for the text within the widget. It must be the word : `"center` `"left` or `"right`. The third input, if specified, is a list which indicates the size of the widget in characters. This list has two elements : width and height. An empty list will be the same as no list and the size of the widget will adapt to fit the size of the displayed text. All the inputs left are the flags which may be specified. The primitive outputs the widget object.

3.2.2 Methods

The Banner widget has a few methods uncommon to all the widgets :

justify

```
$banner~justify word (word)
```

Set or get (according to the value of the first input : `"set` or `"get`) the text justification in the widget. When setting a value, the second input must be one of the valid words : `"left` `"center` or `"right`.

text

```
$banner~text word (string | word)
```

Set or get (according to the value of the first input : `"set` or `"get`) the text displayed by the widget. The second input could be a string or a word. The linked variable to the widget will have its value changed if we set a new text to be displayed.

3.2.3 Configuration

Only one configuration is added to the Banner widget :

3.2.4 Hooks

The Banner widget has no more hooks than the common widget.

Configuration	Purpose	Value
`"variable`	Set or get the linked variable of the widget	the name of the variable (a word)

Table 3.5: Banner's configuration

3.2.5 Example

Example 1

```
1  make "win Window "titled 'Banner' [100 100]
2  make "msg 'Click on the button!'
3  make "widget Banner "msg "center
4  make "button Button 'The Button'
5  $button~config "expand.x "set true
6  Hook :button "invoked {
7          make "msg 'You done it!'
8  }
9  Glue :win "top [] :widget :button
10 $win~show
```

In this example, the linked variable is updated when the button is invoked. Lines 6 to 8 set the hook for the button. Line 7 just sets a new value to the variable. The widget is updated at this moment:

Figure 3.1: Banner updated

3.3 The widget BarberPole

A *BarberPole* is a widget that display a barberpole that can be started to stoped. Usually this widget is used to show activity without knowing how long it's gonna take.

3.3.1 Construction

The primitive `BarberPole` is used to build a new widget. Its syntax is :

`BarberPole` *list (word)*

The first input is a list of two number that indicate the size of the widget in pixel (with height). If a second input is given, it must be the word `"left` or `"right`. It indicate the direction the BarberPole must run. By default it is left to right.

3.3.2 Methods

A BarberPole widget has two methods :

start

```
$barber~start
```

The method start the widget spinning.

stop

```
$barber~stop
```

The method stop the widget from spinning.

3.3.3 Configuration

This widget have nothing particular. To control the color of the barberpole, use `high.color` and `low.color`.

3.3.4 Hooks

This widget doesn't have any particular hook.

3.3.5 Example

Example 2

```
make "MyWin Window "titled 'BarberPole' [100 100] "not.closable
make "pole BarberPole [30 10]
$pole~config "high.color "set :Blue
Glue :MyWin "top [] :pole
$MyWin~show

$pole~start
for ["i 1 2] {
        wait 1
}
$pole~stop

$MyWin~quit
```

In this example, we create a simple Window with only a BarberPole in it, then we set it spinning and we execute a loop that will take 2 seconds to complete, then we stop the the BarberPole and we ask the Window to quit. :

Figure 3.2: Spining BarberPole

3.4 The widget Box

A ”Box” is a container widget which draws a labeled border around its children. A Box has three styles of border (`"plain "fancy` or `"none`). The label drawn by the widget is usually text, but it could also be another widget.

3.4.1 Construction

The primitive `Box` is used to build a new Box widget. Its syntax is :

`Box` *word* | *widget (list (words))*

The first input is the label of the Box; it could be either a string or a widget. The primitive accepts a second and third input if needed. They are the size of the widget in a list in pixels (width and height) and a set of the usual widget flags.

3.4.2 Methods

A Box widget has three methods :

reglue

```
$box~reglue
```

This primitive asks the *geometry manager* to glue all the widgets within the Box a second time. This primitive is useful when a child from the Box is removed and new gluing is needed.

style

```
$box~style word (word)
```

Set or get (according to the value of the first input : `"set` or `"get`) the style of the border. When setting a value, the second input must be one of the valid words : `"plain "fancy` or `"none`.

widgets

```
$box~wigets
```

Output all the widgets glued on the Box.

3.4.3 Configuration

Only one configuration is added to the Box widget :

Configuration	Purpose	Value
`"label`	Set or get the label of the widget	could be a word, a string or another widget.

Table 3.6: Box's configuration

3.4.4 Hooks

This widget has nothing particular.

3.4.5 Example

Example 3

```
1  make "win Window "titled 'Box' [100 100]
2  make "box Box 'A Box'
3  make "frame Frame "flattened [50 50]
4  Glue :box "top [] :frame
5  Glue :win "top [] :box
6  $win~show
```

On line 3, we create a 50x50 pixel Frame widget. It's used in this example to fill the Box widget.

Figure 3.3: Box with a text label

In the next example, we use a button to label the Box :

Example 4

```
1  make "win Window "titled 'Banner' [100 100]
2  make "label Button 'Click me'
3  make "box Box :label
4  make "frame Frame "flattened [50 50]
5  Glue :box "top [] :frame
6  Glue :win "top [] :box
7  $win~show
```

The primitive `Box` on line 3 calls the button object instead of a simple string for labeling the Box.

Figure 3.4: Box with a widget label

3.5 The widget Button

This widget is a labeled button which executes a function or a block when clicked or operated with the keyboard.

3.5.1 Construction

The primitive `Button` is used to build a new Button widget. Its syntax is :

`Button` *word* | *string (list (words))*

The first input is the label of the button. The primitive accepts optional second and third inputs. The second input must be two integers, specifying the width and height of the button in characters. The last input is the flags.

3.5.2 Methods

invoke

`$button~invoke`

Execute the hook from the event `invoked` by the button.

default

`$button~default boolean`

Make the button the default button for the window if the input is `true`, else the widget has lost this status.

is.default

`$button~is.default`

Output `true` if the widget is the default button for the window, `false` otherwise.

3.5.3 Configuration

Only one configuration is added to the Button widget :

Configuration	Purpose	Value
`"label`	Set or get the label of the widget	could be a word or a string.

Table 3.7: Button's configuration

Name	Description	Function prototype
"invoked	The button has been clicked	to invoked :src ; src is the widget object end

Table 3.8: Button's hooks

3.5.4 Hooks

3.5.5 Example

Example 5

```
1  make "win Window "titled 'Button' [100 100]
2  make "b1 Button 'Doing something'
3  make "b2 Button 'Doing nothing'
4  Glue :win "top [] :b1 :b2
5  $win~show
```

When the button's label is not the same size as that in the example 5, the result is not very nice.

Figure 3.5: Unsized buttons

One solution that has already been shown in the first chapter is to set up the buttons to expand their size to fill the empty place on the window. This is shown in the next example :

Example 6

```
1  make "win Window "titled 'Button' [100 100]
2  make "b1 Button 'Doing something'
3  make "b2 Button 'Doing nothing'
4  $b1~config "expand.x "set true
5  $b2~config "expand.x "set true
6  Glue :win "top [] :b1 :b2
7  $win~show
```

Line 4 and 5 set the `expand.x` of the two buttons.

Another solution is to fix the character's width in both buttons :

Example 7

Figure 3.6: Expanded buttons

```
1  make "win Window "titled 'Button' [100 100]
2  make "b1 Button 'Doing something' [14 0]
3  make "b2 Button 'Doing nothing' [14 0]
4  Glue :win "top [] :b1 :b2
5  $win~show
```

On line 2 and 3 you will notice the second input to the primitive `Button`. This two element list gives the width and height of the button. Here, the height is 0 for both buttons. This is interpreted by SQUIRREL as a free dimension and will therefore be set by the *geometry manager*.

