# Prima - the perl graphic toolkit

Dmitry Karasik

August 19, 2024

# Contents

# 1 Introduction

### Preface

Prima is an extensible Perl toolkit for multi-platform GUI development. Platforms supported include Linux, Windows, and UNIX/X11 workstations (FreeBSD, IRIX, SunOS, Solaris, and others). The toolkit contains a rich set of standard widgets and has an emphasis on 2D image processing tasks. A Perl program using Prima looks and behaves identically on X11 and Win32.

The Prima project was started in 1997 in Protein Laboratory, Copenhagen, by Anton Berezin, Dmitry Karasik, and Vadim Belman.

This document describes the programming with Prima graphic toolkit and is a collection of manual pages of the Prima application program interface ( API ).

### Requirements

Prima supports perl versions 5.12 and above. The recommended perl versions are 5.20 and above. In the Unix environments, Prima can use the following graphic libraries: libjpeg, libgif, libtiff, libpng, libXpm, libwebp, and libheif.

### Installation

The toolkit can be downloaded from `http://www.prima.eu.org` in source and binary forms. Before installing, check the content of the README file in the distribution. The installation from the source is performed by executing commands

```
perl Makefile.PL
make
make test
make install
```

### Authors

Dmitry Karasik, Anton Berezin, Vadim Belman

### Credits

David Scott, Kai Fiebach, Johannes Blankenstein, Teo Sankaro, Mike Castle, H.Merijn Brand, Richard Morgan, Kevin Ryde, Chris Marshall, Slaven Rezic, Waldemar Biernacki, Andreas Hernitscheck, David Mertens, Teo Sankaro, Gabor Szabo, Fabio D'Alfonso, Rob "Sisyphus", Chris Marshall, Reini Urban, Nadim Khemir, Vikas N Kumar, Upasana Shukla, Sergey Romanov, Mathieu Arnold, Petr Pisar, Judy Hawkins, Myra Nelson, Sean Healy, Ali Yassen, Maximilian Lika, kmx, Mario Roy, Timothy Witham, Mohammad S Anwar, Jean-Damien Durand, Zsban Ambrus, Max Maischein, Reinier Maliepaard

– thank you for your help.

# 2 Tutorials

## 2.1 Prima::tutorial

Introductory tutorial

### Description

Programming of the graphic interfaces is often considered a somewhat boring business, and not without a cause. There is little pride in knowing that your buttons and scrollbars work exactly as millions of other buttons and scrollbars do, so whichever GUI toolkit is chosen, it is usually regarded as a tool of small importance, and the less it is obtrusive, the better. Given that, and trying to live up to the famous Perl 'making easy things easy and hard things possible' mantra, this manual page is an introductory tutorial meant to show how to write the easy things easy. The hard things are explained in the other Prima manual pages ( see the *Prima* section ).

### Introduction - a "Hello world" program

Prima is written and is expected to be used in some traditions of Perl coding, such as DWIM ( do what I mean ) or TMTOWTDI ( there is more than one way to do it). Perl itself is the language (arguably) most effective in small programs, as the programmer doesn't need to include lines and lines of prerequisite code before even getting to the problem itself. Prima can't compete with that, but the introductory fee is low; a minimal working 'Hello world' can be written in just three lines of code:

```
use Prima qw(Application);
Prima::MainWindow-> new( text => 'Hello world!');
run Prima;
```


Line 1 is the invocation of modules *Prima* and *Prima::Application*. One can also explicitly invoke both `use Prima` and `use Prima::Application`, but since the module Prima doesn't export any method names, the syntax in the code example above allows one to write programs in a more concise style.

Line 2 creates a new window object, and instance of the *Prima::MainWindow* class, which is visualized as a window rectangle on the screen, with the title 'Hello world'. The class terminates the application (and the program) when the window is closed; this is the only difference from the

windows that are objects instances of the `Prima::Window` class, which do nothing after they are closed by the user.

( Note: In this tutorial the `Prima::` prefix in class names will be omitted and will be used only when necessary, such as in code examples ).

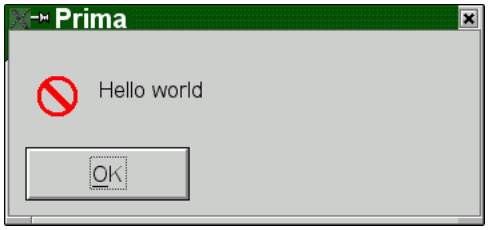
Line 3 enters the Prima event loop. The loop is terminated when the only instance of the *Application* class (that is created by the `use Prima::Application` invocation) and stored in `$::application` scalar, is destroyed.

Strictly speaking, a minimal 'hello world' program can be written even in two lines:

```
use Prima;
Prima::message('Hello world');
```

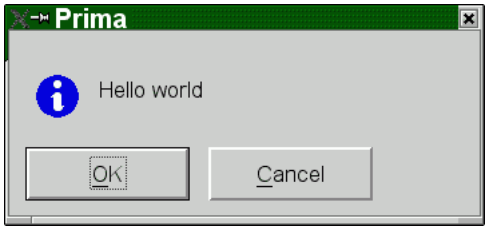

but it is not very illustrative and not useful. The `Prima::message` call is rarely used and is one of the few methods contained in the `Prima::` namespace. To display a message, the *MsgBox* module is often preferred, with its control over text in the buttons and with its appropriate usage of some pre-defined icons. If using this module instead the code above can be rewritten as this:

```
use Prima qw(Application MsgBox);
message('Hello world');
```




but where `Prima::message` accepts only text scalar parameters, `Prima::MsgBox::message` can do a lot more. For example the code

```
use Prima qw(Application MsgBox);
message('Hello world', mb::OkCancel|mb::Information);
```




displays two buttons and an icon. A small achievement, but the following code is a bit more interesting:

```
use Prima qw(Application MsgBox);
message('Hello world', mb::OkCancel|mb::Information,
        buttons => {
                mb::Cancel => {
                        # there are predefined color constants to use
                        backColor => cl::LightGreen,
```

```
                        # but RGB integers are also o.k.
                        color     => 0xFFFFFF,
                },
                mb::Ok => {
                        text    => 'Indeed',
                },
        }
);
```


Defining many object properties at once is a major feature of Prima and is seen throughout the toolkit. Returning to the very first example we can demonstrate the manipulation of the window properties in the same fashion:

```
use Prima qw(Application);
my $window = Prima::MainWindow-> new(
        text => 'Hello world!',
        backColor => cl::Yellow,
        size => [ 200, 200],
);
run Prima;
```

Note that the `size` property is a two-integer array, and the color constant is registered in the `cl::` namespace. In Prima, there are many similar two- and three-letter namespaces containing (usually integer) constants for various purposes. The design reason for choosing such syntax over the string constants ( as f ex in Perl-Tk, such as `color => 'yellow'` ) is that the syntax is checked on the compilation stage, thus narrowing the possibility of a bug.

There are over a hundred properties, such as color, text, or size, declared on descendants of the *Widget* class. These can be set in a `new` ( alias `create` ) call, or changed later, either individually

```
$window-> size( 300, 150);
```

or in a group

```
$window-> set(
        text => 'Hello again',
        color => cl::Black,
);
```

In addition to these, there are also more than 30 events called whenever a certain action is performed; the events' syntax is identical to the properties' syntax.

Now, back to the code. Here, if we change it again, we can now catch a mouse click on the window:

```
use Prima qw(Application MsgBox);
my $window = Prima::MainWindow-> new(
        text => 'Hello world!',
        size => [ 200, 200],
        onMouseDown => sub {
```

```
                my ( $self, $button, $mod, $x, $y) = @_;
                message("Aww! You've clicked me right in $x:$y!");
        },
);
run Prima;
```

While an interesting concept, it is not really practical if the only thing you want is to catch a click, and this is the part where the standard button widget should probably be used instead:

```
use Prima qw(Application Buttons MsgBox);
my $window = Prima::MainWindow-> new(
        text     => 'Hello world!',
        size     => [ 200, 200],
);
$window-> insert( Button =>
        text     => 'Click me',
        growMode => gm::Center,
        onClick  => sub { message("Hello!") }
);
run Prima;
```


For those who know Perl-Tk and prefer its ways of positioning a widget, Prima provides the *pack* and *place* interfaces. Here one can replace the line

```
growMode => gm::Center,
```

with this line:

```
pack     => { expand => 1 },
```

where both produce the same effect.

## Overview of the widget classes

Prima contains a set of standard ( in GUI terms ) widgets, such as buttons, input lines, list boxes, scroll bars, etc. These are diluted with the other more exotic widgets, such as the POD viewer or docking windows. Technically, these are collected in `Prima/*.pm` modules and each contains its own manual page, but for informational reasons here is the full table of the widget modules, an excerpt from the `Prima` manpage:

the *Prima::Buttons* section - buttons and button grouping widgets

the *Prima::Calendar* section - calendar widget

the *Prima::ComboBox* section - combo box widget

the *Prima::DetailedList* section - multi-column list viewer with a controlling header widget

the *Prima::DetailedOutline* section - a multi-column outline viewer with controlling header widget

the *Prima::DockManager* section - advanced dockable widgets

the *Prima::Docks* section - dockable widgets
the *Prima::Edit* section - text editor widget
the *Prima::ExtLists* section - listbox with checkboxes
the *Prima::FrameSet* section - frameset widget class
the *Prima::Grids* section - grid widgets
the *Prima::Widget::Header* section - multi-column header widget
the *Prima::ImageViewer* section - bitmap viewer
the *Prima::InputLine* section - input line widget
the *Prima::Label* section - static text widget
the *Prima::Lists* section - user-selectable item list widgets
the *Prima::MDI* section - top-level windows emulation classes
the *Prima::Notebooks* section - multipage widgets
the *Prima::Outlines* section - tree view widgets
the *Prima::PodView* section - POD browser widget
the *Prima::ScrollBar* section - scroll bars
the *Prima::Sliders* section - sliding bars, spin buttons, dial widget, etc.
the *Prima::TextView* section - rich text browser widget

## Building a menu

In Prima, a tree-like menu is built by building a set of nested arrays, where each array corresponds to a single menu entry. Such as, to modify the hello-world program to contain a simple menu, it is enough to write the code like this:

```
use Prima qw(Application MsgBox);
my $window = Prima::MainWindow-> new(
        text => 'Hello world!',
        menuItems => [
                [ '~File' => [
                        ['~Open', 'Ctrl+O', '^O', sub { message('open!') }],
                        ['~Save as...', sub { message('save as!') }],
                        [],
                        ['~Exit', 'Alt+X', km::Alt | ord('x'), sub { shift-> close } ],
                ]],
        ],
);
run Prima;
```


Each of the five arrays here in the example is written using different semantics, to represent either a text menu item, a sub-menu entry, or a menu separator. Strictly speaking, menus can also display images, but that syntax is practically identical to the text item syntax.

The idea behind all this complexity is to be able to tell what exactly the menu item is, just by looking at the number of items in each array. So, zero or one item is treated as a menu separator:

```
[],
[ 'my_separator' ]
```

The one-item syntax is needed when the separator menu item needs to be addressed explicitly. This means that each menu item after it is created is assigned a (unique) identifier, and that identifier looks like '#1', '#2', etc., unless it is given by the programmer. Here, for example, it is possible to delete the separator, after the menu is created:

```
$window-> menu-> remove('my_separator');
```

It is also possible to assign the identifier to any menu item, not just to separators. The other types (text, image, sub-menu) are differentiated by looking at the type of scalars they contain. Thus, a two-item array with the last item an array reference (or, as before, three-item for the explicit ID set), is a sub-menu. The reference, as in the example, may contain even more menu items:

```
menuItems => [
        [ '~File' => [
                [ '~Level1' => [
                        [ '~Level2' => [
                                [ '~Level3' => [
                                        []
                                ]],
                        ]],
                ]],
        ]],
],
```


Finally, text items, with the most complex syntax, can be constructed with three to six items in the array. One can set the left-aligned text string for the item, the right-aligned text string for the display of the hotkey, if any, the definition of the hotkey itself, and the action to be taken if the user has pressed either the menu item or the hotkey combination. Also, as in the previous cases, an explicit menu item ID can be set, and also an arbitrary data scalar, for the generic needs of the programmer.

Here are the combinations of scalars in an array that are allowed for defining a text menu item:

Three items - [ ID, text, action ]

Four items - [ text, hotkey text, hotkey, action ]

Five items - [ ID, text, hotkey text, hotkey, action ]

Six items - [ ID, text, hotkey text, hotkey, action, data ]

The image menu items are fully analogous to the text items, except that instead of the text string, an image object is supplied:

```
use Prima qw(Application MsgBox);
use Prima::Utils qw(find_image);

my $i = Prima::Image-> load( find_image( 'examples/Hand.gif'));
$i ||= 'No image found or can be loaded';

my $window = Prima::MainWindow-> new(
        text => 'Hello world!',
        menuItems => [
                [ '~File' => [
                        [ $i, sub {} ],
                ]],
```

```
        ],
);
run Prima;
```


The action item of the menu description array points to the code executed when the menu item is selected. It is either an anonymous subroutine, as it is shown in all the examples above, or a string. The latter case will cause the method of the menu owner ( in this example, the window ) to be called. This can be useful when constructing a generic class where the menu actions could be overridden:

```
use Prima qw(Application);

package MyWindow;
use vars qw(@ISA);
@ISA = qw(Prima::MainWindow);

sub action
{
        my ( $self, $menu_item) = @_;
        print "hey! $menu_item called me!\n"
}

my $window = MyWindow-> new(
        menuItems => [
                [ '~File' => [
                        ['~Action', q(action) ],
                ]],
        ],
);

run Prima;
```

All actions are called with the menu item identifier passed in as a string parameter.

Another useful trick here is how to define a hotkey. While the description of the hotkey can be an arbitrary string, which will be displayed as is, the definition of the hotkey is not that simple because one needs to encode the key combination that would trigger the menu item action. A hotkey can be defined in two ways. The hotkey definition scalar should either be a literal string such as `^A` for Control+A, or `@B` for Alt+B, or `^@#F10` for Control+Alt+Shift+F10. Or it should be a combination of the `km::` constants with the base key that is either the ordinal of the character letter, or the keycode, represented by one of the `kb::` constants. The latter method produces a less readable code, but is more explicit and powerful:

```
[ '~Reboot', 'Ctrl+Alt+Delete', km::Alt | km::Ctrl | kb::Delete, sub {
        print "wow!\n";
}],
[ '~Or not reboot?', 'Ctrl+Alt+R', km::Alt | km::Ctrl | ord('r'), sub {}],
```

This concludes the short tutorial on menus. To read more, see the *Prima::Menu* section .

## Adding help to your program

The toolkit comes with the POD viewer program `podview` which can be easily incorporated into any application. This is meant to be rather straightforward so you can write an application manual directly in the POD format.

- First, add some pod content to your main script, such as f ex:

  ```
  #!/usr/bin/env perl
  ...
  =pod

  =head1 NAME

  My program

  =cut
  ```

  exactly as if you wanted `perldoc` to display it.

- Second, add the invocation code, possibly inside the menu:

  ```
  ['~Help' => 'F1' => 'F1' => sub {
          $::application-> open_help("file://$0|Description");
  }],
  ```

  The `open_help` method can also take the standard `L<link>` syntax so for example the code

  ```
  open_help("My::Module/help")
  ```

  is also okay.

- Finally, consider if the text-only POD is okay for you or if you need any images embedded in the pod documentation. This is somewhat tricky because the perl maintainers actively reject the idea of having images in the pod format, while metacpan.org can display images in the perl documentation just fine, and so does Prima, however, both use different syntaxes.

  Here is an example of the mixed content that shows graphics when the graphic display is available, and just plain text otherwise:

  ```
  =for podview <img src="illustration.gif" cut=1 title="Horizontal font measurements">

  =for html <p>
  <figure>
  <img src="https://raw.githubusercontent.com/dk/Prima/master/pod/Prima/leadings.gif">
  <figcaption>Horizontal font measurements</figcaption>
  </figure>
  <!--

     .. plain text illustration ..

  =for html -->

  =for podview </cut>
  ```

The GIF image format is chosen because Prima keeps all of its internal images as multi-frame GIFs, so in a way, it is also the safest fallback. However, any other image file format will do too.

If you don't need the text fallback, just just this:

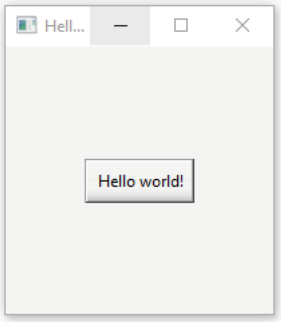
```
=for podview <img src="illustration.gif">

=for html <p><img src="https://raw.githubusercontent.com/me/myproject/master/illustration.
```

# 3 Core toolkit classes

## 3.1 Prima

A Perl graphic toolkit

### Synopsis




```
use Prima qw(Application Buttons);

Prima::MainWindow->new(
        text     => 'Hello world!',
        size     => [ 200, 200],
)-> insert( Button =>
        centered => 1,
        text     => 'Hello world!',
        onClick  => sub { $::application-> close },
);

run Prima;
```

See more screenshots at the *http:prima.eu.orgbig-picture* entry.

### Description

Prima is a classic 2D GUI toolkit that works under Windows and X11 environments. The toolkit features a rich widget library, extensive 2D graphic support, PDF generation, modern Unicode text input and output, and supports a wide set of image formats. Additionally, the RAD-style Visual Builder and POD viewer are included. The toolkit can interoperate with other popular event loop libraries.

## CLASS HIERARCHY

The toolkit is built with a combination of two basic sets of classes - core and external. The core classes are coded in C and form a baseline for every Prima object written in Perl. The usage of C is possible together with the toolkit; however, its full power is revealed in the Perl domain. The external classes present an easily expandable set of widgets, written entirely in Perl and communicating with the system using Prima library calls.

The core classes form a hierarchy, which is displayed below:

```
Prima::Object
        Prima::Component
                Prima::AbstractMenu
                        Prima::AccelTable
                        Prima::Menu
                        Prima::Popup
                Prima::Clipboard
                Prima::Drawable
                        Prima::DeviceBitmap
                        Prima::Printer
                        Prima::Image
                                Prima::Icon
                Prima::File
                Prima::Region
                Prima::Timer
                Prima::Widget
                        Prima::Application
                        Prima::Window
```

The external classes are derived from these; the list of widget classes can be found below in the *SEE ALSO* entry.

### Basic program

The very basic code shown in the *Synopsis* entry is explained here. The code creates a window with a 'Hello, world' title and a button with the same text. The program terminates after the button is pressed.

A basic construct for a program written with Prima requires

```
use Prima;
```

code; however, effective programming requires the usage of the other modules, for example, `Prima::Buttons`, which contains various button widgets. The `Prima.pm` module can be invoked together with a list of such modules, which makes the construction

```
use Prima;
use Prima::Application;
use Prima::Buttons;
```

shorter by using the following scheme:

```
use Prima qw(Application Buttons);
```

Another basic issue is the event loop, which is called by

```
run Prima;
```

code and requires a `Prima::Application` object to be created beforehand. Invoking the `Prima::Application` standard module is one of the possible ways to create an application object. The program usually terminates after the event loop is finished.

The main window is created by invoking

```
Prima::MainWindow->new();
```

or

```
Prima::MainWindow->create()
```

code with additional parameters. All Prima objects are created by the same scheme; the class name is passed as the first parameter, and a custom set of parameters is passed afterward:

```
$new_object = Class->new(
        parameter => value,
        parameter => value,
        ...
);
```

Here, parameters are the class property names, and they differ from class to class. Classes often have common properties, primarily due to object inheritance.