Figure 3.7: Fixed buttons

One of the problems with this solution is the difficulty getting the size of the right widget (as we can see from the previous figure). We set the size to 14 characters but both of the buttons are bigger. This is due to the fact that the font used is a TrueType font. If the font was a fixed size font like *Monospac821*, we would have achieved a correct size for the button. This can be seen in the next example:

Example 8

```
1  Font.init
2  make "font Font 'Monospac821 BT'
3  make "win Window "titled 'Button' [100 100]
4  $win~config "font "set :font
5  make "b1 Button 'Doing something' [14 0]
6  make "b2 Button 'Doing nothing' [14 0]
7  Glue :win "top [] :b1 :b2
8  $win~show
```

Line 1 and 2 create a font using one of the system's font `Monospac821 BT`. On line 4, we set this font to the default font for the window.

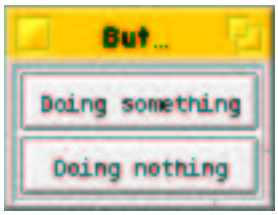


Figure 3.8: Fixed buttons with a fixed-size font

Now the sizes of the buttons match 14 characters.

3.6 The widget CheckBox

This widget is a labeled check widget. This widget changes its state (true or false) according to the user's actions : mouse clicks or keyboard stroke. This widget is linked to a variable which will be updated when the state of the widget changes. In addition, an update for the variable will change the widget's state.

3.6.1 Construction

The primitive `CheckBox` is used to build a new CheckBox widget. Its syntax is :

`CheckBox` *word* | *string word (list (words))*

The first input (either a word or a string) is the label displayed by the widget. The second input is the linked variable name. If the variable doesn't already exist, it will be created. If specified, the following inputs will be the size of the characters (two integers) and some flags.

3.6.2 Methods

invoke

```
$checkbox~invoke
```

Invoke the widget as though the widget has been clicked.

3.6.3 Configuration

A CheckBox widget has three specific configuration items :

Configuration	Purpose	Value
`"label`	Set or get the label of the widget	could be a word or a string.
`"value`	Set or get the state of the widget	`true` or `false`
`"variable`	Set or get the linked variable of the widget	a word

Table 3.9: CheckBox's configuration

3.6.4 Hooks

Name	Description	Function prototype
"invoked	The widget has been clicked	to invoked :src :state ; src is the widget object ; state is the state of the widget (true or false) end

Table 3.10: CheckBox's hooks

3.6.5 Example

Example 9 allows the user to select several systems to be checked. When the user clicks on a button, the checking will begin.

Example 9

```
1  make "win Window "titled 'CheckBox' [100 100]
2  make "box Box 'Check systems'
3  make "c1 CheckBox 'Power' "power
4  make "c2 CheckBox 'A/C' "ac
5  make "c3 CheckBox 'Computers' "computers
6  make "c4 CheckBox 'Life Systems' "life
7  make "c5 CheckBox 'Cryogenic Systems' "cryo
8  make "do Button 'Check now'
9  $do~config "expand.x "set true
10 Hook :do "invoked {
11 if :power {
12 Question "warning ["Proceed "Cancel] '' 'Please confirm the power test.'
13 }
14 }
15 Glue :box "top [] :c1 :c2 :c3 :c4 :c5
16 Glue :win "top [] :box :do
17 $win~show
```

Lines 3 to 7 create all the CheckBox widgets. For each widget, we give a variable name. The variables will be created and their value will be set to `false` by default. On line 12, we use the primitive `Question` that still needs to be described in the next chapter. This primitive simply creates a message window.

The next example shows how we could change the state of a CheckBox by changing the linked variable value :

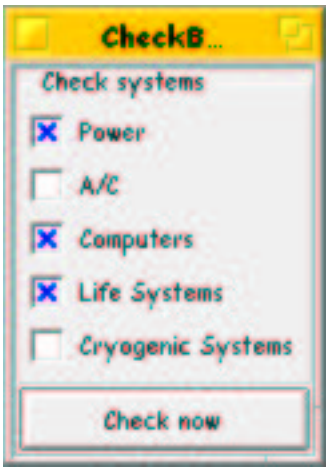


Figure 3.9: Several CheckBox

Example 10

```
make "win Window "titled 'CheckBox' [100 100]
make "box Box 'Check systems'
make "life true
make "c1 CheckBox 'Power' "power
make "c2 CheckBox 'A/C' "ac
make "c3 CheckBox 'Computers' "computers
make "c4 CheckBox 'Life Systems' "life
make "c5 CheckBox 'Cryogenic Systems' "cryo
make "do Button 'Check now'
$do~config "expand.x "set true
Hook :do "invoked {
if :power {
if (Question "warning ["Proceed "Cancel] '' 'Please confirm the power test.') {
        make "power false
}
}
}
Glue :box "top [] :c1 :c2 :c3 :c4 :c5
Glue :win "top [] :box :do
$win~show
```

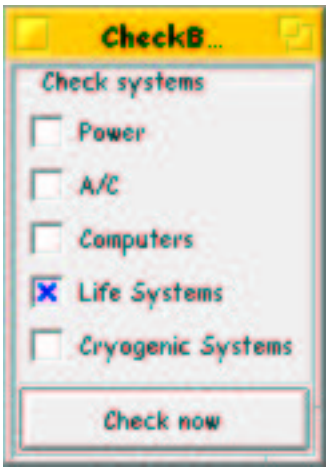


Figure 3.10: CheckBox with a default state

On line 3, we have set the variable `life` to `true`. The widget `c4` will be checked. The button `do` will be invoked, if the user cancels the *Power Systems* test. The variable `power` will then be set to false, and the widget will be updated :

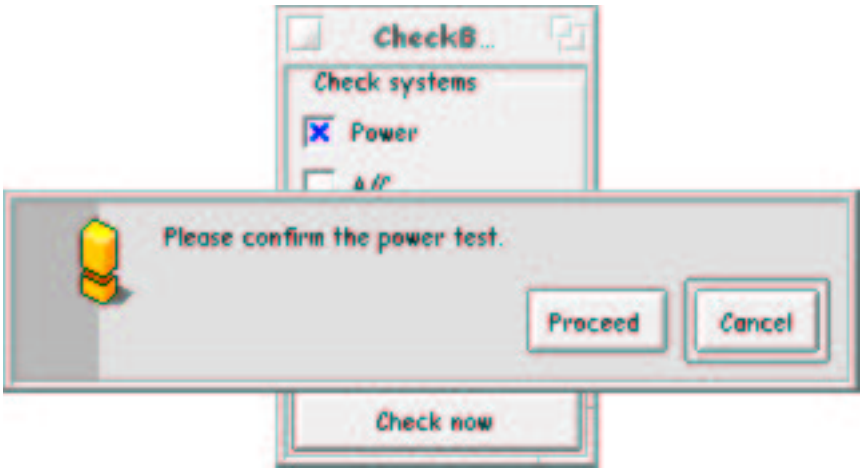


Figure 3.11: Button invoked

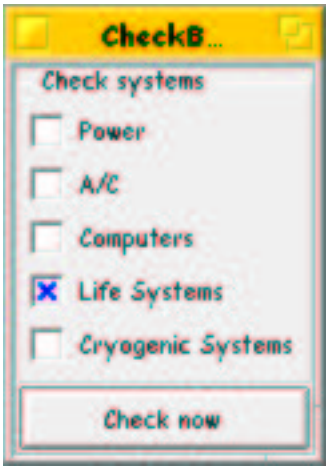


Figure 3.12: CheckBox's state updated by the variable's value change

3.7 The widget ColorControl

This widget permits the user to pick a color by choosing the RGB components of the color. The widget is linked to a variable. The available colors in this widget are functions of the screen configuration.