In the example, the following properties are used:

```
Window::text
Window::size
Button::text
Button::centered
Button::onClick
```

Property values can be of any scalar type. For example, the `::text` property accepts a string, `::size` - an anonymous array of two integers, and `onClick` - a sub.

onXxxx are special properties that describe *events* that can be used together with the `new`/`create` syntax, and are additive when the regular properties are substitutive (read more in the *Prima::Object* section). Events are called in the object context when a specific condition occurs. The `onClick` event here, for example, is called when the user presses (or otherwise activates) the button.

## API

This section describes miscellaneous methods, registered in the `Prima::` namespace.

**message TEXT**

Displays a system message box with TEXT.

**open_file, save_file**

When the `Prima::Dialog::FileDialog` module is loaded, these shortcut methods are registered in the `Prima::` namespace as an alternative to the same methods in the module's namespace. The methods execute standard file open and save dialogs, correspondingly.

See the *Prima::Dialog::FileDialog* section for more.

**run**

Enters the program event loop. The loop is ended when `Prima::Application`'s `destroy` or `close` method is called.

**parse_argv @ARGS**

Parses Prima options from @ARGS, returns unparsed arguments.

## OPTIONS

Prima applications do not have a portable set of arguments; it depends on the particular platform. Run

```
perl -e '$ARGV[0]=q(--help); require Prima'
```

or any Prima program with a `--help` argument to get the list of supported arguments. Programmatically, setting and obtaining these options can be done by using the `Prima::options` routine.

In cases where the Prima argument parsing conflicts with the application options, use the *Prima::noARGV* section to disable the automatic parsing; also see the *parse_argv* entry. Alternatively, the construct

```
BEGIN { local @ARGV; require Prima; }
```

will also do.

## 3.2 Prima::Object

Base toolkit classes

### Synopsis

```
if ( $obj-> isa('Prima::Component')) {

    # set and get a property
    my $name = $obj-> name;
    $obj->name( 'an object' );

    # set a notification callback
    $obj-> onPostMessage( sub {
        shift;
        print "hey! I've received this: @_\n";
    });

    # can set multiple properties. note, that 'name' and 'owner',
    # replaces the old values, while onPostMessage subs are aggregated.
    $obj-> set(
        name => 'AnObject',
        owner => $new_owner,
        onPostMessage => sub {
           shift;
           print "hey! me too!\n";
        },
    );

    # de-reference by name
    $new_owner-> AnObject-> post_message(1,2);
}
```

### Description

Prima::Object and Prima::Component are the root classes of the Prima toolkit hierarchy. All the other classes are derived from the Component class, which in turn is the only descendant of the Object class. Both of these classes are never used for instantiating objects, although this is possible with the

```
Prima::Component-> new( .. parameters ... );
```

call. This document describes the basic concepts of the OO programming with the Prima toolkit. Although the Component class has wider functionality than the Object class, all the examples will be explained using Component, since Object has no descendant classes other than Component anyway, and all the functionality of Object is present in Component too. This document partially overlaps with the information from the *Prima::internals* section, where the latter though focuses on a C programmer's perspective, while this document is about the perl programming.

### Object base features

#### Creation

Object creation has fixed syntax:

```
$new_object = Class-> new(
  parameter => value,
  parameter => value,
  ...
);
```

The parameters and the values form a hash which is passed to the new() method. This hash is applied to the default parameter-value hash ( a *profile* ), specific to every Prima class. The object creation is performed in several stages.

**new**

The new() constructor method calls the profile_default() method that returns ( as its name states ) the default profile, a hash with the appropriate default values assigned to its keys. The Component class defaults are

```
name        => ref $_[ 0],
owner       => $::application,
delegations => undef,
```

(also, see the source file Prima/Classes.pm):

While the exact meaning of these parameters is described later in the *Properties* entry, the idea is that a newly created object will have the 'owner' parameter set to '$::application' and 'delegations' to undef, etc etc - unless these parameters are explicitly passed to new(). Example:

```
$a1 = Prima::Component-> new();
```

$a1's owner will be $::application

```
$a2 = Prima::Component-> new( owner => $a1);
```

$a2's owner will be $a1. The actual merging of the default and the parameter hashes is performed in the next stage, in the profile_check_in() method which is called inside the profile_add() method.

Note: the older syntax used create() instead of new(), which is still valid but is not preferred.

**profile_check_in**

The profile_check_in() method merges the default and the parameter profiles. By default, all the specified parameters have the ultimate precedence over the default ones, but in case the parameter specification is incomplete or ambiguous, the profile_check_in()'s task is to determine the actual parameter values. For example, the Component::profile_check_in method maintains simple automatic naming of the newly created objects. That is, if the object's name was not passed to new() as a parameter, then it is assigned to a string formed from the class name and some number, for example, Component1, Component2, etc.

In another example, the Prima::Widget::profile_check_in() method resolves eventual ambiguities caused by different ways of assigning widget positions on the screen. A widget's horizontal position can be specified by using the `left` and `width` parameters, as well as by the `right`, `size`, and/or `rect`. The default of both `left` and `width` properties is 100. But if, for example, only the `right` parameter was passed to new(), then it is the profile_check_in()'s job to calculate the value for the `left` property, given that `width` is still 100.

After the profiles are merged, the resulting hash is passed to the third stage, init().

**init**

The init() method's task is to map the profile hash into the newly created object, e.g., assign the `name` parameter value to the `name` property, and so on - for all relevant parameters. After that, it has to return the profile so that the overridden init() methods can perform the same actions. This stage along with the previous one can be found in almost all Prima classes.

Note: usually the init() attaches the object to its owner in keep the newly-created object instance from being deleted by the garbage-collection mechanisms. See more on that later ( see the *Links between objects* entry).

After the init() finishes, the new() method calls the setup() method

**setup**

The setup() method is a convenience function, it is used when any post-init actions must be taken. It is seldom overloaded, primarily because the Component::setup() method calls the `onCreate` notification, which is more convenient to overload than the setup().

As can be noticed from the code pieces above, a successful new() call returns a newly created object. If an error condition occurs, undef is returned. Only the errors that were generated via die() during the init() stage result in undef. Other errors raise an exception instead. It is not recommended to wrap the new() calls in an `eval{}` block and to recover after the error because it can only occur in the two following situations. The first is a system error, either inside perl or Prima core (f ex an out-of-memory error), and not much can be done here since that error can very probably lead to an unstable program. The second reason is the programmer's error when a nonexistent parameter key or an invalid value is passed.

After a call to the new() method, the object can participate in the toolkit's event flow. The `onCreate` event is always the first event the object receives, and after it, other events can be sent and received.

### Destruction

Object destruction can be caused by many conditions, but the execution ultimately passes through the destroy() method. destroy(), as well as new(), performs the following finalizing steps:

**cleanup**

The first method called inside the destroy() is cleanup(). The cleanup() method is a counter-method to setup(), as destroy() is the counter-method to new(). cleanup() generates the `onDestroy` event, which again can be overridden more easily than the cleanup() itself.

`onDestroy` is always the last event the object sees. After the cleanup() no events are allowed to circulate.

**done**

The done() method is the counter-method to init() and is the proper place to free all object resources. Although it is as safe to overload done() as init(), it gets overloaded, primarily because overloading `onDestroy` is easier.

The typical conditions that lead to the object destruction are a direct destroy() call, the garbage collection mechanisms, the user-initiated window closing action ( for `Prima::Window` only ), and an exception during the init() stage. Thus, one must be careful implementing done() which could be also called if init() throws an exception.

### Methods

The class methods are declared and used with the perl OO syntax, which allows two ways of referencing a method in the object's class:

```
$object-> method();
```

and

```
method( $object);
```

The actual code is a sub, located under the object class package. The overloaded methods that call their ancestor code use the

```
$object-> SUPER::method();
```

syntax. Most of the Prima methods have a fixed number of parameters.

### Properties

Properties are methods that combine the functionality of two ephemeral methods, "get" and "set". The idea behind properties is that many object parameters require two independent methods, one that returns some internal state and another that changes it. For example, for managing the object name, set_name() and get_name() methods are needed. Indeed, the early Prima implementation dealt with a large amount of these get's and set's, but later these method pairs were deprecated in favor of the properties. Instead, there is now only one method name() ( referred to as `::name` later in the documentation ).

A property returns a value if no parameters (except the object itself) are passed, and changes the internal data to the passed parameters otherwise. Here's a sketch code for the `::name` property implementation:

```
sub name
{
   return $_[0]-> {name} unless $#_;
   $_[0]->{name} = $_[1];
}
```

There are many examples of properties throughout the toolkit. Not all properties deal with scalar values, some accept arrays or hashes as well. The properties can be set-called not only by name like

```
$object-> name( "new name");
```

but also with the set() method. The set() method accepts a hash, that is similar to hashes passed to new(), and also assigns its values to the corresponding properties. For example, the code

```
$object-> name( "new name");
$object-> owner( $owner);
```

can be rewritten as

```
$object-> set(
   name  => "new name",
   owner => $owner
);
```

A minor speed-up is gained here by eliminating some of the C-to-perl and perl-to-C calls, especially if the code called is implemented in C. The only problem with this technique is that the order in which the properties are set is undefined. Therefore, the usage of set() is recommended either when the property order is irrelevant, or it is known beforehand that such a call speeds up the code, or is the only way to achieve the required result. An example of the latter case shows that Prima::Image calls

```
$image-> type( $a);
$image-> palette( $b);
```

and

```
$image-> palette( $b);
$image-> type( $a);
```

produce different results. It is indeed the only solution to request a change that converts an image using both type and palette at the same time, to use the following code:

```
$image-> set(
   type => $a,
   palette => $b
);
```

This though makes sense only when it is known beforehand that `Prima::Image::set` is aware of this combination and calls neither `::type` nor `::palette` but performs another image conversion instead.

Some properties are read-only and some are write-only. Some methods that might be declared as properties are not; these are declared as plain methods with get_ or set_ name prefix. There is not much certainty about what methods are better off being declared as properties and vice versa.

However, if get_ or set_ methods cannot be used in, correspondingly, write or read fashion, the R/O and W/O properties can. They raise an exception in an attempt to do so.

### Links between objects

Prima::Component descendants can be used as containers, ie objects that are on a higher hierarchy level than the others, f ex the child-parent relationship. The 'children' objects have the `::owner` property value assigned to a reference to an 'owner' object, while the 'owner' object contains the list of its children. It is a one-to-many hierarchy scheme, as a 'child' object can only have a single owner, while an 'owner' object can have many children. The same object can be an owner and a child at the same time, so the owner-child hierarchy can be viewed as a tree-like structure, too.

The Prima::Component::owner property maintains such a relation, and is writable - the object can change its owner dynamically. There is no corresponding property that manages children objects, but there is the method get_components(), that returns an array of the child references.

The owner-child relationship is used in several ways in the toolkit. For example, the widgets that are children of another widget appear ( usually, but not always ) inside of the rectangular area occupied by the owner widget. Some events ( keyboard events, for example ) are propagated automatically up and/or down the object tree. Another important feature is that when an object gets destroyed its children are destroyed first. In a typical program the whole object tree roots in a Prima::Application object instance. When the application finishes, this feature helps clean up the widgets and quit gracefully.

Implementation note: the name 'owner' was taken instead of the initial 'parent', because the 'parent' is a fixed term for widget hierarchy relationship description. The Prima::Widget relationship between owner and child is not the same as GUI's parent-to-child. The parent is the widget for the children widgets located in and clipped by its inferior. The owner widget is more than that, its children can be located outside its owner's boundaries.

An alternative to the new() method, the insert() method is used to explicitly select the owner of the newly created object. The insert() method too can be considered a constructor in the OO-terms. It makes the code

```
$obj = Class-> new( owner => $owner, name => 'name);
```

more readable by introducing the

```
$obj = $owner-> insert( 'Class', name => 'name');
```

syntax. These two code blocks are identical

There is another type of relation where objects can hold references to each other. Internally this link level is used to keep objects from deletion by garbage collection mechanisms. This relation is the many-to-many scheme, where every object can have many links to other objects. This functionality is managed by the attach() and detach() methods.

## Events

Prima::Component descendants employ a well-developed event propagation mechanism, which allows the handling of events using several different schemes. An event is a condition, caused by the system or the user, or an explicit notify() call. The formerly described events onCreate and onDestroy are triggered after a new object is created or before it gets destroyed. These two events, and the described below onPostMessage event are available for all Prima objects. New classes can register their own events and define their execution flow, using the notification_types() method. This method returns all available information about the events registered in a class.

Prima defines also the non-object event dispatching and filtering mechanism, available through the the *event_hook* entry static method.

### Propagation

The event propagation mechanism has three different schemes of registering a user-defined callback, either on the object itself, on the object class, or on the class of some other object

In the descriptions of the schemes below, there are example codes of how to catch the following event depending on the scheme used:

```
$obj-> notify("PostMessage", $data1, $data2);
```

#### Direct methods

As is usual in the OO programming, event callback routines are declared as methods. 'Direct methods' employ this paradigm too, so if the class method named `on_postmessage` is present, it will be called as a method ( i.e., in the object context ) when the `onPostMessage` event is sent. For example:

```
sub on_postmessage
{
   my ( $self, $data1, $data2) = @_;
   ...
}
```

The callback name is the modified lower-case event name: the name for the Create event is on_create, PostMessage - on_postmessage, etc. These methods can be overloaded in the object's class descendants. The only note on declaring these methods in the first instance is that no `::SUPER` call is needed because these methods are not defined by default.

Usually, the direct methods are used for internal bookkeeping, reacting to the events that are not meant to be passed to the program. For example, the Prima::Button class catches mouse and keyboard events in such a way, because usually, the only notification that is interesting for the code that employs push-buttons is `Click`, and rarely anything else. This scheme is convenient when an event handling routine serves internal, implementation-specific needs.

#### Delegated methods

The delegated methods are used when objects ( mostly widgets ) include other dependent objects, and the functionality requires interaction between these. The callback functions

here are the same methods as the direct methods, except that they get called in the context of two, not one, objects. If, for example, an $obj's owner, $owner, would be interested in $obj's PostMessage event, it would register the notification callback by issuing the following call:

```
$obj-> delegations([ $owner, 'PostMessage']);
```

where the actual callback sub will be declared as

```
sub Obj_PostMessage
{
   my ( $self, $obj, $data1, $data2) = @_;
}
```

Note that the naming style is different - the callback name is constructed from the object name ( let's assume that $obj's name is 'Obj') and the event name. ( This is one of the reasons why Component::profile_check_in() performs automatic naming of newly created objects). Note also that the context objects are $self ( that equals $owner in this case) and $obj.

The delegated methods can be used not only for owner-child relations. Every Prima object is free to add a delegation method to every other object. However, if the objects are in other than the owner-child relation, it is a good practice to add Destroy notification to the object whose events are of interest, so if it gets destroyed, the partner object gets a message about that.

**Anonymous subroutines**

The two previous callback types are more relevant when a separate class is designed. However, in the Prima toolkit, it is not necessary to declare a new class every time the event handling is needed. It is possible to use the third and the most powerful event hook scheme using perl anonymous subroutines ( subs ) for easy customization.

Contrary to the usual OO event implementations, when only one routine per class dispatches an event and calls the inherited handlers when it is appropriate, the Prima event handling mechanism can accept many event handlers for one object ( it is greatly facilitated by the fact that perl has *anonymous subs*, however).

All the callback routines are called when an event is triggered, one by one in turn. If the direct and delegated methods can only be multiplexed by the usual OO inheritance, the anonymous subs are allowed to be many by design. There are three syntaxes for setting such an event hook; the example below sets a hook on $obj using each syntax for a different situation:

- during new():

```
$obj = Class-> new(
 ...
 onPostMessage => sub {
    my ( $self, $data1, $data2) = @_;
 },
 ...
 );
```

- after new() using set()

```
$obj-> set( onPostMessage => sub {
    my ( $self, $data1, $data2) = @_;
});
```

- after new() using the event name:

```
$obj-> onPostMessage( sub {
    my ( $self, $data1, $data2) = @_;
});
```

The events can be addressed as properties, with the exception that they are not substitutive but additive. The additivity means that when the latter type of syntax is used, the subs already registered do not get overwritten or discarded but stack in the internal object queue. Thus,

```
$obj-> onPostMessage( sub { print "1" });
$obj-> onPostMessage( sub { print "2" });
$obj-> notify( "PostMessage", 0, 0);
```

code block would print

```
21
```

as the execution result.

It is a distinctive feature of the Prima toolkit that two objects of the same class may have different set of event handlers.

### Flow

When there is more than one handler of a particular event type present on an object, a question may arise about what are the callback's call priorities and when the event processing stops. One of the ways to regulate the event flow is based on prototyping events, by using the notification_types() event type description. This function returns a hash, where the keys are the event names and the values are the constants that describe the event flow. A constant is a bitwise OR combination of several basic flow `nt::XXX` constants, that control the following three aspects of the event flow:

### Order

If both anonymous subs and direct/delegated methods are present, the object needs to decide which callback class must be called first. Both 'orders' are useful: for example, if a class is designed in such a way that some default action is meant to be overridden, it is better to call the custom actions first. If, on the contrary, a class event handler does most of the heavy lifting, then the reverse order may be preferred instead. One of the two `nt::PrivateFirst` and `nt::CustomFirst` constants defines the event execution order.

### Direction

Almost the same as the order, but used for finer granulation of the event flow, the direction constants `nt::FluxNormal` and `nt::FluxReverse` are used. The 'normal flux' defines the FIFO ( first in first out ) direction. That means, that the sooner the callback is registered, the greater priority it would have during the execution. The code block from the example above

```
$obj-> onPostMessage( sub { print "1" });
$obj-> onPostMessage( sub { print "2" });
$obj-> notify( "PostMessage", 0, 0);
```

results in `21`, not `12` because the PostMessage event type is prototyped as `nt::FluxReverse`.

**Execution control**

It was stated above that the events are additive, - the callback storage is never discarded when 'set'-syntax is used. However, the event can be told to behave like a substitutive property, e.g. to call one and only one callback. This functionality is managed by the `nt::Single` bit in the execution control constant set, which consists of the following constants:

```
nt::Single
nt::Multiple
nt::Event
```

These constants are mutually exclusive, and may not appear together in an event type declaration. A `nt::Single`-prototyped notification calls only the first ( or the last - depending on order and direction bits ) callback. The usage of this constant is somewhat limited.

In contrast with `nt::Single`, the `nt::Multiple` constant sets the execution control to call all the available callbacks, with respect to the direction and the order bits.

The third constant, `nt::Event`, is the same as `nt::Multiple`, except that the event flow can be stopped at any time by calling the clear_event() method.

Although there are 12 possible event type combinations, half of them are not usable for anything. The combinations from another half were assigned more-less descriptive names:

```
nt::Default       ( PrivateFirst | Multiple | FluxReverse)
nt::Property      ( PrivateFirst | Single   | FluxNormal )
nt::Request       ( PrivateFirst | Event    | FluxNormal )
nt::Notification  ( CustomFirst  | Multiple | FluxReverse )
nt::Action        ( CustomFirst  | Single   | FluxReverse )
nt::Command       ( CustomFirst  | Event    | FluxReverse )
```

**Success state**

Events do not return values, although the event generator, the notify() method does - it returns either 1 or 0, which is the value of the event state. The 0 and 1 results however do not mean either success or failure, they simply reflect the fact whether the clear_event() method was called during the processing - 1 if it was not, 0 otherwise. The state is kept during the whole processing stage and can be accessed by the Component::eventFlag property. Since it is allowed to call the notify() method inside event callbacks, the object maintains a stack for those states. The Component::eventFlag property always works with the topmost one and fails if is called from outside the event processing stage; clear_event() is no more than an alias for the eventFlag(0) call. The state stack is operated by the push_event() and pop_event() methods.