3.7.1 Construction

The primitive `ColorControl` is used to build a new ColorControl widget. Its syntax is :

`ColorControl` *number number word*

The first input is the number of cells to displayed by row. It should be either 4, 8, 16, 32 or 64. The second input is the size of the cell. Both inputs will set the size of the widget. The last input is the name of the linked variable.

The value of the variable will be a *color list*. It's a list of three (or four) integers which describe the color in RGB. The first element is red, the second is green and the third is blue. A fourth element could be specified, and it would be the Alpha component of the color.

3.7.2 Methods

invoke

`$colorcontrol~invoke`

Invoke the widget as though the selected color has been changed by the user

3.7.3 Configuration

A ColorControl widget has four specific configuration items :

Configuration	Purpose	Value
`"cellside`	Set or get the size of the cell side	a number
`"layout`	Set or get the number of cell by row	the number 4 8 16 32 or 64
`"value`	Set or get the state of the widget	a color list
`"variable`	Set or get the linked variable of the widget	a word

Table 3.11: ColorControl's configuration

3.7.4 Hooks

Name	Description	Function prototype
"invoked	The selected color has been changed	to invoked :src :color ; src is the widget object ; color is the new color selected end

Table 3.12: ColorControl's hooks

3.7.5 Example

Example 11

```
1  make "win Window "titled 'CheckBox' [100 100]
2  make "color ColorControl 32 5 "thecolor
3  make "get Button 'Get the color'
4  $get~config "expand.x "set true
5  Hook :get "invoked {
6  Info "info ["ok] '' 'You have selected the color' string :thecolor
7  }
8  Glue :win "top [] :color :get
9  $win~show
```

On line 2, we create the ColorControl widget. A layout and cell size of 32 and 5 is the most common. The button `get` will display an information *Message Box* displaying the color selected by the user.

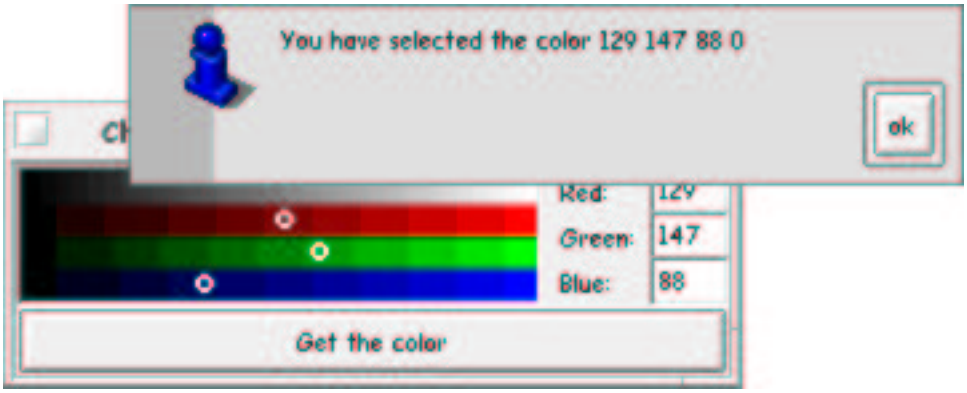


Figure 3.13: A ColorControl

In the next example, we are going to use the hook `invoked` by the ColorControl to change the background color of a Frame widget.

Example 12

```
1  make "win Window "titled 'CheckBox' [100 100]
2  make "color ColorControl 32 5 "thecolor
3  Hook :color "invoked {
4          $frame~config "bgcolor "set :thecolor
5  }
6  make "frame Frame "flattened [0 20]
7  $frame~config "expand.x "set true
8  Glue :win "top [] :color :frame
9  $win~show
```

The hook set on line 3 will modify the configuration of the Frame widget `frame`. On line 6, we have created the Frame. We have set the size of this frame to be 20 pixels height. The width is set to 0 and will adapt itself to whatever window size (with help of the `expand.x` configuration on line 7).

Figure 3.14: A ColorControl

3.8 The widget DropList

This widget is a labeled drop down list of items. When the user clicks on the drop down list, all the possible items are displayed. The widget always displays the current selection. A variable is linked to the widget.

3.8.1 Construction

The primitive `DropList` is used to build a new DropList widget. Its syntax is :

`DropList` *string* | *word word list (list (words))*

The first input is the label of the widget. The second input is the name of the linked variable. The third is the list of items. This could be any kind of data like a word, a string or a number. If specified, a fourth input will be the size of the widget in characters and any other input would make up the flags.

3.8.2 Methods

This widget doesn't have specific methods.

3.8.3 Configuration

A DropList widget has three specific configuration items :

Configuration	Purpose	Value
`"label`	Set or get the label of the widget	a string or a word
`"value`	Set or get the selected item	a thing
`"variable`	Set or get the linked variable of the widget	a word

Table 3.13: DropList's configuration

3.8.4 Hooks

Name	Description	Function prototype

`"selected`	The user has selected another item	`to selected :src :index :value`
		`; src is the widget object`
		`; index is the index in the list of the selected item`
		`; value is the selected value end`

Table 3.14: DropList's hooks

3.8.5 Example

Example 13

```
make "win Window "titled 'DropList' [100 100]
make "weather "Rainy
make "list DropList 'How\'s the weather ?' "weather ["Sunny "Rainy "Overcast]
make "button Button 'Make it better'
$button~config "align.h "set "center
Hook :button "invoked {
        make "weather "Sunny
}
Glue :win "top [] :list :button
$win~show
```

When the user clicks on the button `button`, the linked variable is changed to the value `Sunny` and the DropList widget will be updated.

Figure 3.15: A DropList

3.9 The widget Entry

This widget is a simple labeled text field. After modifying the text by pressing the *Enter* key or by changing the focus, an event will be generated and the linked variable will be updated.

3.9.1 Construction

The primitive `Entry` is used to build a new Entry widget. Its syntax is :

`Entry` *string* | *word word (list (words))*

The first input is the label of the widget. The second input is the name of the linked variable. If specified, a fourth input will be a list containing the size of the label in characters and the size of the entry field in characters. All other inputs would make up the flags.

The widget adapt itself to the value of the linked variable. If the value is a number, only number will be allowed in the Entry and once changed by the user, the new value set to the linked variable will be a number.

3.9.2 Methods

invoke

```
$entry~invoke
```

Invoke the widget as though the text was modified by the user.

entry

```
$entry~entry word  word (string | word)
```

Set or get the configuration of the entry field. The first input must be the word `"align` or `"expand`. When `"align` is used, the method will get or set the alignment of the label within the widget. When the first input is the word `"expand`, the expand property of the field will be set or get, and a boolean value will be required. When setting the value, the third input must be the word : `"left "right` or `"center`.

label

```
$entry~label word word (string | word)
```

Set or get the configuration of the label. The first input must be the word `"align` or `"text`. When `"align` is used, the method will get or set the alignment of the entry field within the widget. If the first input is the word `"text`, the label string will be set or get. The third input when setting the value must be the word : `"left "right` or `"center` when working on the alignment, otherwise it must be a string.

3.9.3 Configuration

Configuration	Purpose	Value
`"value`	Set or get the value of the entry field	a string or word
`"variable`	Set or get the linked variable of the widget	a word

Table 3.15: Entry's configuration

3.9.4 Hooks

Name	Description	Function prototype
"changed	The string in the entry field have been modified	to changed :src :old :new ; src is the widget object ; old is the old string ; new is the new string entered by the user end

Table 3.16: Entry's hooks

3.9.5 Example

Example 14

```
make "win Window "titled 'Entry' [100 100]
make "f Frame
make "name Entry 'What\'s your name ?' "friend
Hook :name "changed {
        Info "none ["Hello] '' 'Hello' :friend '!'
}
Glue :f "top [] :name
Glue :win "top [] :f
$win~show
```

When the user hits the *Enter* key after an update in the entry field, the hook `changed` will be called.