Implementation note: a call to clear_event() inside a `nt::Event`-prototyped event call does not automatically stop the execution. The execution stops if the state value equals 0 after the callback is finished. The eventFlag(1) call thus cancels the effect of clear_event().

A particular coding style is used when the event is `nt::Single`-prototyped and is called many times in a row, so overheads of calling notify() become a burden. Although the notify() logic is somewhat complicated, it is rather simple in the `nt::Single` case. The helper function get_notify_sub() returns the context of the callback to be called, so it can be used to emulate the notify() behavior. For example:

```
for ( ... ) {
   $result = $obj-> notify( "Measure", @parms);
}
```

can be expressed in more cumbersome, but efficient code if the `nt::Single`-prototyped event is used:

```
my ( $notifier, @notifyParms) = $obj-> get_notify_sub( "Measure" );
$obj-> push_event;
for ( ... ) {
    $notifier-> ( @notifyParms, @parms);
    # $result = $obj-> eventFlag; # this is optional
}
$result = $obj-> pop_event;
```

## Inheritance

The design of the Prima classes is meant to be as close as possible to the standard perl OO model. F.ex. to subclass a new package, a standard

```
use base qw(ParentClass);
```

or even

```
out @ISA = qw(ParentClass);
```

should be just fine.

However, there are special considerations about the multiple inheritance and the order of the ancestor classes. First, the base class should be a Prima class, i e

```
use base qw(Prima::Widget MyRole);
```

not

```
use base qw(MyRole Prima::Widget);
```

This is caused by the perl OO model where if more than one base class has the same method, only the first method will be actual, and Prima conforms to that. F ex defining a `MyRole::init` won't have any effect where MyRole is not the first base class (and things will explode badly if it is).

In a very special case where the MyRole class *needs* to have methods that overload Prima core, XS-implemented methods, a special technique is used:

- First, in MyRole, declare a special method `CORE_METHODS`, returning all names of the core symbols to be overloaded in that role:

  ```
  package MyRole;

  sub CORE_METHODS { qw(setup) }
  ```

  Do not subclass MyRole from Prima objects though.

- Define the methods as if you would define a normal overridden method, with one important exception: since perl's SUPER is package-based, not object-based, the `$self->SUPER::foo()` pattern will not work for calling the methods that are up in the hierarchy. Instead, the first parameter to these methods is an anonymous subroutine that will call the needed SUPER method:

  ```
  sub setup
  {
      my ( $orig, $self ) = ( shift, shift );
      ...
      $orig->( $self, @_ );
      ...
  }
  ```

If you know the *Moose* entry standard syntax `around`, this is the same idea.

Note that this method will be called *after* the descendant class `setup` if the class has one. This is a bit confusing as in all types of OO inheritance sub-class code is always called after the super-class, not vice versa. This might change in the future, too.

- In the descendant class, inherit from the MyRole normally, but in addition to that make the call to overload its special methods:

```
package MyWidget;
use base qw(Prima::Widget MyRole);
__PACKAGE__->inherit_core_methods('MyRole');
```

Check also the *Prima::Widget::GroupScroller* section as an example.

# API

## Prima::Object methods

### alive

Returns the object 'vitality' state - true if the object is alive and usable, false otherwise. This method can be used as a general checkout if the scalar passed is a Prima object, and if it is usable. The true return value can be 1 for normal and operational object state, and 2 if the object is alive but in its init() stage. Example:

```
print $obj-> name if Prima::Object::alive( $obj);
```

### cleanup

Called right after the destroy() started. Used to initiate the `cmDestroy` event. Is never called directly.

### create CLASS, %PARAMETERS

Same as the *new* entry.

### destroy

Initiates the object destruction. Calls cleanup() and then done(). destroy() can be called several times and is the only Prima re-entrant function, therefore may not be overloaded.

### done

Called by the destroy() method after cleanup() is finished. Used to free the object resources, as a finalization stage. During done() no events are allowed to circulate, and alive() returns 0. The object is not usable after done() finishes. Is never called directly.

Note: the eventual child objects are destroyed inside the done() call.

### get @PARAMETERS

Returns a hash where the keys are @PARAMETERS and values are the corresponding object properties.

### init %PARAMETERS

The most important stage of the object creation process. %PARAMETERS is the modified hash that was passed to new(). The modification consists of merging with the result of the profile_default() method inside the profile_check_in() method. init() is responsible for applying the relevant data from PARAMETERS to the corresponding object properties. Is never called directly.

**insert CLASS, %PARAMETERS**

A convenience wrapper for new() that explicitly sets the owner property for a newly created object.

```
$obj = $owner-> insert( 'Class', name => 'name');
```

is identical to

```
$obj = Class-> new( owner => $owner, name => 'name);
```

insert() has another syntax that allows simultaneous creation of several objects:

```
@objects = $owner-> insert(
  [ 'Class', %parameters],
  [ 'Class', %parameters],
  ...
);
```

With this syntax, all newly created objects would have $owner set to their 'owner' properties.

**new CLASS, %PARAMETERS**

Creates a new object instance of the given CLASS and sets its properties corresponding to the passed parameter hash. Examples:

```
$obj = Class-> new( PARAMETERS);
$obj = Prima::Object::new( "class" , PARAMETERS);
```

Is never called in an object context.

Alias: create()

**profile_add PROFILE**

The first stage of the object creation process. The PROFILE is a reference to the PARAMETERS hash, passed to the new() method. The hash is merged with the hash produced by the profile_default() method after passing both through the profile_check_in(). The merge result is stored back in PROFILE.

The method is never called directly.

**profile_check_in CUSTOM_PROFILE, DEFAULT_PROFILE**

The second stage of the object creation process. Resolves eventual ambiguities in CUSTOM_PROFILE, which is the reference to the PARAMETERS passed to new(), by comparing to and using the default values from the DEFAULT_PROFILE, which in turn is the result of the profile_default() method.

The method is never called directly.

**profile_default**

Returns a hash of the appropriate default values for all properties of the class. In the object creation process serves as a provider of fall-back values, and is called (once) during the process. The method can be used directly, contrary to the other creation process-related functions.

Can be called in a context of a class.

**raise_ro TEXT**

Throws an exception with text TEXT when a read-only property is called in a set- context.

**raise_wo TEXT**

Throws an exception with text TEXT when a write-only property is called in a get-context.

**set %PARAMETERS**

The default behavior is equivalent to the following code:

```
sub set
{
   my $obj = shift;
   my %PARAMETERS = @_;
   $obj-> $_( $PARAMETERS{$_}) for keys %PARAMETERS;
}
```

Assigns the object properties correspondingly to the PARAMETERS hash. Many Prima::Component descendants overload set() to make it more efficient for particular parameter key patterns.

Like the code above, raises an exception if the key in PARAMETERS has no correspondent object property.

**setup**

The last stage of the object creation process. Called after init() finishes. Used to initiate the `onCreate` event. Is never called directly.

### Prima::Component methods

**add_notification NAME, SUB, REFERRER = undef, INDEX = -1**

Adds the SUB to the list of notifications for the event NAME. REFEREE is the object reference, which is used to create a context to the SUB and is also passed as a parameter to it when the event callback is called. If the REFEREE is undef ( or is not specified ), then the caller object is assumed. REFEREE also gets implicitly attached to the object, - the implementation frees the link between the objects when one of these gets destroyed.

INDEX is a desired insert position in the notification list. By default, it is -1, which means 'in the start'. If the notification type contains nt::FluxNormal bit set, the newly inserted SUB will be called first. If it has nt::FluxReverse, it is called last, correspondingly.

Returns a positive integer value on success, and 0 on failure. This value can be later used to refer to the SUB in remove_notification().

See also: `remove_notification`, `get_notification`.

**attach OBJECT**

Inserts the OBJECT into the list of the attached objects and increases the OBJECT's reference count. The list may not hold more than one reference to the same object; the warning is issued on such an attempt.

See also: `detach`.

**bring NAME, MAX_DEPTH=0**

Looks for the child object that has a name that equal to NAME. Returns its reference on success, undef otherwise. It is a convenience method, that makes possible the usage of the following constructs:

```
$obj-> name( "Obj");
$obj-> owner( $owner);
...
$owner-> Obj-> destroy;
...
$obj-> deepChildLookup(1);
$obj-> insert(Foo => name => 'Bar');
$owner-> Bar-> do_something;
```

See also: `find_component`, `deepChildLookup`

**can_event**

Returns true if the object event circulation is allowed. In general, the same as `alive() == 1`, except that can_event() fails if an invalid object reference is passed.

**clear_event**

Clears the event state, that is set to 1 when the event processing begins. Signals the event execution stop for the nt::Event-prototyped events.

See also: the *Events* entry, `push_event`, `pop_event`, `::eventFlag`, `notify`.

Use this call in your overloaded event handlers when signalling that further processing should be stopped, f ex onMouseDown doing something else than the base widget.

See more in the *Execution control* entry. Check the exact `nt::` type of the event in the Prima/Classes.pm source.

**detach OBJECT, KILL**

Removes the OBJECT from the list of the attached objects and decreases the OBJECT's reference count. If KILL is true, destroys the OBJECT.

See also: `attach`

**event_error**

Issues a system-dependent warning sound signal.

**event_hook [ SUB ]**

Installs the SUB to receive all events on all Prima objects. The SUB receives the same parameters passed to the *notify* entry and must return an integer, either 1 or 0, to pass or block the event respectively.

If no SUB is set, returns the currently installed event hook pointer. If SUB is set, replaces the old hook sub with SUB. If SUB is `'undef'`, event filtering is not used.

Since the `'event_hook'` mechanism allows only one hook routine to be installed at a time, direct usage of the method is discouraged. Instead, use the the *Prima::EventHook* section API for multiplexing access to the hook.

The method is static and can be called either with or without a class or an object as a first parameter.

**find_component NAME**

Performs a depth-first search on children tree hierarchy, matching the object that has a name equal to NAME. Returns its reference on success, undef otherwise.

See also: `bring`

**get_components**

Returns an array of the child objects.

See: `new`, the *Links between objects* entry.

**get_handle**

Returns a system-dependent handle for the object. For example, Prima::Widget returns its system Window/HWND handles, Prima::DeviceBitmap - its system Pixmap/HBITMAP handles, etc.

Can be used to pass the handle value outside the program, for eventual interprocess communication.

**get_notification NAME, @INDEX_LIST**

For each index in the INDEX_LIST returns three scalars, bound to the index position in the NAME event notification list. These three scalars are REFERRER, SUB, and ID. REFERRER and SUB are those passed to `add_notification`, and ID is its saved result.

See also: `remove_notification`, `add_notification`.

**get_notify_sub NAME**

A convenience method for the nt::Single-prototyped events. Returns the code reference and the context for the first notification sub for event NAME.

See the *Success state* entry for example.

**notification_types**

Returns a hash, where the keys are the event names and the values are the `nt::` constants that describe the event flow.

Can be called in the context of a class.

See the *Events* entry and the *Flow* entry for details.

**notify NAME, @PARAMETERS**

Calls the subroutines bound to the event NAME with parameters @PARAMETERS in the context of the object. The calling order is described by the `nt::` constants, from the hash returned by the notification_types().

notify() accepts a variable number of parameters, and while it is possible, it is not recommended to call notify() with the excessive number of parameters. The call with the deficient number of parameters results in an exception.

Example:

```
$obj-> notify( "PostMessage", 0, 1);
```

See the *Events* entry and the *Flow* entry for details.

**pop_event**

Closes the event processing stage bracket.

See `push_event`, the *Events* entry

**post_message SCALAR1, SCALAR2**

Calls the `PostMessage` event with parameters SCALAR1 and SCALAR2 once during the next idle event loop. Returns immediately. Does not guarantee that `PostMessage` will be called, however.

See also the **post** entry in the *Prima::Utils* section

**push_event**

Opens the event processing stage bracket.

See `pop_event`, the *Events* entry

**remove_notification ID**

Removes the notification subroutine that was registered before using the `add_notification` method, and where the ID was its result. After the successful removal, the eventual context object gets implicitly detached from the storage object.

See also: `add_notification`, `get_notification`.

**set_notification NAME, SUB**

Adds the SUB to the event NAME notification list. Rarely used directly, but is a key point in enabling the following syntax:

```
$obj-> onPostMessage( sub { ... });
```

or

```
$obj-> set( onPostMessage => sub { ... });
```

that are shortcuts for

```
$obj-> add_notification( "PostMessage", sub { ... });
```

**unlink_notifier REFERRER**

Removes all notification subs from all event lists bound to the REFERRER object.

**Prima::Component properties**

**deepChildLookup BOOL**

If set, the lookup by name uses a breadth-first deep lookup into the object hierarchy. If unset (default), only immediate children objects are searched.

```
$self->deepChildLookup(0);
$self->Child1->GrandChild2;
...
$self->deepChildLookup(1);
$self->GrandChild2;
```

**eventFlag STATE**

Provides access to the last event processing state in the object event state stack.

See also: the *Success state* entry, `clear_event`, the *Events* entry.

**delegations [ <REFERRER>, NAME, <NAME>, < <REFERRER>, NAME, ... > ]**

Accepts an anonymous array in the *set-* context, which consists of a list of event NAMEs, that a REFERRER object ( the caller object by default ) is interested in. Registers notification entries if the subs with the naming scheme REFERRER_NAME are present on the REFERRER namespace. The example code

```
$obj-> name("Obj");
$obj-> delegations([ $owner, 'PostMessage']);
```

registers the Obj_PostMessage callback if it is present in the $owner namespace.

In the *get-* context returns an array reference that reflects the object's delegated events list content.

See also: the *Delegated methods* entry.

**name NAME**

Maintains the object name. NAME can be an arbitrary string, however it is recommended against the usage of special characters and spaces in NAME, to facilitate the indirect object access coding style:

```
$obj-> name( "Obj");
$obj-> owner( $owner);
...
$owner-> Obj-> destroy;
```

and to prevent system-dependent issues. If the system provides capabilities that allow to predefining some object parameters by its name (or its class), then it is impossible to know beforehand the system naming restrictions. For example, in the X11 window system the following resource string would make all Prima toolkit buttons green:

```
Prima*Button*backColor: green
```

In this case, using special characters such as `:` or `*` in the name of an object would make the X11 resource unusable.

**owner OBJECT**

Sets the owner of the object, which may be a Prima::Component descendant. Setting an owner to an object does not alter its reference count. Some classes allow OBJECT to be undef, while some do not. All widget objects can not exist without a valid owner; Prima::Application on the contrary can only exist with the owner set to undef. Prima::Image objects are indifferent to the value of the owner property.

Changing the owner dynamically is allowed, but it is a main source of implementation bugs since the whole hierarchy tree needs to be recreated. Although this effect is not visible in perl, the results are deeply system-dependent, and the code that changes owner property should be thoroughly tested.

Changes to the `owner` result in up to three notifications: `ChangeOwner`, which is called to the object itself, `ChildLeave`, which notifies the previous owner that the object is about to leave, and `ChildEnter`, telling the new owner about the new child.

### Prima::Component events

**ChangeOwner OLD_OWNER**

Called when the object changes its owner.

**ChildEnter CHILD**

Triggered when a child object is attached, either as a new instance or as a result of runtime owner change.

**ChildLeave CHILD**

Triggered when a child object is detached, either because it is getting destroyed or as a result of runtime owner change.

**Create**

The first event the object sees. Called automatically after init() is finished. Is never called directly.

**Destroy**

The last event the object sees. Called automatically before done() is started. Is never called directly.

**PostMessage SCALAR1, SCALAR2**

Called after the post_message() call is issued, however not inside post_message() but after the next idle event loop. SCALAR1 and SCALAR2 are the data passed to the post_message().

**SysHandle**

Sometimes Prima needs to implicitly re-create the system handle of a component. The re-creation usually happens deep inside the Prima core, however, if widgets on the screen are re-created, then they might get repainted. This happens when the underlying system either doesn't have API to change a certain property during the runtime or when such a re-creation happens on one of the component's parents, leading to a downward cascade of re-creation of the children. Also, it may happen when the user changes some system settings resolution so that some resources have to be changed accordingly.

This event will be only needed when the system handle (that can be acquired by `get_handle` ) is used further, or in the case when Prima doesn't restore some properties bound to the system handle.

## 3.3 Prima::Classes

Binder module for the built-in classes.

### Description

`Prima::Classes` and the *Prima::Const* section form a minimal set of perl modules needed for the toolkit to run. Since the module provides bindings for the core classes, it is required to be included in every Prima-related module and program.

## 3.4 Prima::Drawable

Generic 2-D graphic interface

### Synopsis

```
if ( $object-> isa('Prima::Drawable')) {
     $object-> begin_paint;
     $object-> color( cl::Black);
     $object-> line( 100, 100, 200, 200);
     $object-> ellipse( 100, 100, 200, 200);
     $object-> end_paint;
}
```

### Description

Prima::Drawable is a descendant of the Prima::Component class. It provides access to the system graphic context and canvas through its methods and properties. The Prima::Drawable descendants Prima::Widget, Prima::Image, Prima::DeviceBitmap, and Prima::Printer are backed by system-dependent routines that allow drawing and painting on the system objects.

### Usage

Prima::Drawable, as well as its ancestors Prima::Component and Prima::Object, is never used directly because the Prima::Drawable class by itself provides only the interface. It provides a three-state object access - when drawing and painting are enabled, when these are disabled, and the information acquisition state. By default, the object is created in a paint-disabled state. To switch to the enabled state, the begin_paint() method is used. Once in the enabled state, the object drawing and painting methods apply to the system canvas. To return to the disabled state, the end_paint() method is called. The information state can be managed by using begin_paint_info() and end_paint_info() methods pair. An object cannot be triggered from the information state to the enabled state ( and vice versa ) directly.

#### Graphic context and canvas

The graphic context is the set of variables, that control how exactly graphic primitives are rendered. The variable examples are color, font, line width, etc. Another term used here is *canvas* - the graphic area of a certain extent, connected to the Prima object, where the drawing and painting methods are used.

In all three states, a graphic context is allowed to be modified, but in different ways. In the disabled state, a graphic context value is saved as a template; when an object enters the information or the enabled state, all values are preserved, but when the object is back to the disabled state, the graphic context is restored to the values last assigned before entering the enabled state. The code example below illustrates the idea:

```
$d = Prima::Drawable-> create;
$d-> lineWidth( 5);
$d-> begin_paint_info;
# lineWidth is 5 here
$d-> lineWidth( 1);
# lineWidth is 1
$d-> end_paint_info;
# lineWidth is 5 again
```

( Note: `::region` and `::clipRect` properties are exceptions. They cannot be used in the disabled state. The values of these properties, as well as the property `::matrix` are neither recorded nor used as a template).

That is, in the disabled state any Drawable maintains only the graphic context values. To draw on a canvas, the object must enter the enabled state by calling begin_paint(). This function can be unsuccessful because the object binds with system resources during this stage, and allocation of those may fail. Only after the enabled state is entered, the canvas is accessible:

```
$d = Prima::Image-> create( width => 100, height => 100);
if ( $d-> begin_paint) {
   $d-> color( cl::Black);
   $d-> bar( 0, 0, $d-> size);
   $d-> color( cl::White);
   $d-> fill_ellipse( $d-> width / 2, $d-> height / 2, 30, 30);
   $d-> end_paint;
} else {
   die "can't draw on image:$@";
}
```

Different objects are mapped to different types of canvases - `Prima::Image` canvas retains its content after end_paint(), `Prima::Widget` maps it to some screen area, which content more transitory, etc.