Figure 3.16: A Entry widget

The next example play with the linked variable :

Example 15

```
to IncrTime :src :s :t
        if :s {
```

```
                    make "Time string (parse.number :Time) + :t
          } {
                    make "Time string (parse.number :Time) - :t
          }
end

make "win Window "titled 'Entry' [100 100]
make "box Box 'Test to do'
$box~config "expand.x "set true
make "c1 CheckBox 'Computer' "computer
make "c2 CheckBox 'AC' "ac
make "c3 CheckBox 'Power' "power
Hook :c1 "invoked "IncrTime 10
Hook :c2 "invoked "IncrTime 20
Hook :c3 "invoked "IncrTime 30
Glue :box "top [] :c1 :c2 :c3
make "f Frame
make "Time '0'
make "name Entry 'Time :' "Time [0 4]
Glue :f "top [] :name
Glue :win "top [] :box :f
$win~show
```

The function `IncrTime` is called each time one of the CheckBoxes is checked or unchecked. This function modifies the variable which is always a string.

Figure 3.17: A Entry widget updated by its linked variable

3.10 The widget Frame

A ”Frame” is a container widget displaying a relief border. This border could be raised, lowered or flattened.

3.10.1 Construction

The primitive `Frame` is used to build a new Frame widget. Its syntax is :

`Entry` *word (list (words))*

The first input is the relief style of the frame, and it must be a valid word : `"flattened "raised "bordered` or `"lowered`. The second input, if specified, is the size of the widget, consisting of a list of two elements (width height). Any other inputs make up the flags.

3.10.2 Methods

A Frame widget has three methods :

reglue

```
$frame~reglue
```

This primitive asks the *geometry manager* to glue all the widgets within the Frame a second time. This primitive is useful when a child of the Frame has been removed and new gluing is required.

relief

```
$frame~relief word (word)
```

Set or get (according to the value of the first input : `"set` or `"get`) the relief of the border. When setting a value, the second input must be one of the valid words : `"lowered "flattened "bordered` or `"raised`.

widgets

```
$frame~wigets
```

Output all the widgets glued on the Frame.

3.10.3 Configuration

3.10.4 Hooks

This widget doesn’t have any hooks.

Configuration	Purpose	Value
`"level`	Set or get the level of the relief	a number

Table 3.17: Frame's configuration

3.10.5 Example

Example 16

```
1  make "win Window "titled 'Frame' [100 100]
2  make "f Frame "bordered [70 70]
3  $f~config "level "set 5
4  Glue :win "top [] :f
5  $win~show
```

On line 3, we set the level of relief of the border to 5.

Figure 3.18: A Frame widget with a bordered relief

Example 17

```
1  make "win Window "titled 'Frame' [100 100]
2  make "f Frame "raised [70 70]
3  $f~config "level "set 2
4  Glue :win "top [] :f
5  $win~show
```

On line 3, we set the level of the relief of the border to 5.

Example 18

```
1  make "win Window "titled 'Frame' [100 100]
2  make "f Frame "lowered [70 70]
3  Glue :win "top [] :f
4  $win~show
```

A lower relief gives a nice sunken feature to the frame.

Example 19

Figure 3.19: A Frame widget with a raised relief

Figure 3.20: A Frame widget with a lowered relief

```
make "win Window "titled 'Frame' [100 100]
make "f Frame "flattened [70 70]
Glue :win "top [] :f
$win~show
```

Figure 3.21: A Frame widget without a border

3.11 The widgets MenuBar and Menu

This widget displays a pull down list of menu items. Once filled with menu items, the menu could be glued anywhere in a container (window or widget). The widget MenuBar is a container widget which accepts only Menu widgets.

3.11.1 Construction

The primitive `MenuBar` is used to build a new MenuBar widget. Its syntax is :

`MenuBar` *(word)*

If specified, the first and only input of the primitive must be the word : `"column` or `row`. This is the layout of the menu in the MenuBar. By default the layout is in columns.

The primitive `Menu` is used to create a new Menu widget. Its syntax is :

`Menu` *word* `|` *string* `|` *image*

The first input is the label of the menu. In an *Image* is specifed of it will be displayed instead of a text label.

3.11.2 Methods

MenuBar

A MenuBar widget has three methods :

add

```
$menubar~add Menu (Menu)*
```

Add a menu (or several) on the MenuBar.

find

```
$menubar~find word | string
```

The Menu widget output has the label as input to the primitive. If no Menu matches, -1 is returned by the method.

remove

```
$menubar~remove Menu
```

Remove from the MenuBar a Menu.

Menu

Like a MenuBar, a Menu had a few methods:

add

```
$menu~add menu | (list | word | string) ((word things...) | block)
```

Add an item to a menu. An item could be another Menu then when added, this Menu will be a submenu. When using a string, word or a list, a second input could be specified. It could be either the name of a function (and then some input to pass on to this function) or a block. This will be the function or the block executed when the menu item is invoked by the user. The method output the index of the new item in the menu. If the first input is a list, it can specify the label to display as first element of the list, then the shortcut (as a string) and then if specified the modifiers to add to `ALT` for the shortcut. Modifiers can be `"alt`, `"shift`, `"control` and `"option`. When the item is invoked, the callback function will run on a separate thread.

find

```
$menu~find word | string
```

Find a menu item or submenu in the Menu. If the item is found, its position in the Menu is returned by the method. If it's a submenu, the Menu widget is returned. When nothing is found, the method returns -1.

i.enable

```
$menu~i.enable number boolean
```

Enable or disable an item of the menu. The first input is the index of the item in the menu. The second input is `true` or `false`.

i.font

```
$menu~i.font number font
```

Set the font used by a menu item.

i.mark

```
$menu~i.mark number boolean
```

Mark or unmark an item of the menu. The first input is the index of the item in the menu. The second input is `true` or `false`.

remove

```
$menu~remove string | word | Menu
```

Remove an item (simple or submenu) from the Menu.

3.11.3 Configuration

Although a MenuBar doesn't have any specific configuration, a Menu widget has only one specific configuration but it doesn't support the usual widget configuration :

Configuration	Purpose	Value
`"radio`	Set or get the radio mode of the menu	a boolean

Table 3.18: Menu's configuration

3.11.4 Hooks

A MenuBar and a Menu widget don't have any specific hooks. The Menu widget doesn't even have the standard widget hooks.