The information state is as same as the enabled state, but the changes to the canvas are not visible. Its sole purpose is to read, not to write information. Because begin_paint() requires some amount of system resources, there is a chance that a resource request can fail, for any reason. The begin_paint_info() requires some resources as well, but usually much less, and therefore if only information is desired, it is usually faster and cheaper to obtain it inside the information state. A notable example is the get_text_width() method, which returns the length of a text string in pixels. It works in both enabled and information states, but code

```
$d = Prima::Image-> create( width => 10000, height => 10000);
$d-> begin_paint;
$x = $d-> get_text_width('A');
$d-> end_paint;
```

is much more expensive than

```
$d = Prima::Image-> create( width => 10000, height => 10000);
$d-> begin_paint_info;
$x = $d-> get_text_width('A');
$d-> end_paint_info;
```

for the obvious reasons.

It must be noted that some information methods like get_text_width() work even under the disabled state; the object is switched to the information state implicitly if it is necessary.

See also: the *graphic_context* entry.

### Color space

Graphic context and canvas operations rely completely on a system implementation. The internal canvas color representation is therefore system-specific, and usually could not be described in Prima definitions. Often the only information available about color space is its color depth.

Therefore all color manipulations, including dithering and antialiasing are subject to system implementation, and can not be controlled from perl code. When a property is set on the object in the disabled state, it is recorded verbatim; color properties are no exception. After the object

switches to the enabled state, a color value is translated to the system color representation, which might be different from Prima's. For example, if the display color depth is 15 bits, 5 bits for every component, then the white color value 0xffffff is mapped to

```
11111000 11111000 11111000
--R----- --G----- --B-----
```

that equals to 0xf8f8f8, not 0xffffff ( See the *Prima::gp-problems* section for inevident graphic issues discussion ).

The Prima::Drawable color format is RRGGBB, with each component resolution of 8 bits, thus allowing 2^24 color combinations. If the device color space depth is different, the color is truncated or expanded automatically. In case the device color depth is insufficient, dithering algorithms may apply.

Note: not only color properties but all graphic context properties allow all possible values in the disabled state, which are translated into system-allowed values when entering the enabled and the information states. This feature can be used to test if a graphic device is capable of performing certain operations ( for example, if it supports raster operations - the printers usually do not ). Example:

```
$d-> begin_paint;
$d-> rop( rop::Or);
if ( $d-> rop != rop::Or) { # this assertion is always false without
   ...                      # begin_paint/end_paint brackets
}
$d-> end_paint;
```

There are two color properties on each drawable - `::color` and `::backColor`. The values they operate are unsigned integers in the discussed above RRGGBB 24-bit format, however, the toolkit defines some mnemonic color constants as well:

```
cl::Black
cl::Blue
cl::Green
cl::Cyan
cl::Red
cl::Magenta
cl::Brown
cl::LightGray
cl::DarkGray
cl::LightBlue
cl::LightGreen
cl::LightCyan
cl::LightRed
cl::LightMagenta
cl::Yellow
cl::White
cl::Gray
```

It is not unlikely that if a device's color depth is insufficient, the primitives could be drawn with dithered or incorrect colors. This usually happens on paletted displays, with 256 or fewer colors.

There exist two methods that facilitate the correct color representation. The first way is to get as much information as possible about the device. The methods get_nearest_color() and get_physical_palette() provide a possibility to avoid mixed colors drawing by obtaining indirect information about solid colors, supported by a device. Another method is to use the `::palette`

property. It works by inserting the colors into the system palette, so if an application knows the colors it needs beforehand, it can employ this method - however, this might result in a system palette flash when a window focus toggles.

Both of these methods are applicable both with drawing routines and image output. An application that desires to display an image with the least distortion is advised to export its palette to an output device because images usually are not subject to automatic dithering algorithms. The Prima::ImageViewer module employs this scheme.

### Antialiasing and alpha

If the system has the capability for antialiased drawing and alpha rendering, Prima can use it. The render mode can be turned on by calling

```
$drawable->antialias(1)
```

which turns on the following effects:

- All primitives except images, `pixel`, and `bar_alpha` can be plotted with antialiased edges (text is always antialiased when possible)
- Graphic coordinates are then used as floating point numbers, not integers. That means that f ex a call

  ```
  $drawable->rectangle(5.3, 5.3, 10, 10)
  ```

  that had its coordinates automatically rounded to (5,5,10,10), now will render the primitive with subpixel precision, where its two edges will be divided between pixels 5 and 6, by using half-tones.

  Another note on the rounding of coordinates: historically almost all Prima pixel coordinates were integers, and implicit rounding of the passed numbers was done using the `int` function, i e int(0.7)=0 and int(-0.7)=0. This was later changed, breaking some backward compatibility, and now the rounding function is a more robust `R = floor(x + 0.5)`, where `R(0.7) = 1` and `R(-0.7) = -1`.
- The coordinate offset (0,0) moves to the ephemeral point between screen pixels (0,0),(-1,0),(-1,-1),(0,-1), which in turn leads to different results when plotting closed shapes. F.ex. an ellipse with a diameter of 3 pixels looks like this:

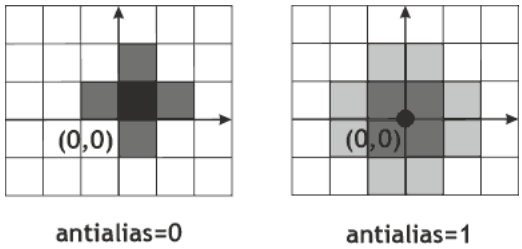



Figure 3.1: 3px ellipse and origin of the coordinate grid

  That is why if you need to draw a robust graphic routine that would have more or less identical results in both antialiasing modes, the closed shapes may need to decrease the diameter by 1 pixel when the antialiasing mode is set.

For the cases where the system does not support antialiasing, Prima provides the the *Prima::Drawable::Antialias* section emulation class, available through the `new_aa_surface` call.

To see if alpha and antialiasing are supported on the system, check the `sv::Antialias` value. To see if a particular drawable supports alpha layering, check the `can_draw_alpha` method.

Note that in the render mode, all painting operations treat the alpha channel differently, which can have a dramatic difference on layered surfaces. In the normal mode, the alpha channel is completely ignored, and using normal mode paints on a layered widget always produces a translucent window because the alpha value will always be 0, and the color bits are assumed to be already premultiplied. In the render mode, the alpha channel is addressed by the `alpha` property when drawing primitives, or in the `mask` property when drawing icons (again, drawing images and non-layered bitmaps assumes alpha = 0). The same is valid for fill pattern images and fill pattern icons.

### Monochrome bitmaps

Prima has special rules when drawing a monochrome Prima::DeviceBitmap. Such objects don't possess an inherent color palette, and by definition are bitmaps with only two pixel values present, 0s and 1s. When a monochrome bitmap is drawn, 0s are painted using the color value of the target canvas `color` property, and 1s using the `backColor` value.

That means that the following code

```
$bitmap-> color(0);
$bitmap-> line(0,0,100,100);
$target-> color(cl::Green);
$target-> put_image(0,0,$bitmap);
```

produces a green line on `$target`.

When using monochrome bitmaps for logical operations, note that the target colors should not be explicit 0 and 0xffffff, nor `cl::Black` and `cl::White`, but `cl::Clear` and `cl::Set` instead. The reason is that on paletted displays, the system palette may not necessarily contain the white color under palette index (2^ScreenDepth-1). `cl::Set` thus signals that the value should be "all ones", no matter what color it represents because it will be used for logical operations.

### Fonts

Prima maintains its own font naming convention which usually does not conform to the system's. Since Prima's goal is interoperability, it might be so that some system fonts would not be accessible from within the toolkit.

Prima::Drawable provides property `::font` that accepts/returns a hash, that represents the state of a font in the system graphic context. The font hash keys that are acceptable on the set-call are:

**name**

The font name string. If there is no such font, a default font name is used. To select the default font, a 'Default' string can be passed with the same result ( unless the system has a font named 'Default', of course).

**height**

An integer value from 1 to MAX_INT. Specifies the desired extent of a font glyph between descent and ascent lines in pixels.

**size**

An integer value from 1 to MAX_INT. Specifies the desired extent of a font glyph between descent and internal leading lines in points. The relation between `size` and `height` is

$$size = \frac{height - internal\_leading}{resolution} * 72.27$$

That differs from some other system representations: Win32, for example, rounds 72.27 constant to 72.

**width**

An integer value from 0 to MAX_INT. If greater than 0, specifies the desired extent of a font glyph width in pixels. If 0, sets the default ( designed ) width corresponding to the font size or height.

**style**

A combination of `fs::` ( font style ) constants. The constants

```
fs::Normal
fs::Bold
fs::Thin
fs::Italic
fs::Underlined
fs::StruckOut
fs::Outline
```

can be OR-ed together to express the font style. fs::Normal equals 0 and is usually never used. If some styles are not supported by a system-dependent font subsystem, they are ignored.

**pitch**

One of three constants:

```
fp::Default
fp::Fixed
fp::Variable
```

`fp::Default` specifies no interest in the font pitch selection. `fp::Fixed` is set when a monospaced (all glyphs are of the same width) font is desired. `fp::Variable` pitch specifies a font with different glyph widths. This key is of the highest priority; all other keys may be altered for the consistency of the pitch key.

**vector**

One of three constants:

```
fv::Default
fv::Bitmap
fv::Outline
```

`fv::Default` specifies no interest in the font type selection, `fv::Bitmap` sets the priority for the bitmap fonts, and `fv::Outline` for the vector fonts.

Additionally, font entries returned from the `fonts` method may set the `vector` field to `fv::ScalableBitmap`, to distinguish a bitmap font face that comes with predefined bitmap sets from a scalable font.

**direction**

A counter-clockwise rotation angle - 0 is default, 90 is pi/2, 180 is pi, etc. If a font cannot be rotated, it is usually substituted with the one that can.

**encoding**

A string value, one of the strings returned by `Prima::Application::font_encodings`. Selects the desired font encoding; if empty, picks the first matched encoding, preferably the locale set up by the user.

The encodings provided by different systems are different; in addition, the only encodings are recognizable by the system, that are represented by at least one font in the system.

Unix systems and the toolkit PostScript interface usually provide the following encodings:

```
iso8859-1
iso8859-2
... other iso8859 ...
fontspecific
```

Win32 returns the literal strings like

```
Western
Baltic
Cyrillic
Hebrew
Symbol
```

A hash that `::font` returns, is a tied hash, whose keys are also available as separate properties. For example,

```
$x = $d-> font-> {style};
```

is equivalent to

```
$x = $d-> font-> style;
```

While the latter gives nothing but the arguable coding convenience, its usage in set-call is much more usable:

```
$d-> font-> style( fs::Bold);
```

instead of

```
my %temp = %{$d-> font};
$temp{ style} = fs::Bold;
$d-> font( \%temp);
```

The properties of a font-tied hash are also accessible through the set() call, like in Prima::Object:

```
$d-> font-> style( fs::Bold);
$d-> font-> width( 10);
```

is an equivalent to

```
$d-> font-> set(
   style => fs::Bold,
   width => 10,
);
```

When get-called, the `::font` property returns a hash where more entries than those described above can be found. These keys are read-only, their values are ignored if passed to `::font` in a set-call.

To query the full list of fonts available to a graphic device, the `::fonts` method is used. This method is not present in the Prima::Drawable namespace; it can be found in two built-in class instances, `Prima::Application` and `Prima::Printer`.

`Prima::Application::fonts` returns metrics for the fonts available to the screen device, while `Prima::Printer::fonts` ( or its substitute Prima::PS::Printer ) returns fonts for the printing device. The result of this method is an array of font metrics, fully analogous to these returned by the `Prima::Drawable::font` method.

**family**

A string with font family name. The family is a secondary string key, used for distinguishing between fonts with the same name but of different vendors ( for example, Adobe Courier and Microsoft Courier).

**ascent**

Number of pixels between a glyph baseline and descent line.

**descent**

Number of pixels between a glyph baseline and descent line.

**internalLeading**

Number of pixels between ascent and internal leading lines. Negative if the ascent line is below the internal leading line.

**externalLeading**

Number of pixels between ascent and external leading lines. Negative if the ascent line is above the external leading line.

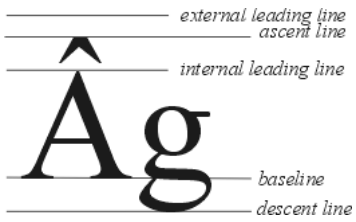



Figure 3.2: Horizontal font measurements

**weight**

Font designed weight. Can be one of

```
fw::UltraLight
fw::ExtraLight
fw::Light
fw::SemiLight
fw::Medium
fw::SemiBold
fw::Bold
fw::ExtraBold
fw::UltraBold
```

constants.

**maximalWidth**

The maximal extent of a glyph in pixels. Equals to **width** in monospaced fonts.

**xDeviceRes**

Designed horizontal font resolution in dpi.

**yDeviceRes**

Designed vertical font resolution in dpi.

**firstChar**

Index of the first glyph present in a font.

**lastChar**

Index of the last glyph present in a font.

**breakChar**

Index of the default character used to divide words. In a typical Western language font it is 32, the ASCII space character.

**defaultChar**

Index of a glyph that is drawn instead of a nonexistent glyph if its index is passed to the text drawing routines.

**underlinePosition**

Position below baseline where to draw underscore line, in pixels. Is negative.

**underlineThickness**

Pixel width of the underscore line

### Font ABC metrics

Besides these characteristics, every font glyph has an ABC metric, the three integer values that describe the horizontal extents of a glyph's black part relative to the glyph extent:

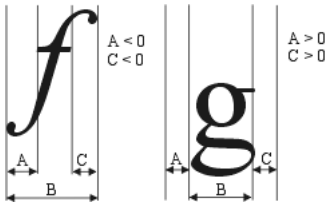



Figure 3.3: Vertical glyph measurements

A and C are negative, if a glyph 'hangs' over its neighbors, as shown in the picture on the left. A and C values are positive if a glyph contains empty space in front or behind the neighbor glyphs, like in the picture on the right. As can be seen, B is the width of a glyph's black part.

ABC metrics are returned by the get_font_abc() method.

The corresponding vertical metrics, called `DEF` metrics, are returned by the get_font_def() method.

### Raster operations

The Prima::Drawable class has two raster operation properties: `::rop` and `::rop2`. These define how the graphic primitives are plotted. `::rop` deals with the foreground color drawing, and `::rop2` with the background.

**Universal ROPs** The toolkit defines the following operations:


Figure 3.4: Classic raster operations

Usually, however, graphic devices support only a small part of the above set, limiting `::rop` to the most important operations: Copy, And, Or, Xor, NoOp. `::rop2` is usually even more restricted, and supports only Copy and NoOp.

The raster operations apply to all graphic primitives except SetPixel.

Note for layering: using layered images and device bitmaps with `put_image` and `stretch_image` can only use `rop::SrcCopy` and `rop::Blend` raster operations on OS-provided surfaces. See more on `rop::Blend` below.

Also, the `rop::AlphaCopy` operation is available for accessing alpha bits only. When used, the source image is treated as an alpha mask, and therefore it has to be grayscale. It can be used to apply the alpha bits independently, without the need to construct an Icon object.

**rop::Blend** `rop::Blend` is the same as `rop::SrcOver` except the source pixels are assumed to be already premultiplied with the source alpha. This is the default raster operation when drawing with 8-bit icon masks or on layered surfaces, and it reflects the expectation of the OS that images come premultiplied.

This is the only blending operator supported on the widgets and bitmaps, and it is advised to premultiply image pixels before drawing images using it with a call to the `Image.premultiply_alpha` method. The core image drawing supports this operator in case these premultiplied images are to be used not only on native surfaces but also on Prima images.

**Default raster operations** The `put_image`, `stretch_image`, and `put_image_indirect` methods are allowed to be called without explicitly specifying the ROP to be used. In this case, the *default ROP*, that depends on the source drawable, will be used. In the majority of cases the default ROP is `rop::CopyPut`, however, when drawing using layered device bitmaps and icons with 8-bit alpha masks, `rop::Blend` is used instead. This is a sort of a DWIM behavior.

This effect is achieved by translating the `rop::Default` constant via the `get_effective_rop` method, that can be used to detect what is the preferred raster operation for a source drawable. `rop::Default` can only be used in the aforementioned there methods, not in the `rop` property, not anywhere else.

**Additional ROPs** Prima core imaging supports extra features for compositing images outside the begin_paint/end_paint brackets. It supports the following 12+1 Porter-Duff operators and some selected Photoshop blend operators:


Figure 3.5: Porter-Duff operators

**Transparency control**

There is defined a set of constants to apply a constant source and destination alpha to when there is no alpha channel available:

```
rop::SrcAlpha
rop::SrcAlphaShift
rop::DstAlpha
rop::DstAlphaShift
```

Combine the rop constant using this formula:

```
$rop = rop::XXX |
   rop::SrcAlpha | ( $src_alpha << rop::SrcAlphaShift ) |
   rop::DstAlpha | ( $dst_alpha << rop::DstAlphaShift )
```

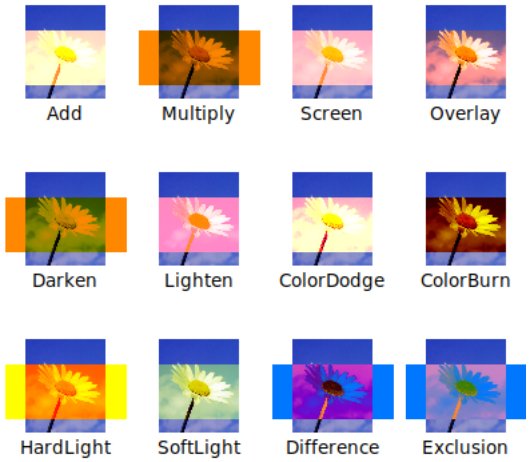



Figure 3.6: Photoshop operators

or by calling `rop::alpha($rop, [ $src_alpha, [ $dst_alpha ]])` that does the same.

Also, the function `rop::blend($alpha)` creates a rop constant for the simple blending of two images by the following formula:

```
$dst = ( $src * $alpha + $dst * ( 255 - $alpha ) ) / 255
```

`rop::alpha` also can be used for drawing on images outside begin_paint/end_paint with all Porter-Duff and Photoshop raster operators:

```
$image->rop( rop::alpha( rop::SrcOver, 128 ));
$image->ellipse( 5, 5, 5, 5 );
```

Note that when the raster operation is set to `rop::SrcOver`, the fully identical effect can be achieved by

```
$image->alpha(128);
$image->ellipse( 5, 5, 5, 5 );
```

as well, in a DWIM fashion. The only corner case here is when `$image->alpha` is 255; add the rop flags `rop::DstAlpha | ( 255 < rop::DstAlphaShift )` to make sure that blending is selected. The corner case happens because the values of the Porter-Duff and bitwise rops clash, as the design was to make the values of `rop::CopyPut` and `rop::Blend` to be the same to serve as a sensible default. This may be resolved in future.

When used with icons, their source and/or destination alpha channels are additionally multiplied by these values.