3.11.5 Example

Example 20

```
1  make "win Window "titled 'Menu' [100 100]
2  make "f Frame "flattened [100 100]
3  make "menu MenuBar
4  $menu~config "expand.x "set true
5  make "file Menu "File
6  $file~add "Load
7  $file~add "Save
8  $file~add "separator
9  $file~add "Quit {
10         $win~quit
11 }
12 make "option Menu "Option
13 $option~config "radio "set true
14 $option~add 'BeOS style'
15 $option~add 'Dos style'
16 make "question Menu '?'
17 $question~add 'Help'
18 $question~add "separator
19 $question~add 'About ...'
20 $menu~add :file :option :question
21 Glue :win "top [] :menu :f
22 $win~show
```

At line 8 of this example, we are creating an item `"separator` in the menu `file`. This word `"separator` is a reserved word which creates a separator item in the menu, like shown below :

In the next example, we will create a submenu :

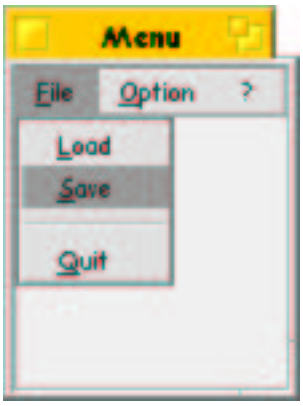


Figure 3.22: A simple MenuBar with Menu

Example 21

```
1  make "win Window "titled 'Menu' [100 100]
2  make "f Frame "flattened [100 100]
3  make "menu MenuBar
4  $menu~config "expand.x "set true
5  make "file Menu "File
6  $file~add "Load
7  $file~add "Save
8  make "export Menu "Export
9  $export~add 'to dos'
10 $export~add 'to mac'
11 $file~add :export
12 $file~add "separator
13 $file~add "Quit {
14         $win~quit
15 }
16 make "option Menu "Option
17 $option~config "radio "set true
18 $option~add 'BeOS style'
19 $option~add 'Dos style'
20 make "question Menu '?'
21 $question~add 'Help'
22 $question~add "separator
23 $question~add 'About ...'
24 $menu~add :file :option :question
25 Glue :win "top [] :menu :f
26 $win~show
```

We create and set the submenu in lines 8 to 14.

Figure 3.23: Menu width submenu

3.12 The widget Memo

This widget is a multi-line entry widget that can display text.

3.12.1 Construction

The primitive `Memo` is used to build a new Memo widget. Its syntax is :

`Memo` *list*

The only input is the size of the widget specified in characters : `[width height]`.

3.12.2 Methods

allow

```
$memo~allow (word | string)+
```

The user will be allowed to enter the characters given as inputs.

allow.all

```
$memo~allow.all
```

The user will be allowed to enter any characters.

delete

```
$memo~delete (integer integer | [integer integer])
```

If no input is given the method will delete the text currently selected by the user. If two integers are given, or a list of two integers, they will be the offset of the text to delete.

disallow

```
$memo~disallow (word | string)+
```

The user will not be allowed to enter the characters given as inputs.

disallow.all

```
$memo~disallow.all
```

The user will not be allowed to enter any characters.

insert

```
$memo~insert (thing)*
```

Insert all the inputs as the current position in the widget.

line

```
$memo~insert (thing)*
```

Insert all the inputs as the current position in the widget.

load

```
$memo~load string
```

Load the text file which the path is given as input in the widget.

save

```
$memo~save string
```

Save in the text file which the path is given as input, the content of the widget.

selection

```
$memo~selection word ([integer integer] | integer integer)
```

Set or get the selection of part of the content of the widget. The first input is the word `"get` or `set`. When using `"set`, the text selection is changed in the widget. The rest of the inputs, a list or 2 numbers specify the offsets of the selections to do. When `"get` is used, the primitive output a list of the offset of the current text selected.

text

```
$memo~text word ([integer integer] | integer integer | string)
```

Set or get the text in the widget. The first inputs is the word `"get` or `"set`. When using `"set`, the second input must be a string. When using `"get`, the primitive output the content of the widget, or a part of the content when a third (and forth if any) input is given. The two integers are the offsets of the part of the text we want to get.

3.12.3 Configuration

A Memo widget has four specific configuration items :

3.12.4 Hooks

This widget don't have any added hooks.

Configuration	Purpose	Value
`"alignment`	Set or get the alignment of content of the widget	`"left "center "right`
`"bgcolor`	Set or get the color of content background	a list describing a color
`"color`	Set or get the color of the widget content	a list describing a color
`"indent`	Set or get the automatic indent of the content	`true` or `false`
`"warp`	Set or get if the content of the widget must be warped when the line are too long	a word

Table 3.19: Memo's configuration

3.12.5 Example

Example 22

```
1  Font.init
2  make "font Font 'Squirrel'
3  $font~size "set 15.4
4
5  make "MyWin Window "titled 'Memo widget' [100 100]
6  make "text Memo [20 10]
7
8  $text~text "set 'this is a text\nwith severals lines\nin it!'
9  $text~config "font "set :font
10 $text~config "wrap "set false
11 $text~config "color "set :Red
12 $text~config "expand "set (true) (true)
13
14 Glue :MyWin "top [] :text
15 $MyWin~show
```

Lines 1 to 3 define a font (Family and size). We set the font as well as the text displayed by the widget on lines 8 and 9.

Figure 3.24: Memo example

3.13 The widget Odometer

This widget is part of the *Widgets* Add-On and it display a number in a Odo-Meter kind of look.

3.13.1 Construction

The primitive `Odometer` is used to build a new Odometer widget. Its syntax is:

`Odometer number`

The only input is the number of digits the widget shall display at the maximum.

3.13.2 Methods

The Viewer widget has two specific methods:

display

```
$odometer~display number
```

Display the number given as input.If the number is a floting point value, a dot will be displayed as well.

precision

```
$odometer~precison(integer)
```

if no input is given, the method output the floating point precision used. If a number is given as input, the method set the precision of the floating point value displayed.

3.13.3 Configuration

This widget don't have any specific configuration.

3.13.4 Hooks

This widget don't have specific hooks.

3.13.5 Example

Example 23

```
use 'GUI' 'Widgets'

make "win Window "titled 'Odometer' [100 100]
make "odo Odometer 11
Glue :win "top [] :odo
$win~show

$odo~precision 3
$odo~display -551.6543
```

Figure 3.25: The Odometer widget

3.14 The widget RadioButton

This widget is a labeled two state button which is often used with several other similar widgets. Only one of these widgets within the same container (same group) could be *on* at a time.

All the RadioButtons of the same group share the same linked variable.

3.14.1 Construction

The primitive `RadioButton` is used to build a new RadioButton widget. Its syntax is :

`RadioButton` *word* | *string word thing (list (words))*

The first input (either a word or a string) is the label to be displayed by the widget. The second input is the name of the linked variable. If the variable doesn't already exist, it will be created. The third input is the value to be given to the variable when the widget is clicked by the user. The following inputs, if specified, will be the size in characters (two integers) and the flags of the widget.

3.14.2 Methods

invoke

`$radiobutton~invoke`

Invoke the widget as though the widget has been clicked.

3.14.3 Configuration

A RadioButton widget has three specific configuration items :

Configuration	Purpose	Value
`"label`	Set or get the label of the widget	could be a word or a string.
`"value`	Set or get the state of the widget	`true` or `false`
`"variable`	Set or get the linked variable of the widget	a word

Table 3.20: RadioButton's configuration

3.14.4 Hooks

Name	Description	Function prototype
"invoked	The widget has been clicked	to invoked :src :old :new ; src is the widget object ; old is the previous value ; new is the new value end

Table 3.21: RadioButton's hooks

3.14.5 Example

Example 24

```
1  to linux
2          Info "warning ['ok'] '' 'Rebooting under Linux now'
3  end
4
5  to beos
6          Info "warning ['ok'] '' 'Rebooting under BeOS now'
7  end
8
9  to os2
10         Info "warning ['ok'] '' 'Rebooting under OS/2 now'
11 end
12
13 to windows
14         Info "warning ['ok'] '' 'Rebooting under Windows now'
15 end
16
17 make "win Window "titled 'RadioButton' [100 100]
18 make "os "beos
19 make "r1 RadioButton 'Linux' "os "linux
20 make "r2 RadioButton 'BeOS' "os "beos
21 make "r3 RadioButton 'OS/2' "os "os2
22 make "r4 RadioButton 'Windows' "os "windows
23 $r1~config "expand.x "set true
24 $r2~config "expand.x "set true
25 $r3~config "expand.x "set true
26 $r4~config "expand.x "set true
27 make "b Button 'Reboot now'
28 Hook :b "invoked {
29         call :os
30 }
31 Glue :win "top [] :r1 :r2 :r3 :r4 :b
32 $win~show
```

This example gives the user the possibility to reboot his system under the desired operating system. On line 29, we use the primitive `call` to execute the function corresponding to the OS selected by the user.