**rop::ConstantColor**

This bit is used when the alpha is defined but the main bits aren't. In this case, the main bits are filled from the destination's image color, and the source image is treated as the source alpha channel. The following code applies a solid green shade with a mask loaded from a file.

```
$src->load('8-bit gray mask.png');
$dst->color(cl::LightGreen);
$dst->put_image( 0,0,$src,rop::SrcOver | rop::ConstantColor);
```

### Coordinates

The Prima toolkit employs the XY grid where X ascends rightwards and Y ascends upwards. There, the (0,0) location is the bottom-left pixel of a canvas.

All graphic primitives use inclusive-inclusive boundaries. For example,

```
$d-> bar( 0, 0, 1, 1);
```

plots a bar that covers 4 pixels: (0,0), (0,1), (1,0) and (1,1).

The coordinate origin can be shifted using the `::matrix` property that translates the (0,0) point to the given offset. Calls to `::matrix`, `::clipRect` and `::region` always use the 'physical' (0,0) point, whereas the plotting methods use the transformation result, the 'logical' (0,0) point.

As noted before, these three properties cannot be used when an object is in the disabled state.

**Matrix** The `Prima::Drawable` class accepts matrix scalars in the form of 6-item arrays that constitute the following 2-D transformation matrix

```
A B DX
C D DY
0 0 1
```

where the coordinate transformation follows this formula:

```
x' = A x + B x + DX
y' = C y + D y + DY
```

There are two accessors, `get_matrix` and `set_matrix` that return a copy of the current matrix or copy a new matrix, correspondingly, from and to the drawable object.

The 6-item array that `get_matrix` returns is also a blessed object of type `Prima::matrix` ( see the **Prima::matrix** entry in the *Prima::types* section ) that is a separate copy of the object matrix. `Prima::matrix` objects feature a set of operations such as `rotate` and `translate` that transform the matrix further. To apply this matrix, one calls `$drawable->set_matrix($matrix)` or `$matrix->apply($drawable)`.

There is also a property `matrix` that is not orthogonal to the `get_matrix`/ `set_matrix` method pair, as `matrix` returns a `Prima::Matrix` object, that, contrary to a `Prima::matrix` object returned by `get_matrix`, is a direct accessor to the drawable's matrix:

```
my $m = $self->get_matrix;
$m->translate(1,2);
$self->set_matrix($m); # or $self->matrix($m);
```

is the same as

```
$self->matrix->translate(1,2);
```

and does not need an explicit call to `set_matrix` or `apply`.

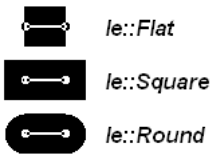
This class has all the methods the `Prima::matrix` has. See more in the **Prima::Matrix** entry in the *Prima::types* section. See also: the *reset_matrix* entry.

### Custom line end styles

Prima uses its own plotting mechanism in the *Prima::Drawable::Path* section to produce all primitive shapes, including lines. The way lines are plotted is managed by the properties `lineEnd`, `lineJoin`, `linePattern`, and `miterLimit`. All of these properties are described below, however, `lineEnd` provides rich capabilities to generate custom line ends, and this is what is described in this section.

**le:: namespace**

There are 3 predefined `le::` integer constants




that are hardcoded in Prima, that are accepted by the `lineEnd` property. These constants are somewhat limited in the way one can operate them, for the sake of efficiency.

It is also possible to supply an array descriptor that defines a set of primitives that plot a line cap, automatically transformed with respect to the current line tangent and width. Details of the descriptor syntax itself are given in the next section, while here the set of non-hardcoded line end functions is listed, which is also suitable for use in the `lineEnd` property.

```
le::Arrow
le::Cusp
le::InvCusp
le::Knob
le::Rect
le::RoundRect
le::Spearhead
le::Tail
```

These descriptors could be arbitrarily transformed (f ex scaled):

```
# all three lines produce the same effect
$x->lineEnd( le::scale( Arrow => 1.5 ) );
$x->lineEnd( le::transform( le::Arrow, [1.5,0,0,1.5,0,0] ) );
$x->lineEnd( le::transform( le::Arrow, Prima::matrix->new->scale(1.5) ) );
```

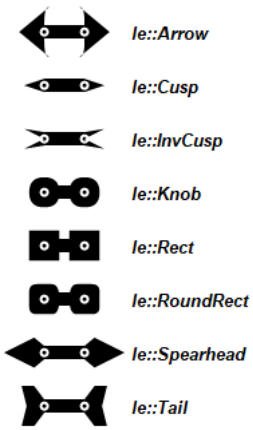
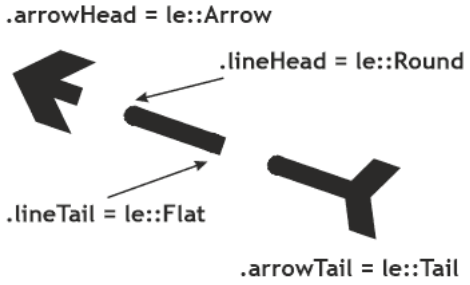



Figure 3.7: Line ends with x 1.5 scaling

### lineEnd accessors

The `lineEnd` property can accept up to 4 line end definitions, depending on syntax. When Prima plots lines, depending on the location of a line end, one of these 4 definitions is selected. Each definition also has its own property - see the *lineHead* entry, the *lineTail* entry, the *arrowHead* entry, and the *arrowTail* entry, or an index-based accessor `lineEndIndex`:




### Line end syntax

The syntax for the custom line end style is:

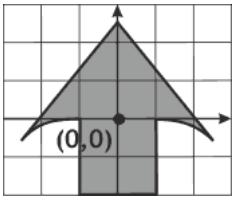
```
syntax  ::= array of pairs
pair    ::= command, array of points
command ::= line or conic or cubic
points  ::= set of x,y coordinate pairs (vertices)
```

where individual commands accept an arbitrary number of vertices, but no less than 1 for `line`, 2 for `conic`, and 3 for `cubic`.

The commands are executed sequentially on a coordinate grid where the line width is assumed to be 1, the plotting starts from the point at (1,0) and ends at (-1,0).

For example, this is the definition of `le::Arrow`:

```
[
   conic => [1,0,2.5,0,2.5,-0.5],
   line  => [0,2.5],
   conic => [-2.5,-0.5,-2.5,0,-1,0]
]
```




## API

### Graphic context properties

### alpha INTEGER

Sets the alpha component of the brush, where 0 is fully transparent and 255 is fully opaque.

Note that premultiplication of `color` and `backColor` is not necessary as it is done internally.

Default: 255

**antialias BOOLEAN**

Turns on and off antialiased drawing on all primitives, excluding image, `pixel`, and `bar_alpha` calls.

It will not be possible to turn the property on if the system does not support it. Also, monochrome images won't support it as well.

See the *Antialiasing and alpha* entry.

**arrowHead**

Defines the style to paint line heads on starting or ending points used to define a line or polygon. Is never used on closed shapes. If `undef`, the heads are painted with the same style as `lineHead`

Default value: `le::Round`. Cannot be `undef`.

**arrowTail**

Defines the style to paint line tails on starting or ending points used to define a line or a polygon. Is never used on closed shapes. If `undef`, the heads are painted with the same style as `lineTail`; if it is also `undef`, then the style of `lineHead` is used.

Default value: `undef`

**backColor COLOR**

Sets background color to the graphic context. All drawing routines that use non-solid or transparent fill or line patterns use this property value.

**color COLOR**

Sets foreground color to the graphic context. All drawing routines use this property value.

**clipRect X1, Y1, X2, Y2**

Selects the clipping rectangle corresponding to the physical canvas origin. On get-call, returns the extent of the clipping area, if it is not rectangular, or the clipping rectangle otherwise. The code

```
$d-> clipRect( 1, 1, 2, 2);
$d-> bar( 0, 0, 1, 1);
```

thus affects only one pixel at (1,1).

Set-call discards the previous `::region` value.

Note: `::clipRect` can not be used while the object is in the paint-disabled state, its context is neither recorded nor used as a template ( see the *Graphic context and canvas* entry) -- except on images.

**fillMode INTEGER**

Affects the filling style of complex polygonal shapes filled by `fillpoly`. If `fm::Winding`, the filled shape contains no holes; if `fm::Alternate`, holes are present where the shape edges cross.

The `fm::Overlay` flag can be combined with these to counter an intrinsic defect of filled shapes both in Win32 and X11 that don't exactly follow polygon vertices. When supplied, it overlays a polygon over the filled shape, so that the latter falls exactly in the boundaries defined by vertices. This is desirable when one wants the shape to be defined exactly by polygon vertices but is not desirable when a shape has holes in it and is connected in a way that the polygon overlay may leave visible connection edges over them.


Default value: `fm::Winding`|`fm::Overlay`

**fillPattern ( [ @PATTERN ] ) or ( fp::XXX ) or IMAGE**

Selects 8x8 fill pattern that affects primitives that plot filled shapes: bar(), fill_chord(), fill_ellipse(), fillpoly(), fill_sector(), floodfill().

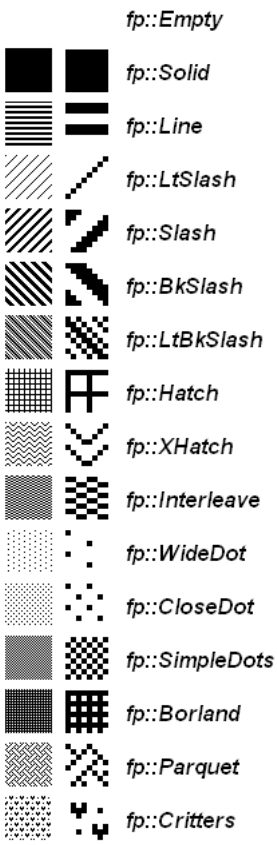
Accepts either a `fp::` constant or a reference to an array of 8 integers, or an image reference. In all cases, except where the image is colored, treats 0s in the pattern as the currently selected `backColor`, and 1s as `color`. When `rop2` is set to `rop::NoOper`, treats 0s as fully transparent pixels. Additionally, when in the render mode respects the `alpha` property.

Note: Drawing over a monochrome bitmap or image will not respect its `rop` on Win32.

Depending on the parameters, treats the input as follows:

**fp:: constant**

There are some predefined patterns, that can be referred to via `fp::` constants:




( the actual patterns are hardcoded in api/api.c ) The default pattern is fp::Solid.

On a get-call, does not return the `fp::` value but the corresponding array (see below). If a constant value is needed to be recovered, use the `fp::builtin` function that would return the constant from the array. There are also two shortcut functions, `fp::is_solid` and `fp::is_empty` that check if the fill pattern is all-ones or all-zeros, correspondingly.

**Array**

Wants an 8-item array where each item is a byte value, representing 8 bits of each line in a pattern. The first integer is the topmost pattern line, and the bit 0x80 is the leftmost pixel in the line.

An example below shows the encoding of the fp::Parquet pattern:

```
# 76543210
  84218421  Hex

0  $ $   $  51
1   $   $   22
2    $ $ $  15
3 $   $     88
4  $   $ $  45
5   $   $   22
6  $ $ $    54
7 $   $     88
```

```
$d-> fillPattern([ 0x51, 0x22, 0x15, 0x88, 0x45, 0x22, 0x54, 0x88 ]);
```

**Monochrome image**

Like the *array* above, wants an image consisting of 0s and 1s where these would represent the target canvas' `backColor` and `color` property values, correspondingly, when rendered. In the same fashion, when `rop2` is set to `rop::NoOper`, zeros will be treated as transparent pixels.

**Color image**

Ignores color, and backColor, and rop2. Just uses the tiles and the current `alpha` value.

**Icon**

Ignores color, backColor, and rop2. Uses the alpha pixel values from the icon's mask and the current `alpha` value.

**fillPatternOffset X, Y**

Origin coordinates for the `fillPattern`. Image patterns origin (0,0) is system-dependent.

**font \%FONT**

Manages font context. FONT hash acceptable values are `name`, `height`, `size`, `width`, `style` and `pitch`.

Synopsis:

```
$d-> font-> size( 10);
$d-> font-> name( 'Courier');
$d-> font-> set(
  style => $x-> font-> style | fs::Bold,
  width => 22
);
```

See the *Fonts* entry for the detailed descriptions.

Applies to text_out(), get_text_width(), get_text_box(), get_font_abc(), get_font_def(), render_glyph().

**font_mapper**

Returns a font mapper object that provides the interface to a set of fonts used for substitution in polyfont shaping (see the *text_shape* entry). The fonts can be accessed there either by their font hash (name and style only, currently), or the font's corresponding index.

There are two font lists used in the substitution mechanism, *passive* and *active*. The passive font list is initiated during the toolkit start and is never changed. Each font there is addressed by an index. When the actual search for a glyph is initiated, these fonts are queried in the loop and are checked if they contain the required glyphs. These queries are also cached so that next time lookups run much quicker. That way, an `active` font list is built, and the next substitutions use it before trying to look into the passive list. Since the ordering of fonts is system-based and is rather random, some fonts may not be a good or aesthetic substitution. Therefore the mapper can assist in adding or removing particular fonts to the active list, potentially allowing an application to store and load a user-driven selection of substitution fonts.

The following methods are available on the font mapper object:

**activate %FONT**

Adds the FONT into the active substitution list if the font is not disabled

**get INDEX**

Returns the font hash registered under INDEX

**count**

Returns the number of all fonts in the collection

**index**
: Returns what index is assigned to the currently used font, if any

**passivate %FONT**
: Remove the font from the active substitution list

**is_active %FONT**
: Returns whether the font is in the active substitution list or not

**enable %FONT**
: Enables the font entry, the font will be considered for the polyfont lookup

**disable %FONT**
: Disables the font entry, the font will not be considered for the polyfont lookup

**is_enabled %FONT**
: Returns whether the font is enabled or not

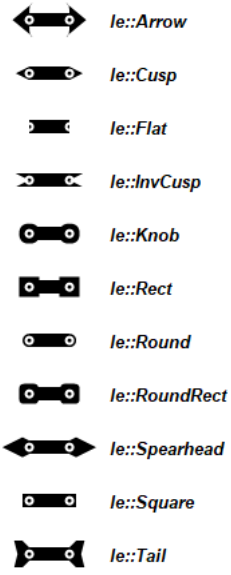
**lineHead**

Defines the style to paint line heads in patterned lines only, on line segments that do not lie on line starting or ending points used to define a line or polygon.

Default value: `undef`

**lineEnd VALUE**

Selects a line ending cap for plotting primitives. VALUE can be one of




constants, `undef`, a custom line end description, or an array of four where each entry is one of the values above.

The `undef` value is only accepted in the array syntax, and not for the index 0 (the *lineHead* entry). The other indexes behave differently if are set to `undef` - see more in the *lineTail* entry, the *arrowHead* entry, the *arrowTail* entry, and the *lineEndIndex* entry.

`le::Round` is the default value.

See also: the *Custom line end styles* entry.

**lineEndIndex INDEX, VALUE**

Same as `lineEnd` except only addresses a single line ending style.

Allows VALUE to be `undef` for indexes greater than 0; depending on the index, the line style will be different ( see more in the *lineTail* entry, the *arrowHead* entry, the *arrowTail* entry ).

Allows special INDEX values or'ed with `le::Only`, that behave differently, if the line end style is `undef`: while normal INDEX queries may return `undef`, or possibly affect neighbor indexing (if these are, in turn, `undef`s), the calls with the `le::Only` bit set never return `undef` on get-calls, and never affect neighbor styles on set-calls.

The following lookup rules are used if a line end style is undef:

```
lei::LineTail  - can never be undef
lei::LineHead  - if undef, same as lei::LineTail
lei::ArrowTail - if undef, same as lei::LineTail
lei::ArrowHead - if undef, same as lei::LineHead, and if it also is undef, then same as le
```

**lineJoin VALUE**

Selects a line joining style for polygons. VALUE can be one of


lj::Round


lj::Bevel


lj::Miter

constants. lj::Round is the default value.

**linePattern PATTERN**

Selects a line pattern for plotting primitives. PATTERN is either a predefined `lp::` constant, or a string where each even byte is the length of a dash, and each odd byte is the length of a gap.

The predefined constants are:


lp::DashDot


lp::DotDot


lp::Solid


lp::ShortDash


lp::LongDash


lp::Dash


lp::Dot


lp::DashDotDot

lp::Null

Not all systems are capable of accepting user-defined line patterns and in such a situation the `lp::` constants are mapped to the system-defined patterns. In Win9x, for example, lp::DashDotDot is much different from its string definition. This however is only actual for lines with width=0, as wider lines are rendered by the Prima internal code.

The default value is lp::Solid.

**lineTail**

Defines the style to paint line tails in patterned lines only, on line segments that do not lie on line starting or ending points used to define a line or polygon. If `undef`, line tails are painted with the same style as `lineHead`.

Default value: `undef`

**lineWidth WIDTH**

Selects a line width for plotting primitives when `antialias` is 0. If a VALUE is 0, then a *cosmetic* pen is used - the thinnest possible line that a device can plot. If a VALUE is greater than 0, then a *geometric* pen is used - the line width is set in device units. There is a subtle difference between VALUE 0 and 1 in the way the lines are joined.

When `antialias` is 1, the geometric plotting algorithm is always used.

Default value: 0

**matrix [A,B,C,D,X,Y] | Prima::Matrix**

Sets current matrix transformation that is used in all plotting operations except `clipRect` and `region`. Returns a the **Matrix** entry in the *Prima* section object.

The default value is (1,0,0,1,0,0) or `Prima::matrix::identity`.

Note: `::matrix` can not be used while the object is in the paint-disabled state, its context is neither recorded nor used as a template ( see the *Graphic context and canvas* entry).

See also: the **Prima::matrix** entry in the *Prima::types* section and `Prima::types/Prima::Matrix`.

**miterLimit VALUE**

When path segments connect at a sharp angle, a miter join results in a spike that extends well beyond the connection point. The purpose of the miter limit is to cut off such spikes when they become objectionably long. At any given corner, the miter length is the distance from the point at which the inner edges of the stroke intersect to the point at which the outside edges of the strokes intersect -- in other words, the diagonal length of the miter. This distance increases as the angle between the segments decreases. If the ratio of the miter length to the line width exceeds the miter limit parameter, stroke treats the corner with a bevel join instead of a miter join. The ratio of miter length to line width is directly related to the angle j between the segments by the formula:

```
r = 1/sin(j/2)
```

Default value: 10.0

Assuming the line join is `lj::Miter` and the line angle is 30 degrees:

**palette [ @PALETTE ]**

Requests to install solid colors into the system palette, as many as possible. PALETTE is an array of integer triplets, where each is the R, G, and B components. The call

```
$d-> palette([128, 240, 240]);
```

selects a gray-cyan color, for example.

The return value from the get-call is the content of the previous set-call, not the actual colors that were copied to the system palette.

**region OBJECT**

Selects a clipping region applied to all drawing and painting routines. In the set-call, the OBJECT is either undef, then the clip region is erased ( no clip), or a Prima::Image object with a bit depth of 1, or a `Prima::Region` object. The bit mask of the OBJECT is applied to the system clipping region. If the OBJECT is smaller than the drawable, its exterior is assigned to the clipped area as well. Discards the previous `::clipRect` value; successive get-calls to `::clipRect` return the boundaries of the region.

In the get-mode returns either undef or a Prima::Region object.

Note: `::region` can not be used while the object is in the paint-disabled state, its context is neither recorded nor used as a template ( see §3.4).

**resolution X, Y**

A read-only property. Returns horizontal and vertical device resolution in dpi.

**rop OPERATION**

Selects raster operation that applies to foreground color plotting routines.

See also: `::rop2`, the *Raster operations* entry.

**rop2 OPERATION**

Selects raster operation that applies to background color plotting routines.

See also: `::rop`, the *Raster operations* entry.

**textOpaque FLAG**

If FLAG is 1, then text_out() fills the text background area with the `::backColor` property value before drawing the text. The default value is 0 when text_out() plots text only.

In the system-based text drawing, if the background area is filled, then the alpha value is ignored. In the standalone text drawing, if the background area is filled, then the alpha value is not ignored.

See get_text_box().

**textOutBaseline FLAG**

If FLAG is 1, then text_out() plots text on a given Y coordinate correspondent to font baseline. If FLAG is 0, a Y coordinate is mapped to the font descent line. The default value is 0.

**translate X_OFFSET, Y_OFFSET**

Translates the origin point by X_OFFSET and Y_OFFSET. Does not affect `::clipRect` and `::region`. Not cumulative, so the call sequence

```
$d-> translate( 5, 5);
$d-> translate( 15, 15);
```

is equivalent to

```
$d-> translate( 15, 15);
```

Note: `::translate` can not be used while the object is in the paint-disabled state, its context is neither recorded nor used as a template ( see the *Graphic context and canvas* entry).

### Other properties

**height HEIGHT**

Selects the height of a canvas.

**size WIDTH, HEIGHT**

Selects the extent of a canvas.

**width WIDTH**

Selects the width of a canvas.

### Graphic primitives methods

**arc X, Y, DIAMETER_X, DIAMETER_Y, START_ANGLE, END_ANGLE**

Plots an arc with the center in X, Y, and DIAMETER_X and DIAMETER_Y axes from START_ANGLE to END_ANGLE.

Context used: color, backColor, lineEnd, linePattern, lineWidth, miterLimit, rop, rop2

**bar X1, Y1, X2, Y2**

Draws a filled rectangle

Context used: color, backColor, fillPattern, fillPatternOffset, rop, rop2

**bar_alpha ALPHA <X1, Y1, X2, Y2>**

Fills a rectangle in the alpha channel bits only, using the ALPHA value between (0-255). Can be called without parameters, in this case, fills the whole canvas.

Has only effect on the layered surfaces.

**bars @RECTS**

Draws a set of filled rectangles. RECTS is an array of integer quartets in the format (X1,Y1,X2,Y2).

Context used: color, backColor, fillPattern, fillPatternOffset, rop, rop2

**chord X, Y, DIAMETER_X, DIAMETER_Y, START_ANGLE, END_ANGLE**

Plots an arc with the center in X, Y, and DIAMETER_X and DIAMETER_Y axes from START_ANGLE to END_ANGLE and connects its ends with the straight line.

Context used: color, backColor, lineEnd, linePattern, lineWidth, miterLimit, rop, rop2

**clear <X1, Y1, X2, Y2>**

Draws a rectangle filled with background color. Can be called without parameters, in this case, fills the whole canvas.

Context used: backColor, rop2

**draw_text CANVAS, TEXT, X1, Y1, X2, Y2, [ FLAGS = dt::Default, TAB_INDENT = 1 ]**

Draws several lines of text one under another with respect to align and break rules, specified in FLAGS and TAB_INDENT tab character expansion.

`draw_text` is a convenience wrapper around `text_wrap` for drawing the wrapped text, and also provides the tilde ( ~ )- character underlining support.

The FLAGS is a combination of the following constants:

```
dt::Left                - text is aligned to the left boundary
dt::Right               - text is aligned to the right boundary
dt::Center              - text is aligned horizontally in the center
dt::Top                 - text is aligned to the upper boundary
dt::Bottom              - text is aligned to the lower boundary
dt::VCenter             - text is aligned vertically in the center
dt::DrawMnemonic        - tilde-escapement and underlining is used
dt::DrawSingleChar      - sets tw::BreakSingle option to
                          Prima::Drawable::text_wrap call
dt::NewLineBreak        - sets tw::NewLineBreak option to
                          Prima::Drawable::text_wrap call
dt::SpaceBreak          - sets tw::SpaceBreak option to
                          Prima::Drawable::text_wrap call
dt::WordBreak           - sets tw::WordBreak option to
                          Prima::Drawable::text_wrap call
dt::ExpandTabs          - performs tab character ( \t ) expansion
dt::DrawPartial         - draws the last line, if it is visible partially
dt::UseExternalLeading  - text lines positioned vertically with respect to
                          the font external leading
dt::UseClip             - assign ::clipRect property to the boundary rectangle
dt::QueryLinesDrawn     - calculates and returns the number of lines drawn
                          ( contrary to dt::QueryHeight )
dt::QueryHeight         - if set, calculates and returns vertical extension
                          of the lines drawn
dt::NoWordWrap          - performs no word wrapping by the width of the boundaries
dt::WordWrap            - performs word wrapping by the width of the boundaries
dt::Default             - dt::NewLineBreak|dt::WordBreak|dt::ExpandTabs|
                          dt::UseExternalLeading
```

Context used: color, backColor, font, rop, textOpaque, textOutBaseline

**ellipse X, Y, DIAMETER_X, DIAMETER_Y**

Plots an ellipse with the center in X, Y, and DIAMETER_X and DIAMETER_Y axes.

Context used: color, backColor, linePattern, lineWidth, rop, rop2

**fill_chord X, Y, DIAMETER_X, DIAMETER_Y, START_ANGLE, END_ANGLE**

Fills a chord outline with the center in X, Y, and DIAMETER_X and DIAMETER_Y axes from START_ANGLE to END_ANGLE (see chord()).

Context used: color, backColor, fillPattern, fillPatternOffset, rop, rop2

**fill_ellipse X, Y, DIAMETER_X, DIAMETER_Y**

Fills an elliptical outline with the center in X, Y, and DIAMETER_X and DIAMETER_Y axes.

Context used: color, backColor, fillPattern, fillPatternOffset, rop, rop2

**fillpoly \@POLYGON**

Fills a polygonal area defined by POLYGON set of points. POLYGON must present an array of (X,Y) integer pairs. Example:

```
$d-> fillpoly([ 0, 0, 15, 20, 30, 0]); # triangle
```

Context used: color, backColor, fillPattern, fillPatternOffset, rop, rop2, fillMode

Returns success flag; if failed, `$@` contains the error.

See also: polyline().

**fill_sector X, Y, DIAMETER_X, DIAMETER_Y, START_ANGLE, END_ANGLE**

Fills a sector outline with the center in X, Y, and DIAMETER_X and DIAMETER_Y axes from START_ANGLE to END_ANGLE (see sector()).

Context used: color, backColor, fillPattern, fillPatternOffset, rop, rop2

**fill_spline \@VERTICES, %OPTIONS**

Fills a polygonal area defined by the curve projected by applying a B-spline curve based on a set of VERTICES. VERTICES must present an array of (X,Y) integer pairs. Example:

```
$d-> fill_spline([ 0, 0, 15, 20, 30, 0]);
```

Context used: color, backColor, fillPattern, fillPatternOffset, rop, rop2

Returns success flag; if failed, `$@` contains the error.

See also: spline, render_spline

**flood_fill X, Y, COLOR, SINGLEBORDER = 1**

Fills an area of the canvas using the current fill context. The area is assumed to be bounded as specified by the SINGLEBORDER parameter. SINGLEBORDER can be 0 or 1.

SINGLEBORDER = 0: The fill area is bounded by the color specified by the COLOR parameter.

SINGLEBORDER = 1: The fill area is defined by the color that is specified by COLOR.

Filling continues outward in all directions as long as the color is encountered. This style is useful for filling areas with multicolored boundaries.

Context used: color, backColor, fillPattern, fillPatternOffset, rop, rop2

**line X1, Y1, X2, Y2**

Plots the straight line from (X1,Y1) to (X2,Y2).

Context used: color, backColor, linePattern, lineWidth, rop, rop2

**lines \@LINES**

LINES is an array of integer quartets in format (X1,Y1,X2,Y2). lines() plots the straight line per quartet.

Context used: color, backColor, linePattern, lineWidth, rop, rop2

Returns success flag; if failed, `$@` contains the error.

**new_aa_surface**

Returns a new antialiasing surface object for AA emulation. See the *Prima::Drawable::Antialias* section for usage and details.

**new_gradient**

Returns a new gradient object. See the *Prima::Drawable::Gradient* section for usage and details.

**new_path**

Returns a new path object. See the *Prima::Drawable::Path* section for usage and details.

**pixel X, Y, <COLOR>**

::pixel is a property - on set-call it changes the pixel value at (X,Y) to COLOR, on get-call ( without COLOR ) it does return a pixel value at (X,Y).

No context is used except matrix transformation of the coordinates. May return `cl::Invalid` to signal an error or the out-of-boundaries condition.

**polyline \@POLYGON**

Draws a polygonal area defined by the POLYGON set of points. POLYGON must contain an array of integer pairs in (X,Y) format.

Context used: color, backColor, linePattern, lineWidth, lineJoin, lineEnd, miterLimit, rop, rop2

Returns success flag; if failed, `$@` contains the error.

See also: fillpoly().

**put_image X, Y, OBJECT, [ ROP=rop::Default]**

Draws an OBJECT at coordinates (X,Y). OBJECT must be Prima::Image, Prima::Icon, or Prima::DeviceBitmap. If ROP raster operation is specified, it is used. Otherwise, the current value of the `::rop` property is used.

Returns success flag; if failed, `$@` contains the error.

Context used: rop; color and backColor for a monochrome DeviceBitmap

**put_image_indirect OBJECT, X, Y, X_FROM, Y_FROM, DEST_WIDTH, DEST_HEIGHT, SRC_WIDTH, SRC_HEIGHT, [ROP=rop::Default]**

Draws the OBJECT's source rectangle into the destination rectangle, stretching or compressing the source bits to fit the dimensions of the destination rectangle, if necessary. The source rectangle starts at (X_FROM,Y_FROM), and is SRC_WIDTH pixels wide and SRC_HEIGHT pixels tall. The destination rectangle starts at (X,Y), and is abs(DEST_WIDTH) pixels wide and abs(DEST_HEIGHT) pixels tall. If DEST_WIDTH or DEST_HEIGHT are negative, a mirroring by the respective axis is performed.

OBJECT must be Prima::Image, Prima::Icon, or Prima::DeviceBitmap.

No context is used, except color and backColor for a monochrome DeviceBitmap

Returns success flag; if failed, `$@` contains the error.

**rect3d X1, Y1, X2, Y2, WIDTH, LIGHT_COLOR, DARK_COLOR, [ BACK_COLOR ]**

Draws a 3d-shaded rectangle (X1,Y1 - X2,Y2) with WIDTH line width, and LIGHT_COLOR and DARK_COLOR colors. If BACK_COLOR is specified, paints an inferior rectangle with it, otherwise the inferior rectangle is not touched.

Context used: rop; color and backColor for a monochrome DeviceBitmap

**rect_fill X1, Y1, X2, Y2, BORDER_WIDTH, FOREGROUND, BACKGROUND**

Draws a rectangle with outline color FOREGROUND and BORDER_WIDTH pixels, and fills it with color BACKGROUND. If FOREGROUND and/or BACKGROUND are undefined, current colors are used. BORDER_WIDTH is 1 pixel if omitted.

Contrary to a call to `rectangle()` with the line width greater than 1, never paints pixels outside the given rectangle; the border is painted inwards.

Context used: rop, fillPattern

**rect_focus X1, Y1, X2, Y2, [ WIDTH = 1 ]**

Draws a marquee rectangle in boundaries X1,Y1 - X2,Y2 with WIDTH line width.

No context is used.

**rect_solid X1, Y1, X2, Y2, BORDER_WIDTH, FOREGROUND**

Draws a rectangle with outline color FOREGROUND and BORDER_WIDTH pixels. If FOREGROUND is undefined, a current color is used. BORDER_WIDTH is 1 pixel if omitted.

Contrary to a call to `rectangle()` with line width greater than 1, never paints pixels outside the given rectangle; the border is painted inwards.

Context used: rop

**rectangle X1, Y1, X2, Y2**

Plots a rectangle with (X1,Y1) - (X2,Y2) extents.

Context used: color, backColor, linePattern, lineWidth, rop, rop2

**sector X, Y, DIAMETER_X, DIAMETER_Y, START_ANGLE, END_ANGLE**

Plots an arc with the center in X, Y, and DIAMETER_X and DIAMETER_Y axis from START_ANGLE to END_ANGLE and connects its ends and (X,Y) with two straight lines.

Context used: color, backColor, lineEnd, linePattern, lineWidth, miterLimit, rop, rop2

**spline \@VERTICES, %OPTIONS**

Draws a B-spline curve defined by a set of VERTICES control points. VERTICES must present an array of (X,Y) integer pairs.

The extra options `knots` and `weights` described below allow to upgrade the B-spline into a NURBS curve. See the *https:en.wikipedia.orgwikiNon-uniform_rational_B-spline* entry.

The following options are supported:

**closed BOOL = undef**

When not set, checks if the first and the last vertices point to the same point, and assumes a closed shape if they do. Note - a closed shape rendering is implemented by adding a degree minus two points to the set; this is important if `weight` or `knots` are specified.

**degree INTEGER = 2**

The B-spline degree. Default is 2 (quadratic). The number of points supplied must be at least a degree plus one.

**knots \@INTEGERS**

An array of N integers ( N = number of points plus degree plus one ). By default, if the shape is opened (i.e. first and last points are different), represents a clamped array, so that the first and last points of the final curve match the first and the last control points. If the shape is closed, represents an unclamped array so that no control points lie directly on the curve.

Quote wikipedia: "The knot vector is a sequence of parameter values that determines where and how the control points affect the NURBS curve... The knot vector divides the parametric space in the intervals ... usually referred to as knot spans. Each time the parameter value enters a new knot span, a new control point becomes active, while an old control point is discarded. It follows that the values in the knot vector should be in nondecreasing order, so (0, 0, 1, 2, 3, 3) is valid while (0, 0, 2, 1, 3, 3) is not."

**precision INTEGER = 24**

Defines the number of steps to split the curve into. The value is multiplied by the number of points and the result is used as the number of steps.

**weight \@INTEGERS = [ 1, 1, 1, ... ]**

An array of integers, one for each point supplied. Assigning these can be used to convert B-spline into a NURBS. By default set of ones.

Context used: color, backColor, linePattern, lineWidth, lineEnd, miterLimit, rop, rop2

See also: fill_spline, render_spline.

**stretch_image X, Y, DEST_WIDTH, DEST_HEIGHT, OBJECT, [ ROP=rop::Default ]**

Draws the OBJECT on the destination rectangle, stretching or compressing the source bits to fit the dimensions of the destination rectangle, if necessary. If DEST_WIDTH or DEST_HEIGHT are negative, a mirroring is performed. The destination rectangle starts at (X,Y) and is DEST_WIDTH pixels wide and DEST_HEIGHT pixels tall.

If ROP raster operation is specified, it is used. Otherwise, the value of the `::rop` property is used.

OBJECT must be Prima::Image, Prima::Icon, or Prima::DeviceBitmap.

Returns success flag; if failed, `$@` contains the error.

Context used: rop

**text_out TEXT, X, Y**

Draws TEXT string at (X,Y). TEXT is either a character string or a `Prima::Drawable::Glyphs` object returned from `text_shape`, or `Prima::Drawable::Glyphs->glyphs` strings of glyphs.

Returns success flag; if failed, `$@` contains the error.

Context used: color, backColor, font, rop, textOpaque, textOutBaseline

**text_shape TEXT, %OPTIONS**

Converts TEXT into a set of glyphs, returns either a `Prima::Drawable::Glyphs` object, or a `0` integer when shaping is not necessary, or `undef` as an error.

When Prima is compiled with `libfribidi`, the method runs the unicode bidirectional algorithm on TEXT that properly positions embedded directional text (f.ex. a Latin quote inside an Arabic text), see the *Unicode Standard Annex #9 | http:unicode.orgreportstr9tr9-22.html* entry for the details. Without the library only does minimal RTL alignment.

Glyphs returned are positioned according to RTL directions given in TEXT using characters from unicode block "General Punctuation U+2000 .. U+206F". Additionally, character ligation may be performed so that one or more characters are represented by one or more glyphs. Such syntactic units, *clusters*, are adopted in Prima where appropriate, instead of character units, for selection, navigation, etc in f.ex. `Prima::InputLine` and `Prima::Edit`. Helper routines that translate clusters, glyphs, and characters into each other are found in the *Prima::Drawable::Glyphs* section.

Options recognized:

**advances BOOLEAN = false**

The shaping process may or may not fill an integer array of advances and positions for each glyph, depending on the implementation. The advances are needed to represent f.ex. combining graphemes, when TEXT consisting of two characters, `"A"` and combining grave accent U+300 should be drawn as a single À cluster but are represented by

two glyphs `"A"` and `"‘"`. The grave glyph has its own advance for standalone usage, but in this case, it should be ignored, and that is achieved by filling the advance table where the `"A"` advance is the normal glyph advance, whereas the advance of the `"‘"` is zero. Also, the position table additionally shifts glyph position by X and Y coordinates, when that is needed (f.ex. it might be positioned differently by the vertical axis on `"a"` and `"A"`).

Setting these options to `true` will force to fill advance and positioning tables. These tables can be manipulated later, and are respected by `text_out` and `get_text_width`.

**language STRING = undef**

When set, the shaping process can take into account the language of the text. F.ex. text `"ae"` might be shaped as a single glyph *æ* for the Latin language, but never for English.

**level INTEGER = ts::Full**

Selects the shaping (i.e. text to glyph conversion) level, how the system should treat the input text, and how deep the shaping should go.

One of the following `ts::XXX` options:

**ts::Bytes**

Treats input text as non-unicode locale-specific codepoints, characters higher than 255 are treated as chr(255). Reordering never happens, font substitution never happens, kerning and ligation never happen; returns glyph indexes in a 1:1 mapping for each codepoint.

**ts::None**

Performs quick null shaping without mapping to the font glyphs, but only running the bidirectional algorithm on the text. On the return, `glyphs`, as well as eventual `advances` and `positions`, are filled with zeros, but `indexes` are filled with the proper character offsets, effectively making it a visual-to-logical map since the number of glyphs will always be equal to the number of characters in TEXT because ligation never happens here (except when TEXT contains unicode directional characters such as isolates etc - those are removed from the output).

By default, advances and positions are not filled, but if the `advances` option is set, fills them with zeros.

**ts::Glyphs**

Applies the unicode bidi algorithm and maps the result onto font glyphs. Ligation and kerning don't happen here, it's the same as `ts::None` but with the glyph mapping part.

By default, advances and positions are not filled, but if the `advances` option is set, fills the advances array with character glyph advances and the positions array with zeros.

May fill the `fonts` array if the `polyfont` option is set.

**ts::Full**

Applies the unicode bidi algorithm and runs the full shaping on the result. Ligation and kerning may occur. Always fills the advances and positions array; the `advances` option is ignored.

If the system or the selected font does not support shaping, tries to ligate known Arabic shapes using the *fribidi* library, if available. Also in this case does not return the advances and positions by default, but if the `advances` option is set, fills the advances array with character glyph advances and the positions array with zeros.

May fill the `fonts` array if the `polyfont` option is set.

**pitch INTEGER**

When the `polyfont` is set (default) and thus font substitution is desired, filters only fonts that match `pitch`, either `fp::Variable` or `fp::Fixed`. By default will be set to

`fp::Fixed` if the current for is monospaced, but to `fp::Default` matching all fonts, otherwise.