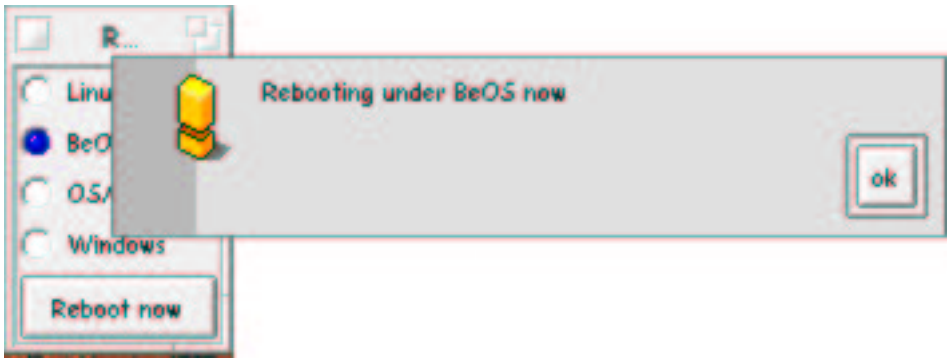


Figure 3.26: RadioButton example

3.15 The widget SimpleList

A SimpleList widget displays a list of simple items that the user can select and invoke. The items could be any kind of simple object like : a word, a string or a number. The widget has a linked variable.

3.15.1 Construction

The primitive `SimpleList` is used to build a new SimpleList widget. Its syntax is :

`SimpleList` *word word word list (list (words))*

The first input is the type of the SimpleList : `"single` or `"multiple`. When the list is of type `"multiple` the user will be able to select several items and the linked variable will be set to a list of items. The type `"single` only allows one item to be selected. The second input is the layout of the scrollbar in the widget : `"left` or `"right`. The third input is the name of the linked variable. The fourth input is a list of items. If specified, a fifth input will be the size of the widget in characters and any other inputs would make up the flags.

The size of the widget could be specified with two numbers. The first is always the width of the widget (without the size of the scrollbar) and the second is the number of items displayed at one time by the widget.

3.15.2 Methods

add

```
$simplelist~add (thing | list)+
```

Add items to the end of the list.

add.at

```
$simplelist~add.at number (thing | list)+
```

Add items at a position in the list. The first input is the position. 0 is the first element of the list.

items

```
$simplelist~items word (list)
```

If the first input is the word `"get` the method will return the list of items. If the first input is `"set` the second input of the method must be a list of items. This list will replace the items.

3.15.3 Configuration

The widget doesn't have any specific configuration.

3.15.4 Hooks

Name	Description	Function prototype
invoked	The user has double clicked on an item	`to invoked :src :index :value ; src is the widget object ; index is the position of the invoked item in the list ; value is the value of the invoked item in the list end`
selected	The user has selected a new item	`to invoked :src :index :value ; src is the widget object ; index is the position of the selected item(s) in the list (could be a list if multiple selection) ; value is the value of the selected item(s) in the list (could be a list if multiple selec- tion) end`

Table 3.22: SimpleList's hooks

3.15.5 Example

Example 25

```
1  make "items gseq 1990 2000
2  make "year 1998
3  make "win Window "titled 'SimpleList' [100 100]
4  make "list SimpleList "single "right "year :items [0 5]
5  $list~config "expand.x "set true
6  make "button Button 'Process'
7  Hook :button "invoked {
8          Info "info ["ok] '' 'Processing data of' :year ' ....'
9  }
10 Glue :win "top [] :list :button
11 $win~show
```

This simple example shows how the linked variable is updated when the user changes the selection. On line 4, we have set the height of the widget to 5, meaning that we want only 5 items to be displayed.

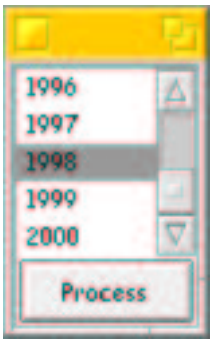


Figure 3.27: Single selection in a SimpleList

The next example shows a multiple selection. Using the *Shift* key of the keyboard allows us to make a multiple selection. The *Option* key terminates a selection.

Example 26

```
make "friends ["Fred "Roger "Ben "Zack "Tom]
make "friend []
make "win Window "titled 'SimpleList' [100 100]
make "list SimpleList "multiple "right "friend :friends [0 5]
$list~config "expand.x "set true
make "button Button 'Send'
Hook :button "invoked {
        Info "info ["ok] '' 'Sending data to :' :friend
}
Glue :win "top [] :list :button
$win~show
```

Figure 3.28: Multiple selection in a SimpleList

3.16 The widget StatusBar

This widget displays a progress bar, that indicate the progression and pace of a certain task.

3.16.1 Construction

The primitive `StatusBar` is used to build a new StatusBar widget. Its syntax is :

`StatusBar` *list integer (string (string))*

The first input is a list that gives the width (in characters) that the widget must allocate for the text and trailing text that will be later updated. The second input is the maximum value that the widget can reach. If given, the third input will be a string to display as the label, and fourth input will be the trailing label to display.

3.16.2 Methods

The StatusBar widget has two specific methods:

reset

```
$banner~reset (string (string))
```

Reset the status bar to 0. If given, the second input will be the text and a third input will be the trailing text.

update

```
$banner~update integer (string (string))
```

Update the status bar by adding the first input to the current value. If given, the second input will be the text and a third input will be the trailing text.

3.16.3 Configuration

Configuration	Purpose	Value
"bar.color	Set or get the color of the progress bar	a color list
"max	Set or get the maximum value of the status bar	an integer
"text	Set or Get the current text displayed	a string or word
"trailing	Set or Get the current trailing displayed	a string or word
"value	Set or Get the current value	an integer

Table 3.23: StatusBar's configuration

3.16.4 Hooks

This widget don't have specific hooks.

3.16.5 Example

Example 27

```
1  make "MyWin Window "titled 'StatusBar' [100 100] "not.closable
2  make "frame Frame "flatened
3  make "status StatusBar [0 2] 10 'Processing' ' items remaining'
4  $status~config "expand "set (true) (true)
5
6  Glue :frame "top [2 2] :status
7  Glue :MyWin "top [] :frame
8  $MyWin~show
9
10 for ["i 1 10] {
11         $status~update 1 '' string (10-:i)
12         wait 1
13 }
14 $MyWin~quit
```

On line 13, we update the status bar in a loop. Each iteration we change the trailing text to display the remaining items to process.

Figure 3.29: A StatusBar example

3.17 The widget Text

This widget displays a static string. It's a simpler version of the widget Banner. There's no linked variable to this widget, but modification of the displayed string is always possible.

3.17.1 Construction

The primitive `Text` is used to build a new Text widget. Its syntax is :

`Text` *word* | *string (list (words))*

The first input (either a word or a string) is the text to be displayed. The second input, if specified, is a list which indicates the size of the widget in characters. This list has two elements : width and height. An empty list will be the same as no list. All the inputs to the left are the flags, which may be specified. The primitive outputs the widget object.