**polyfont BOOLEAN = true**

If set, checks if the currently selected font supports all ithe required unicode points, and if not, selects substitutions from a pre-populated list, taking into account the font pitch (see `pitch` above). In cases where the current font does not have enough glyphs to shape all the requested unicode points, font substitution is performed, and the result contains an extra array `fonts` (see the **fonts** entry in the *Prima::Drawable::Glyphs* section). When the current font has all the needed glyphs, the fonts array is not created.

The font list access is available through the *font_mapper* entry.

Valid only with shaping levels `ts::Glyphs` and `ts::Full`.

**reorder BOOLEAN = true**

When set, the unicode bidi algorithm is used to reorder codepoints, and additionally, RTL codepoints may be reversed (depending on the direction context).

When unset, no such reordering occurs, to emulate as much as possible a behavior that each text grapheme is being mapped to a glyph cluster exactly as it occurs in the input text, from left to right. Note that bidi reordering still may occur internally, since system shapers may reverse the placement of RTL characters, so the Prima reordering is needed to cancel this. In theory the caller shouldn't see the difference as these should cancel each other, but if Prima miscalculates the expected way the system shaper does the bidi processing, it might.

A similar effect can be reached by prepending the text with U+202D (LEFT-TO-RIGHT OVERRIDE).

**replace_tabs INTEGER = -1**

If set to 0 or more, replaces each tab character with the space character and sets their widths to the width of the latter multiplied by the given number. Since it needs the advances table to operate, automatically sets the `advances` option. If the string passed indeed contains tab characters, also turns off the `skip_if_simple` option.

Note: if using the result later in `text_wrap`, set the `tabIndent` parameters there to 1 to avoid double multiplication of the tab character width.

**rtl BOOLEAN**

If set to 1, the default text direction is assumed as RTL, and as LTR if set to 0. If unset, the text direction is taken from the **textDirection** entry in the *Prima::Application* section.

**skip_if_simple BOOLEAN = false**

When set, checks whether the shaping result is identical to the input, in the sense that a call to `text_out(TEXT)` and a call to `text_shape_out(TEXT)` produce identical results. The majority of English text will fall into that category, and when that indeed happens, returns an integer value of `0` instead of a glyph object.

See also `text_shape_out`, `get_text_shape_width`, `text_wrap_shape`.

**text_shape_out TEXT, X, Y[, RTL]**

Runs shaping on TEXT character string with the RTL flag (or `$::application->textDirection`. Draws the resulting glyph string at (X,Y).

Returns success flag; if failed, `$@` contains the error.

Context used: color, backColor, font, rop, textOpaque, textOutBaseline

### Methods

**begin_paint**

Enters the enabled ( active paint ) state and returns the success flag; if failed, `$@` contains the error. Once the object is in the enabled state, painting and drawing methods can write on the canvas.

See also: `end_paint`, `begin_paint_info`, the *Graphic context and canvas* entry

**begin_paint_info**

Enters the information state and returns the success flag; if failed, `$@` contains the error. The object information state is the same as the enabled state ( see `begin_paint`), except painting and drawing methods do not change the object canvas.

See also: `end_paint_info`, `begin_paint`, the *Graphic context and canvas* entry

**can_draw_alpha**

Returns whether using alpha bits operation on the drawable will have any effect or not. Note that the drawable may not necessarily have an alpha channel, for example, a normal RGB image is capable of being painted on with alpha while not having any alpha on its own. On Unix, all non-1-bit drawables return true if Prima was compiled with XRender support and if that extension is present on the X server. On windows, all non-1-bit drawables return true unconditionally.

See also: `has_alpha_layer`

**end_paint**

Exits the enabled state and returns the object to a disabled state.

See also: `begin_paint`, the *Graphic context and canvas* entry

**end_paint_info**

Exits the information state and returns the object to a disabled state.

See also: `begin_paint_info`, the *Graphic context and canvas* entry

**font_match \%SOURCE, \%DEST, PICK = 1**

Performs merging of two font hashes, SOURCE and DEST. Returns the merge result. If PICK is true, matches the result with a system font repository.

Called implicitly by `::font` on set-call, allowing the following example to work:

```
$d-> font-> set( size => 10);
$d-> font-> set( style => fs::Bold);
```

In the example, the hash 'style => fs::Bold' does not overwrite the previous font context ( 'size => 10' ) but gets added to it ( by font_match()), providing the resulting font with both font properties set.

**fonts <FAMILY = "", ENCODING = "">**

Member of `Prima::Application` and `Prima::Printer`, does not present in `Prima::Drawable`.

Returns an array of font metric hashes for a given font FAMILY and ENCODING. Every hash has a full set of elements described in the *Fonts* entry.

If called without parameters, returns an array of the same hashes where each hash represents a member of the font family from every system font set. In this special case, each font hash

contains an additional `encodings` entry, which points to an array of encodings available for the font.

If called with FAMILY parameter set but no ENCODING is set, enumerates all combinations of fonts with all available encodings.

If called with FAMILY set to an empty string, but ENCODING specified, returns only fonts that can be displayed with the encoding.

Example:

```
print sort map {"$_->{name}\n"} @{$::application-> fonts};
```

**get_bpp**

Returns device color depth. 1 is for black-and-white monochrome, 24 for true color, etc.

**get_effective_rop ROP**

Converts a given ROP depending on the drawable type. The majority of cases only convert `rop::Default` to `rop::CopyPut`, however, layered device bitmaps and icons with 8-bit alpha masks return `rop::Blend` instead.

**get_font_abc FIRST_CHAR = -1, LAST_CHAR = -1, UNICODE = 0**

Returns ABC font metrics for the given range, starting at FIRST_CHAR and ending with LAST_CHAR. If these two parameters are both -1, the default range ( 0 and 255 ) is assumed. The UNICODE boolean flag is responsible for the representation of characters in the 127-255 range. If 0, the default, encoding-dependent characters are assumed. If 1, the U007F-U00FF glyphs from the Latin-1 set are used.

The result is an integer array reference, where every character glyph is referred to by three integers, each triplet containing A, B and C values.

For a detailed explanation of ABC meaning, see the *Font ABC metrics* entry;

Context used: font

**get_font_def FIRST_CHAR = -1, LAST_CHAR = -1, UNICODE = 0**

Same as `get_font_abc` but for the vertical metrics. Is expensive on bitmap fonts because in order to find out the correct values Prima has to render glyphs on bitmaps and scan for black and white pixels.

Vector fonts are not subject to this, and the call is as effective as `get_font_abc`.

**get_font_languages**

Returns an array of ISO 639 strings that can be displayed using glyphs available in the currently selected font.

**get_font_ranges**

Returns an array of integer pairs denoting unicode indices of glyphs covered by the currently selected font. Each pair is the first and the last index of a contiguous range.

Context used: font

**get_nearest_color COLOR**

Returns the nearest possible solid color in the representation of the graphic device. Always returns the same color if the device bit depth is equal to or greater than 24.

**get_paint_state**

Returns the paint state value as one of the `ps::` constants - `ps::Disabled` if the object is in the disabled state, `ps::Enabled` for the enabled state, `ps::Information` for the information state.

The `ps::Disabled` constant is equal to 0 so this allows for simple boolean testing whether one can get/set graphical properties on the object.

See the *Graphic context and canvas* entry for more.

**get_physical_palette**

Returns an array of (R,G,B) integer triplets where each color entry is in the 0 - 255 range.

The physical palette array is non-empty only on paletted graphic devices, the true color devices always return an empty array.

The physical palette reflects the solid colors currently available to all programs in the system. The information is volatile if the system palette can change colors, since any other application may request to change the system colors at any moment.

**get_text_shape_width TEXT, [ FLAGS ]**

Runs shaping on TEXT character string with the text direction either taken from the `FLAGS & to::RTL` value or from the `$::application->textDirection` property. Returns the width of the shaping result as if it would be drawn using the currently selected font.

If `FLAGS & to::AddOverhangs` is set, the first character's absolute A value and the last character's absolute C value are added to the string if they are negative.

**get_text_width TEXT, ADD_OVERHANG = 0**

Returns the TEXT string width if it would be drawn using the currently selected font. TEXT is either a character string, or a `Prima::Drawable::Glyphs` object returned from `text_shape`, or a `Prima::Drawable::Glyphs-> glyphs` glyph string.

If ADD_OVERHANG is 1, the first character's absolute A value and the last character's absolute C value are added to the string if they are negative.

See more on ABC values at the *Font ABC metrics* entry.

Context used: font

**get_text_box TEXT**

Returns the TEXT string extensions if it would be drawn using the currently selected font. TEXT is either a character string or a `Prima::Drawable::Glyphs` object returned from the `text_shape` method, or `Prima::Drawable::Glyphs-> glyphs` glyph string.

The result is an anonymous array of 5 points ( 5 integer pairs in (X,Y) format). These 5 points are pixel offsets for the following string extents, given the string is plotted at (0,0):

1: start of string at the ascent line ( top left )

2: start of string at the descent line ( bottom left )

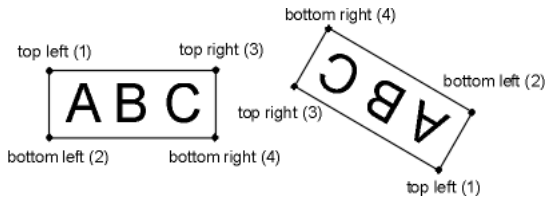
3: end of string at the ascent line ( top right )

4: end of string at the descent line ( bottom right )

5: concatenation point

The concatenation point coordinates (XC,YC) are the values passed to the consequent text_out() call so that the conjoint string would plot as if it was a part of the TEXT. Depending on the value of the `textOutBaseline` property, the concatenation point is located either on the baseline or on the descent line.

Context used: font, textOutBaseline




**graphic_context %GC, $CALLBACK**

A shortcut method that saves the graphic context, applies changes in %GC, calls $CALLBACK, and finally restores the context. F ex:

```
$self->graphic_context( fillPattern => fp::Solid, sub { $self-> bar(..) } );
```

**graphic_context_pop**

Restores the graphic context properties from the stack.

**graphic_context_push**

Saves the graphic context properties on the stack.

**has_alpha_layer**

Returns true if the drawable has an alpha channel. If the drawable is treated as a source, it means its alpha content will be respected when drawing on another surface. If the drawable is treated as a destination, it means that its alpha content will be updated if drawing on it uses alpha bits.

See also: `can_draw_alpha`.

**render_glyph INDEX, %OPTIONS**

Returns a representation of a glyph as an outline. The outline is an integer array formed as a set of plotting commands. Each command is a `ggo::` constant followed by an integer value with the number of the points returned, followed by the 2D point coordinates in 1/64 pixels.

The following options are recognized:

**glyph BOOL**

If set, INDEX is treated as the glyph index rather than the character index. The default value is false.

**hints BOOL**

If set, hinting is enabled. The default value is true.

**unicode BOOL**

If set, INDEX is treated as a utf8 character index, otherwise a locale-specific index. The default value is false.

The `ggo::` commands are:

```
ggo::Move  - move point
ggo::Line  - plot line
ggo::Conic - plot 2-degree spline
ggo::Cubic - plot 3-degree spline
```

**render_pattern IMAGE|ICON|ARRAY|INDEX, %OPTIONS**

Takes a fill pattern represented by one of the `fp::XXX` constants, an array of 8 bytes, or an image (or icon); the same syntax as in `fillPattern`.

Uses %OPTIONS to generate a new rectangular pattern that can be used in the `fillPattern` property. Since Prima does not provide an individual property that would manage specifically the matrix of a fill pattern, this method can be used to implement this functionality.

Also respects the `preserveType` property of the image, and if it is set, changes the resulting pattern image type back to the original type. In case where fillPattern is given by an ARRAY or an INDEX, always generates an `im::BW` image, so it can be used both in `rop2` transparent and opaque modes, like the original pattern.

Options:

**color COLOR, alpha 0-255**

If `margin` is used, pre-fills the target image with this color. `alpha` is used to pre-fill the target image's mask with this value, if the image is an icon.

**margin XY | [ X, Y ]**

Set margins in X and Y pixels before applying the transformation

**matrix MATRIX**

2D matrix to transform IMAGE

**render_polyline \@POLYLINE, %OPTIONS**

Performs calculations on the POLYLINE, defined by OPTIONS. The following options are recognized:

**aafill BOOLEAN**

If set, renders a 8-bit grayscale image with antialiased filled polygon. The polygon is automatically adjusted so its lower and left boundaries and on the resulting image's lower and left boundaries. Also the image size corresponds to the polygon size.

The filling mode can be specificed with the `mode` option, or the current `fillMode` will be used. Note the the option only accepts the `fm::Winding` and `fm::Alternate` constants, and ignores `fm::Overlay`.

Return three scalars: first two are the X and Y polygon offsets, and the first is the image itself.

**box BOOLEAN**

If set, instead of polyline vertices, calculates the box extents of the polyline (either original or after the matrix transform, depending on whether the `matrix` option was supplied or not), and returns 4 numerics for left, bottom, width, and height of the box enclosure.

**integer BOOLEAN**

By default, the result is returned as a set of floating point numerics, however, if integer results are needed, the results are transformed to integers using the *int = float + ((float < 0) ? -0.5 : 0.5)* formula.

**matrix A,B,C,D,U,V**

If supplied, performs matrix transformation on each polyline vertex:

```
X' = AX + CY + U
Y' = BX + DY + V
```

and returns the new polyline

**mode fm::Windings | fm::Alternate**

See `aafill` above.

**path BOOLEAN**

If set, treats polyline as a path that will get applied `lineEnd`, `lineJoin`, `linePattern`, and `miterLimit` properties (either from the object or from %OPTIONS) and returns a set of commands that would represent the final shape. The commands are: *arc* (6 arguments, same as the `arc` primitive), *line* with 1 argument, a polyline array (respects the `integer` option), and *open* with no arguments.

See the **widen** entry in the *Prima::Drawable::Path* section for usage. See also `line_join_hints` below.

**line_join_hints ARRAY OF INTEGERS**

Only when the `path` option is present:

A specially formatted array of indexes that hint where inside the polyline are the boundaries between the points that need to override `lineJoin` and force it to be `lj::Miter`.

See the **widen** entry in the *Prima::Drawable::Path* section for usage.

**render_spline \@VERTICES, %OPTIONS**

Renders B-spline curve from a set of VERTICES to a polyline with given options.

The method is internally used by `spline` and `fill_spline`, and is provided for cases when these are insufficient. See the description of options in the *spline* entry.

**reset_matrix**

Set the CTM to identity

**text_wrap TEXT, WIDTH, OPTIONS, [TAB_INDENT = 8, FROM = 0, LENGTH = -1, GLYPHS]**

Breaks the TEXT string in chunks that must fit into a WIDTH pixels wide box ( for WIDTH >= 0 ). TEXT is either a character string or a `Prima::Drawable::Glyphs` object returned from `text_shape`, or a `Prima::Drawable::Glyphs->glyphs` string of glyphs. In the latter case some wrapping options are not applicable. It is possible to send both text as TEXT and its shaped representation as GLYPHS.

The breaking algorithm and its result are managed by the OPTIONS integer value that is a combination of the following `tw::` constants:

**tw::CalcMnemonic**

Use 'hotkey' semantics, when a character preceded by the tilde character (`~`) has a special meaning, f ex it gets underlined when used in menus. If this bit is set, the first tilde character used as an escape is not calculated, and never appears in the result apart from the escaped character.

Not applicable in glyph wrapping.

**tw::CollapseTilde**

In addition to tw::CalcMnemonic, removes the tilde character from the resulting chunks.

Not applicable in glyph wrapping.

**tw::CalcTabs**

If set, treats tab ('\t') characters as TAB_INDENT times space characters.

Not applicable in glyph wrapping.

**tw::ExpandTabs**

If set, expands tab ('\t') characters as TAB_INDENT times space characters.

Not applicable in glyph wrapping.

**tw::BreakSingle**

Defines the method behavior when the text cannot fit in WIDTH. Does not affect anything else otherwise.

If set, the method returns an empty array. If unset, returns a text broken by the minimum number of characters per chunk. In the latter case the width of the resulting text blocks **will** exceed the WIDTH.

**tw::NewLineBreak**

Forces the creation of a new chunk after the newline character ('\n'). If the UTF8 text is passed, the unicode line break characters `0x2028` and `0x2029` produce the same effect as the newline character.

Not applicable in glyph wrapping.

**tw::SpaceBreak**

Forces the creation of a new chunk after the space character (' ') or the tab character.

Not applicable in glyph wrapping.

**tw::ReturnChunks**

Defines the result of the text_wrap() method.

If set, the array consists of integer pairs, where each is a text offset within TEXT and its length.

If unset, the resulting array consists of text chunks.

**tw::ReturnGlyphs**

If GLYPHS is set (only together with TEXT), this option becomes available to get the resulting chunks as sub-sets of GLYPHS.

**tw::ReturnLines**

Equals to 0, is a mnemonic to an unset tw::ReturnChunks.

When wrapping glyphs, has the same effect as the `tw::ReturnGlyphs` flag.

**tw::WordBreak**

If unset, the TEXT breaks as soon as the chunk width exceeds WIDTH. If set, tries to keep words in TEXT so they do not appear in two chunks, e.g. breaks TEXT by words, not by characters.

If Prima is compiled with the *libthai* library and Thai text is detected, Prima uses the library to detect the word boundaries because the Thai language does not use spaces between words. This behavior can be disabled by running Prima with `--no-libthai`.

Not applicable in glyph wrapping.

**tw::ReturnFirstLineLength**

If set, `text_wrap` proceeds until the first line is wrapped, either by width or ( if specified ) by break characters. Returns the length of the resulting line. Used for efficiency as the inverted `get_text_width` function.

If OPTIONS has tw::CalcMnemonic or tw::CollapseTilde bits set, then the last scalar of the array is a special hash reference. The hash contains extra information regarding the 'hotkey' position of the underline - it is assumed that the tilde character '~' prefixes the underlined character. The hash contains the following keys:

**tildeLine**

Chunk index that contains the escaped character. Set to undef if no tilde escape was found; the rest of the information in the hash is not relevant in this case.

**tildeStart**

The horizontal offset of the beginning of the line that underlines the escaped character.

**tildeEnd**

The horizontal offset of the end of the line that underlines the escaped character.

**tildeChar**

The escaped character.

Context used: font

### text_wrap_shape TEXT, WIDTH = -1, %OPTIONS

Runs `text_shape` over results from a `text_wrap` call, with `TEXT`, `WIDTH`, `$OPTIONS{options}`, and `$OPTIONS{tabs}`. Other `%OPTIONS` are used in the `text_shape` call. Where `text_wrap` returns text substrings or positions, return glyphs objects or their positions instead.

When called with `tw::CalcMnemonic` options, recalculates the tilde position so it adapts to the glyph positions returned.

If `$OPTIONS{kashida}` is set, performs kashida justification on the last wrapped line, using optional `$OPTIONS{min_kashida}` value ( see the **arabic_justify** entry in the *Prima::Drawable::Glyphs* section ).

If `$OPTIONS{letter}` or `$OPTIONS{word}` is set, performs the interspace justification on all but the last wrapped line.

## 3.5 Prima::Region

Generic shapes for clipping and hit testing

### Synopsis

```
$empty = Prima::Region->new;

$rect  = Prima::Region->new( rect => [ 10, 10, 20, 20 ]);
$rect  = Prima::Region->new( box  => [ 10, 10, 10, 10 ]); # same

$poly  = Prima::Region->new( polygon => [ 0, 0, 100, 0, 100, 100 ]);

$bits  = Prima::Region->new( image => $image );

$drawable-> region( $rect );

my $rgn = $drawable->region;
$rgn->image->save('region.png') if $rgn;
```

### Description

The Prima::Region class is a descendant of the Prima::Component class. A Prima::Region object is a representation of a generic shape that can be applied to a drawable and checked whether points are within its boundaries.