3.17.2 Methods

The Text widget has a few methods uncommon to all the widgets :

justify

```
$banner~justify word (word)
```

Set or get (according to the value of the first input : `"set` or `"get`) the text justification in the widget. When setting a value, the second input must be one of the valid words : `"left` `"center` or `"right`.

text

```
$banner~text word (string | word)
```

Set or get (according to the value of the first input : `"set` or `"get`) the text displayed by the widget. The second input could be a string or a word.

3.17.3 Configuration

This widget doesn't have any specific configuration.

3.17.4 Hooks

This widget has no more hooks than the common widget.

3.17.5 Example

Example 28

```
make "win Window "titled 'SimpleList' [100 100]
make "label Text 'Hello world!'
Glue :win "top [] :label
$win~show
```

Figure 3.30: A simple Text example

3.18 The widget Viewer

This widget is part of the *Imaging* Add-On and it displays an image.

3.18.1 Construction

The primitive `Viewer` is used to build a new Viewer widget. Its syntax is:

```
Viewer image | list
```

The only input can be either an *Image* object or a list. The list specify the size in pixel of the widget. If an image is given, the widget will adapt it size to fit the full image.

3.18.2 Methods

The Viewer widget has two specific methods:

display

```
$viewer~display image (word)
```

Display the image given as first input. If a second input is specified, it must be one of the following words : `"adapt` `"center` `"scale` `"scroll`. This second input give the way the widget shall display the image. By default it is `"adapt`. The following table explain the different style of display:

Style	Purpose
`"adapt`	Adapt the size of the widget to fit the complete image. Resize the window to fit.
`"center`	Display the image without resizing the widget. The image is centered on the center of the widget.
`"scale`	Display the image scaled to fit in the widget.
`"scroll`	Keep the current widget size, display scrollbars for the user to see the image.

Table 3.24: Viewer's display styles

resize.to

```
$viewer~resize.to list
```

Resize the widget to a given size. The method has for effect to ask the window where the widget is to resize as well.

3.18.3 Configuration

This widget don't have any specific configuration.

3.18.4 Hooks

This widget don't have specific hooks.

3.18.5 Example

Example 29

```
use 'List Processing'

if (llength :Args) = 2 {

        use 'GUI' 'Imaging'

        make "my.image Image lindex :Args 2
        if (is.image :my.image) {
        make "win Window "titled (lindex :Args 2) [100 100] "not.resizable
                make "the.viewer Viewer :my.image
                Glue :win "top [] :the.viewer
                $win~show
        } {
                print 'Image file not reconized ... maybe not an image'
        }
} {
        print 'USAGE : image'
}
```

Figure 3.31: The Viewer widget

Chapter 4

Supports

This Add-on give access to several other primitives as well as some new objects used by the widgets.

4.1 Fonts

Fonts management under SQUIRREL is done by using a set of primitive and a new object: Font.

4.1.1 Primitives

Font.init

```
Font.init
```

This primitive in mandatory in SQUIRREL to use with fonts. When fonts are used, the primitive should always be one of the first things a script performs.

Font.families

```
Font.families
```

Output a list of all the font families installed on the computer.

Font.exists

```
Font.exists string
```

Output `true` if a font family given as input to the primitive is installed on the system.

Font.styles

```
Font.styles string
```

Output a list of all the styles available to a font family given as input to the primitive.

4.1.2 Font object

The primitive `Font` creates a new font object for a specified font family. During its lifetime, a font object can change family. The syntax of this primitive is :

`Font` *string*

A Font object has several methods:

aliasing

```
$font~aliasing word (word)
```

Get or set whether the font is using anti-aliasing or not. The first input must be the word `"get` or `"set`. The second input must be the word `"on` or `"off`.

direction

```
$font~direction word (word)
```

Get or set if the font has direction. The first input must be the word `"get` or `"set`. The second input must be the word `"left2right` or `"right2left`.

encoding

```
$font~encoding word (number)
```

Get or set if the font is encoding. The first input must be the word `"get` or `"set`. The second input must be a number from the following table :

UNICODE_UTF8	0
ISO_8859_1	1
ISO_8859_2	2
ISO_8859_3	3
ISO_8859_4	4
ISO_8859_5	5
ISO_8859_6	6
ISO_8859_7	7
ISO_8859_8	8
ISO_8859_9	9
ISO_8859_10	10
MACINTOSH_ROMAN	11

Table 4.1: Font encoding

family

`$font~family word (string)`

Get or set if it's the family of the font object. The first input must be the word `"get` or `"set`. The second input (if any) is the name of the family.

rotation

`$font~rotation word (number)`

Get or set if there's rotation to the font object. The first input must be the word `"get` or `"set`. The second input (if any) is a number between 0 and 360.

shear

`$font~shear word (number)`

Get or set if there's shear to the font object. The first input must be the word `"get` or `"set`. The second input (if any) is a number between 45 and 135.

size

`$font~size word (number)`

Get or set if there's size to the font object. The first input must be the word `"get` or `"set`. The second input (if any) is a number.

spacing

`$font~spacing word (word)`

Get or set if there's spacing to the font object. The first input must be the word `"get` or `"set`. The second input (if any) must be one of the valid word : `"char "string "bitmap "fixed`.

style

`$font~style word (word)`

Get or set if there's style to the font object. The first input must be the word `"get` or `"set`. Style varies from font but it's usually the words or strings : `"Regular "Roman "Bold 'Bold Italic'` or `"Italic`.

4.1.3 A Little example

The next example shows how simple it is to build a Font browser. The window is composed of two SimpleLists, one for the families and one for the style. A Text widget, whose font is changed each time the user selects a font or a style, displays a string :

Example 1

```
Font.init

make "win Window "titled 'Fonts' [100 100]
make "frame Frame
make "Families Font.families
make "Family lindex :Families 1

make "theFont Font :Family
$theFont~style "set lindex (Font.styles :Family) 1
$theFont~size "set 15

make "fbox Box 'Family'
make "families SimpleList "single "right "Family :Families [0 6]
Hook :families "selected {
        $styles~items "set (Font.styles :Family)
        make "Style lindex (Font.styles :Family) 1
}
Glue :fbox "top [] :families
make "sbox Box 'Styles'
make "Style "Roman
make "styles SimpleList "single "right "Style (Font.styles :Family) [0 4]
Hook :styles "selected {
        $theFont~family "set :Family
        $theFont~style "set :Style
        $label~config "font "set :theFont
}
make "label Text 'Select a family and a style!'
$label~config "font "set :theFont
$label~config "bgcolor "set ($frame~config "bgcolor "get)
$label~config "expand.x "set true
$label~justify "set "center
Glue :sbox "top [] :styles
Glue :frame "left [] :fbox :sbox
```

```
Glue :win "top [] :frame :label
$win~show
```

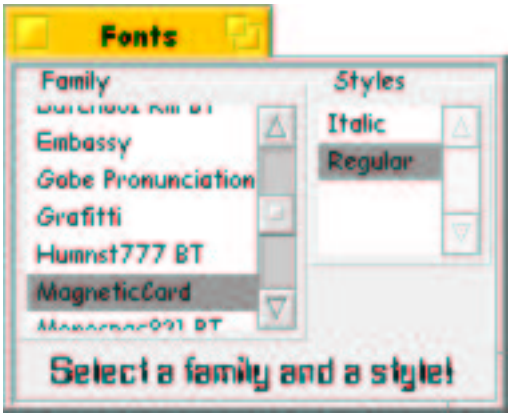


Figure 4.1: Browsing the installed font

4.2 Color List

The file `/boot/apps/Squirrel/Libraries/Colors.sqi` list several useful colors. You may load this file at the beginning of your script in order to use them. You will find several standard colors like red, black etc ... as well as the standard colors of the BeOS interface.

You will notice by looking at this file, that each color is stored in a variable :

```
make "LightBlue [64 162 255 255]
```

A color is a well known mix of the three colors : red , blue and green. In the list, these are always the first three elements. A fourth element is not mandatory. If specified, however, it's the value of the Alpha channel.

4.3 Primitives

Several primitives are added to SQUIRREL by the GUI Add-on to test objects for membership to certain types or set the focus :

Busy

```
Busy boolean (word)
```

Set or unset the application cursor to an animated *busy* cursor. The first input indicate by `true` or `false` if the cursor shall be animated or not. If a second input is given, it must be `"spinwheel` or `"watch`, it indicates the type if cursor to use. By default, `"spinwheel` is used.

Focus

```
Focus widget
```

Set the keyboard focus on a widget.

is.banner

```
is.banner thing
```

Output `true` if the input is a Banner widget, otherwise output `false`.

is.barberpole

```
is.barberpole thing
```

Output `true` if the input is a BarberPole widget, otherwise output `false`.

is.box

```
is.box thing
```

Output `true` if the input is a Box widget, otherwise output `false`.

is.button

```
is.button thing
```

Output `true` if the input is a Button widget, otherwise output `false`.

is.checkbox

```
is.checkbox thing
```

Output `true` if the input is a CheckBox widget, otherwise output `false`.

is.colorcontrol

```
is.colorcontrol thing
```

Output `true` if the input is a ColorControl widget, otherwise output `false`.

is.container

`is.container thing`

Output `true` if the input is a container widget, otherwise output `false`.

is.droplist

`is.droplist thing`

Output `true` if the input is a DropList widget, otherwise output `false`.

is.entry

`is.entry thing`

Output `true` if the input is an Entry widget, otherwise output `false`.

is.font

`is.font thing`

Output `true` if the input is a Font object, otherwise output `false`.

is.frame

`is.frame thing`

Output `true` if the input is a Frame widget, otherwise output `false`.

is.font

`is.font thing`

Output `true` if the input is a Font object, otherwise output `false`.

is.memo

`is.memo thing`

Output `true` if the input is a Memo widget, otherwise output `false`.

is.menu

`is.menubar thing`

Output `true` if the input is a Menu widget, otherwise output `false`.

is.menubar

`is.menubar thing`

Output `true` if the input is a Menubar widget, otherwise output `false`.

is.menubar

`is.menubar thing`

Output `true` if the input is a Menubar widget, otherwise output `false`.

is.odometer

`is.odometer thing`

Output `true` if the input is an Odometer widget, otherwise output `false`.

is.radiobutton

`is.radiobutton thing`

Output `true` if the input is a Radiobutton widget, otherwise output `false`.

is.statusbar

`is.statusbar thing`

Output `true` if the input is a StatusBar widget, otherwise output `false`.

is.text

`is.text thing`

Output `true` if the input is a Text widget, otherwise output `false`.

is.viewer

`is.viewer thing`

Output `true` if the input is a Viewer widget, otherwise output `false`.

is.widget

`is.widget thing`

Output `true` if the input is a widget, otherwise output `false`.

is.window

`is.window thing`

Output `true` if the input is a Menubar widget, otherwise output `false`.

Chapter 5

Release notes

5.1 Release 0.71

This release has been partialy rewrite to improve a bit the performance and lower the memory consumption. The Add-On it-self is 1 Mb lighter than the previous version.

Two new Add-Ons has been added: *Imaging* and *Widgets*.

5.1.1 Changes

- The Memo widget has been modified.
- Menu callback now run on their own thread.
- Some Widget's callback functions run on their own thread.
- Primitive `font.exist` renamed `font.exists`.
- The demo *EzCalc* has been improved.

5.1.2 Additions

- New Add-Ons *Imaging* and *Widgets*.
- New widgets *Viewer* and *Odometer*.
- Added method `invalidate` to the widgets.
- Menu widget can display an image instead of text only.
- Added primitive `Busy` that set the application cursor to an animater cursor.
- Added `cursor` configuration to the widgets. It allow to affect a cursor to a widget.
- Added method `i.font` to the Menu widget. This method set the font used by a menu item.
- Added configurations `constraint` and `limit` to the Window .

5.1.3 Bugs fixed

- Fixed a bug in the Banner widget (linked variable now created when non existent).
- Menu widget now use by default the same font than the MenuBar widget.
- Fixed a bug in the Memo widget (changing contents).

5.2 Release 0.68

5.2.1 Notes

Few evolutions in this release.

5.2.2 Changes

- The messagebox from *Supports* has been moved off this Add-On and are now accessible trought the `Communication Add-On`.

5.2.3 Additions

- Widget's Hooks `drop` that is called when the widget is the target of a drag and drop.

5.2.4 Bugs fixed

None.

5.3 Release 0.67

5.3.1 Notes

Maintenance release.

5.3.2 Changes

None.

5.3.3 Additions

None.

5.3.4 Bugs fixed

- Debug trace when using keydown in BarberPole widget (removed)
- Crash on quitting window by erasing the variable holding the window object
- Crash when a `SimpleList` is destroyed
- Crash when the extra input of a hook function was created within a function

5.4 Release 0.64

5.4.1 Notes

One new widget in this release and some ehencement in the key hooks.

5.4.2 Changes

- Widget's Hooks : `keydown keyup` and `mousedown` requiert a new input that give the key modifiers used when the event occurs.
- Menu accept shortcut and modifiers.

5.4.3 Additions

- Widget `BarberPole` that display text and allow the user to enter text.

5.4.4 Bugs fixed

None.

5.5 Release 0.60

5.5.1 Notes

Two new widgets added in this release and severals bugs fixed.

5.5.2 Changes

- The widget `Entry` has been changed to respect the type of the value in it linked variable.

5.5.3 Additions

- Widget `Memo` that display text and allow the user to enter text.
- Widget `StatusBar` that display a progress bar that can be updated.
- Methods `resize.to resize.by center` added to the Window.

5.5.4 Bugs fixed

- A possible problem with the widget's hooks
- Crash of SQUIRREL when the value of the linked variable of a `DropList` widget is not found in the list.

5.6 Release 0.54

5.6.1 Notes

This release is a maintenance release with few bugs fixed.

5.6.2 Changes

None.

5.6.3 Additions

None.

5.6.4 Bugs fixed

- Using the `invoke` method of a widget within a hook wasn't working
- Using an unknow linked variable with a RadioButton was crashing SQUIRREL

5.7 Release 0.49

5.7.1 Notes

This release is a maintenance release has it fixe moslty a few bugs.

5.7.2 Changes

No changes.

5.7.3 Additions

- new method `i.enable` to the `Menu` widget to enable or disable a menu item.
- new method `i.mark` to the `Menu` widget to mark or unmark a menu item.

5.7.4 Bugs fixed

- Mimimum size of a `Box` widget now set fit the label
- Updating the linked variable of a `CheckBox` is now working correctly
- Change inf the configuration of a widget glued is now working fine properly (*Looper must be looker* error)

5.8 Release 0.46

5.8.1 Notes

About this release

This is the first release of the *new* GUI Add-on for SQUIRREL . This version has been completely rewritten from the old versions of SQUIRREL DR2 and DR3.

The Add-on

Although the Add-on has been tested with several examples that one could find in the SQUIRREL directory, it's still an early and incomplete version :

Several features or widgets are missing in this release:

- A canvas widget allowing to draw within a widget.
- Access to the image files (BBitmap and BPicture)
- Drag & Drop
- More complete set of widgets (all kind)
- BScrollView, BTextView, BStatusBar ...
- Printing

5.8.2 Changes

No change.

5.8.3 Additions

No addition.

5.8.4 Bugs fixed

No bug fixed yet :(

Index