### API

**new %OPTIONS**

Creates a new region object. If called without any options then the resulting region will be empty The following options can be used:

**rect => [ X1, Y1, X2, Y2 ]**

Creates a rectangular region with inclusive-inclusive coordinates.

**box => [ X, Y, WIDTH, HEIGHT ]**

Same as `rect` but using the *box* semantics.

**polygon => \@POINTS, fillMode = 0**

Creates a polygon shape with vertices given in `@POINTS`, and using the optional fillMode ( see the **fillMode** entry in the *Drawable* section ).

**image => IMAGE**

Creates a region from a 1-bit image. If the image contains no pixels that are set to 1, the resulting region is created as an empty region.

**bitmap with_offset => 0, type => dbt::Bitmap**

Paints the region on a newly created bitmap and returns it. By default, the region offset is not included.

**box**

Returns the (X,Y,WIDTH,HEIGHT) bounding box that encloses the smallest possible rectangle, or (0,0,0,0) if the region is empty.

**combine REGION, OPERATION = rgnop::Copy**

Applies one of the following set operations to the region:

**rgnop::Copy**

Makes a copy of the REGION

**rgnop::Intersect**

The resulting region is an intersection of the two regions.

**rgnop::Union**

The resulting region is a union of the two regions.

**rgnop::Xor**

Performs XOR operation on the two regions.

**rgnop::Diff**

The resulting region is a difference between the two regions.

**dup**

Creates a duplicate region object

**get_boxes**

Returns a `Prima::array` object filled with 4-integer tuples, where each is a box defined as a (x,y,width,height) tuple.

**get_handle**

Returns the system handle for the region

**equals REGION**

Returns true if the regions are equal, false otherwise.

**image with_offset => 0, type => dbt::Bitmap**

Paints the region on a newly created image and returns it. By default, the region offset is not included.

**is_empty**

Returns true if the region is empty, false otherwise.

**offset DX, DY**

Shifts the region vertically and/or horizontally

**point_inside X, Y**

Returns true if the (X,Y) point is inside the region

**rect_inside X1,Y1,X2,Y2**

Checks whether a rectangle given by the inclusive-inclusive coordinates is inside, outside, or partially covered by the region. The return value can be one of these flags:

```
rgn::Inside
rgn::Outside
rgn::Partially
```

where the `rgn::Outside` constant has the value of 0.

## 3.6 Prima::Image

2-D graphic interface for images

### Synopsis

```
use Prima qw(Application);

# create a new image from scratch
my $i = Prima::Image-> new(
   width => 32,
   height => 32,
   type   => im::BW, # same as im::bpp1 | im::GrayScale
);

# draw something
$i-> begin_paint;
$i-> color( cl::White);
$i-> ellipse( 5, 5, 10, 10);
$i-> end_paint;

# resize
$i-> size( 64, 64);

# file operations
$i-> load('a.gif') or die "Error loading:$@\n";
$i-> save('a.gif') or die "Error saving:$@\n";

# draw on screen
$::application-> begin_paint;

# the color image is drawn as specified by its palette
$::application-> put_image( 100, 100, $i);

# a bitmap is drawn as specified by the colors of the destination device
$::application-> set( color => cl::Red, backColor => cl::Green);
$::application-> put_image( 200, 100, $i-> bitmap);
```

### Description

`Prima::Image`, `Prima::Icon`, and `Prima::DeviceBitmap` are the classes for bitmap handling, file, and graphic input and output. `Prima::Image` and `Prima::DeviceBitmap` are descendants of `Prima::Drawable` and represent bitmaps, stored in memory. `Prima::Icon` is a descendant of `Prima::Image` and also contains a 1-bit transparency mask or an 8-bit alpha channel.

### Usage

Pixel storage is usually a contiguous memory area, where scanlines of pixels are stored row-wise. The Prima toolkit is no exception, however, it does not assume that the underlying GUI system uses the same memory format. The implicit conversion routines are called when `Prima::Image` is about to be drawn onto the screen, for example. The conversions are not always efficient, therefore the `Prima::DeviceBitmap` class is introduced to represent a bitmap, stored in the system memory in the system pixel format. These two basic classes serve different needs but can be easily converted to each other, with the `image` and `bitmap` methods. `Prima::Image` is a more general bitmap representation, capable of file and graphic input and output, plus it is supplied with a

set of conversion and scaling functions. The `Prima::DeviceBitmap` class has almost none of the additional functionality and is used for efficient graphic input and output.

Note: If you're looking for information on how to display an image, you may want to read first the *Prima::ImageViewer* section manual page, or use `put_image` / `stretch_image` ( the *Prima::Drawable* section ) inside your widget's onPaint callback.

### Graphic input and output

As descendants of `Prima::Drawable`, all `Prima::Image`, `Prima::Icon`, and `Prima::DeviceBitmap` objects are also subject to three-state painting mode - normal ( disabled ), painting ( enabled ), and informational. `Prima::DeviceBitmap`, however, exists only in the enabled state, and cannot be switched to the other two.

When an image enters the enabled state, it can be used as a drawing canvas, so that all `Prima::Drawable` operations can be performed on it. When the image is back in the disabled state, the canvas pixels are copied back to the object- associated memory, in the pixel format supported by the toolkit. When the object enters the enabled state again, the pixels are copied to the system bitmap memory, in the pixel format supported by the system. In case the system pixel representation is less precise than Prima's, f ex when drawing on a 24-RGB image when the system has only 8-bit paletted display, then some pixel information will be lost in the process.

Image objects can be drawn on other images and device bitmaps, as well as on the screen and `Prima::Widget` objects. These operations are performed via one of the `Prima::Drawable::put_image` group methods ( see the *Prima::Drawable* section) and can be called with the image object in any paint state. The following code illustrates the dualism of the image object, where it can serve both as the drawing target and the drawing source:

```
my $a = Prima::Image-> new( width => 100, height => 100, type => im::RGB);
$a-> begin_paint;
$a-> clear;
$a-> color( cl::Green);
$a-> fill_ellipse( 50, 50, 30, 30);
$a-> end_paint;
$a-> rop( rop::XorPut);
$a-> put_image( 10, 10, $a);
$::application-> begin_paint;
$::application-> put_image( 0, 0, $a);
$::application-> end_paint;
```

A special case is a 1-bit ( monochrome ) DeviceBitmap. When it is drawn on a drawable with a bit depth greater than 1, the drawable's `color` and `backColor` properties are used to reflect the source's 1 and 0 bits, respectively.

### File input and output

Depending on the toolkit configuration, images can be read and written in different file formats. This functionality is accessible via the `load()` and `save()` methods. the *Prima::image-load* section describes the loading and saving parameters that can be passed to these methods, so they can handle different aspects of file format-specific options, such as multi-frame operations, auto conversion when a format does not support a particular pixel type, etc. In this document, the `load()` and `save()` methods are illustrated only in their basic, single-frame functionality. When called with no extra parameters, these methods fail only if a disk I/O error occurs or an unsupported image format is used.

### Pixel formats

`Prima::Image` supports several pixel formats, managed by the `::type` property. The property is an integer value, a combination of the `im::XXX` constants. The toolkit defines standard pixel

formats for the color formats (16-color, 256-color, 16M-color), and the gray-scale formats, mapped to C data types - unsigned char, unsigned short, unsigned long, float, and double. The gray-scale formats can be based on real-number types and complex-number types; the latter are represented by two real values per pixel, as the real and imaginary values.

A `Prima::Image` object can also be initialized from other pixel formats, that it does not support internally, but can convert data from. Currently, these are represented by a set of permutations of the 32-bit RGBA format, and 24-bit BGR format. These formats can only be used in conjunction with the `::data` property.

The conversions can be performed between any of the supported formats ( to do so, the `::type` property is to be set-called ). An image of any of these formats can be drawn on the screen, but if the system can not accept the pixel format ( as it is with the non-integer or complex formats ), the bitmap data are implicitly converted. The conversion does not change the data if the image is about to be drawn; the conversion is performed only when the image is about to be served as a drawing surface. If, for any reason, it is desired that the pixel format is not to be changed, the `::preserveType` property must be set to 1. It does not prevent the conversion, but it detects if the image was implicitly converted inside the `end_paint()` call, and reverts it to the previous pixel format.

There are situations when the pixel format must be changed together with down-sampling the image. One of four down-sampling methods can be selected - without halftoning, 8x8 ordered halftoning, error diffusion, and error diffusion combined with the optimized palette. These can be set to the `::conversion` property using one of the `ict::XXX` constants. When the conversion doesn't incur information loss, the `::conversion` property is not used.

Another special case of image downsampling is the conversion with a palette. The following code,

```
$image-> type( im::bpp4);
$image-> palette( $palette);
```

and

```
$image-> palette( $palette);
$image-> type( im::bpp4);
```

produce different results, but none of these takes into account eventual palette remapping because the `::palette` property does not change bitmap pixel data, but overwrites the palette information only. The correct syntax here is

```
$image-> set(
   palette => $palette,
   type    => im::bpp4,
);
```

This syntax is most powerful when conversion is set to those algorithms that can take into account the existing image pixels to produce an optimized palette. These are `ict::Optimized` ( default ) and `ict::Posterization`. This syntax not only allows remapping or downsampling pixels to a predefined color set but also can be used to limit the palette size to a particular number, without knowing the actual values of the final color palette. For example, for a 24-bit image,

```
$image-> set( type => im::bpp8, palette => 32);
```

call would calculate colors in the image, compress them to an optimized palette of 32 cells, and finally convert the image to the 8-bit format using that palette.

Instead of the `palette` property, the `colormap` property can also be used.

### Data access

The individual pixel values can be accessed in the same way as in the `Prima::Drawable` class, via the `::pixel` property. However, `Prima::Image` introduces several helper functions on its own.

The `::data` property is used to set or retrieve the scalar representation of pixel data. The data are expected to be lined up to a 'line size' margin ( 4-byte boundary ), which is calculated as

```
$lineSize = int(( $image->width * ( $image-> type & im::BPP) + 31) / 32) * 4;
```

or returned from the read-only property `::lineSize`.

That value is the actual size of a single row of pixels as stored internally in the object memory, however, the input to the `::data` property should not necessarily be aligned to this value, it can be accompanied by a write-only flag 'lineSize' if the pixels are aligned differently:

```
$image-> set( width => 1, height=> 2);
$image-> type( im::RGB);
$image-> set(
   data => 'RGB----RGB----',
   lineSize => 7,
);
print $image-> data, "\n";

output: RGB-RGB-
```

Internally, Prima contains images in memory so that the first scanline is farthest away from the memory start; this is consistent with general Y-axis orientation in the Prima drawable paradigm but might be inconvenient when importing data that are organized otherwise. Another write-only boolean flag `reverse` can be set to 1 so data then are treated as if the first scanline of the image is closest to the start of data:

```
$image-> set( width => 1, height=> 2, type => im::RGB);
$image-> set(
   data => 'RGB-123-',
   reverse => 1,
);
print $image-> data, "\n";

output: RGB-123-
```

Although it is possible to perform all kinds of calculations and modifications with the pixels returned by the `::data` property, it is not advisable unless the speed does not matter. Standalone PDL package with the help of the *PDL::PrimaImage* entry package, and Prima-derived IPA package provide routines for data and image analysis provide tools for efficient pixel manipulations. Also, the *Prima::Image::Magick* section connects the *ImageMagick* entry with Prima. `Prima::Image` itself provides only the simplest statistical information, namely: the lowest and highest pixel values, the arithmetic sum of pixel values, the sum of pixel squares, the mean value, variance, and standard deviation.

### Standalone usage

All of the drawing functionality can be used standalone, with all other parts of the toolkit being uninitialized. Example:

```
my $i = Prima::Image->new( size => [5,5]);
$i->color(cl::Red);
$i->bar(0,0,$i->size);
$i->save('1.bmp');
```

This feature is useful in non-interactive programs, running in environments with no GUI access, for example, a CGI script with no access to an X11 display. Normally, Prima fails to start in such situations but can be told not to initialize the GUI part by explicitly specifying system-dependent options. See the *Prima::noX11* section for more.

Generally, the standalone methods support all the OS-specific functions (i.e. color, region, etc). Also, the graphic primitives and `put_image` methods support drawing using the Porter-Duff and Photoshop operators that can be specified in the `::rop` property by using values from the extended set of the `rop::XXX` constants, i e rop::SrcOver and above.

All text API is also supported (on unix if Prima is compiled with freetype and fontconfig) and can be used transparently for the caller. The list of available fonts, and their renderings, may differ from the fonts available in the system. For example, where the system may choose to render glyphs with pixel layout optimized for LCD screens, the font query subsystem may not.

See individual methods and properties in the *API* entry that support standalone usage, and how they differ from system-dependent implementation.

### Prima::Icon

The `Prima::Icon` class inherits all properties of `Prima::Image` and features the 1-bit transparency mask or the 8-bit alpha channel. The mask can also be loaded and saved into image files if the format supports transparency.

Similar to the `Prima::Image::data` property, the `Prima::Icon::mask` property provides access to the binary mask data. The mask can be updated automatically after an icon object is subjected to painting, resizing, or other destructive changes. The auxiliary properties `::autoMasking` and `::maskColor`/`::maskIndex` regulate the mask update procedure. For example, if an icon was loaded with the color ( vs. mask ) transparency information, the binary mask will be generated anyway, but it will be also recorded that a particular color is transparent, so eventual conversions can rely on the color value instead.

Drawing using an icon ignores the `::rop` value except when its mask is an 8-bit alpha channel, in which case only the Photoshop and Porter-Duff operations are supported. When drawing happens on the system canvas (i e a widget, bitmap, or an image in the enabled state), the only operations supported are `rop::Blend` and `rop::SrcCopy`.

### Layering

The term *layered window* is borrowed from the Windows world, and means a window with transparency. In Prima, the property the *layered* entry is used to request this functionality. The result of the call `$::application->get_system_value(sv::LayeredWidgets)` can show if this functionality is available; if not, the `::layered` property is ignored. By default, widget layering is turned off.

A layered drawable uses an extra alpha channel to for the transparency pixels. Drawing on widgets looks different as well - for example, drawing with black color will make the black pixels fully transparent, while other colors will blend with the underlying background. Prima provides graphics primitives to draw using alpha effects, and some image functions to address layered surfaces.

The `put_image` and `stretch_image` functions can operate on layered surfaces both as source and destination drawables. To address the alpha channel on a drawable use either a `Prima::Icon` with `maskType(im::bpp8)`, or a layered `DeviceBitmap`.

The corresponding `Prima::DeviceBitmap` type is `dbt::Layered`, and is fully compatible with layered widgets in the same fashion as `DeviceBitmap` with type `dbt::Pixmap` is fully compatible with normal widgets. One of the ways to put a constant alpha value over a rectangle is, for example, like this:

```
my $a = Prima::Icon->new(
    width    => 1,
```

```
    height   => 1,
    type     => im::RGB,
    maskType => im::bpp8,
    data     => "\0\0\0",
    mask     => chr( $constant_alpha ),
);
$drawable-> stretch_image( 0, 0, 100, 100, $a, rop::SrcOver );
```

If displaying a picture with a pre-existing alpha channel, you'll need to call the *premultiply_alpha* entry because the picture renderer assumes that pixel values are premultiplied.

Even though addressing the alpha values of pixels of the layered surfaces is not straightforward, the conversion between images and device bitmaps fully supports alpha pixels. This means that:

* When drawing on an icon with an 8-bit alpha channel (*argb* icon), any changes to the alpha values of pixels will be transferred back to the mask property after `end_paint`

* Calls to the `icon` method on a DeviceBitmap with type `dbt::Layered` produce identical argb icons. Calls to the `bitmap` method on argb icons produce identical layered device bitmaps.

* Putting argb icons and layered device bitmap on other drawables yields identical results.

Putting images on argb source surfaces can be only used with two raster operators, `rop::Blend` (default) and `rop::SrcCopy`. The former produces the blending effect, while the latter copies alpha bits over to the destination surface. Also, a special `rop::AlphaCopy` can be used to treat 8-bit grayscale source images as alpha maps, to replace the alpha pixels only.

Prima's internal implementation of the `put_image` and the `stretch_image` functions extends the allowed set of raster operators when operating on images outside the begin_paint/end_paint brackets. These operators include 12 Porter-Duff operators, a set of Photoshop operators, and special flags to specify constant alpha values to override the existing alpha channel, if any. See more in the **Raster operations** entry in the *Prima::Drawable* section.

Caveats: In Windows, mouse events will not be delivered to the layered widget if the pixel under the mouse pointer is fully transparent.

See also: *examples/layered.pl*.

## API

### Prima::Image properties

#### codec $NAME

In the get-call, returns the codec name that loaded the image, given the extras were loaded using the `loadExtras => 1` options. Returns undef if cannot detect the codec.

In the set-call, assigns the codec to the image extras, so that a `save` call can use that to save in the desired format.

#### colormap @PALETTE

The color palette is used for representing 1, 4, and 8-bit bitmaps when the image object is to be visualized. @PALETTE contains combined RGB colors as 24-bit integers, 8 bits per component. For example, the colormap values for a typical black-and-white monochrome image can be `0,0xffffff`.

See also `palette`.

#### conversion TYPE

Selects the type of dithering algorithm to be used for pixel down-sampling. TYPE is one of the `ict::XXX` constants:

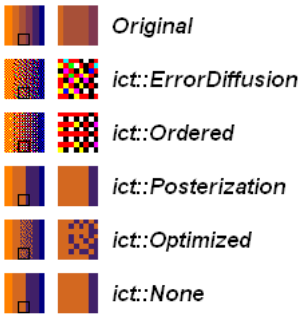
```
ict::None            - no dithering, with a static palette or palette optimized by the so
ict::Posterization   - no dithering, with palette optimized by the source pixels
```

```
ict::Ordered         - fast 8x8 ordered halftone dithering with a static palette
ict::ErrorDiffusion  - error diffusion dithering with a static palette
ict::Optimized       - error diffusion dithering with an optimized palette
```

As an example, if a 4x4 color image with every pixel set to RGB(32,32,32) is downsampled to a 1-bit image, the following results may occur:

```
ict::None, ict::Posterization:
  [ 0 0 0 0 ]
  [ 0 0 0 0 ]
  [ 0 0 0 0 ]
  [ 0 0 0 0 ]

ict::Ordered:
  [ 0 0 0 0 ]
  [ 0 0 1 0 ]
  [ 0 0 0 0 ]
  [ 1 0 0 0 ]

ict::ErrorDiffusion, ict::Ordered:
  [ 0 0 1 0 ]
  [ 0 0 0 1 ]
  [ 0 0 0 0 ]
  [ 0 0 0 0 ]
```

Values of these constants are made from the **ictp::** entry in the *Prima::Const* section and the **ictd::** entry in the *Prima::Const* section constants.




### data SCALAR

Provides access to the pixel data. On the get-call returns all the bitmap pixels, aligned to a 4-byte boundary. On the set-call, stores the provided data with the same 4-byte alignment. The alignment can be altered by submitting the write-only `lineSize` flag to the set-call. The ordering of scan lines can be altered by setting the write-only `reverse` flag ( see the *Data access* entry ).

### exif HASH

A shortcut to the Exif parser and compiler, `Prima::Image::Exif` , to operate on the image extras.

### height INTEGER

Manages the vertical dimension of the image data. On the set-call, the image content is changed to adapt to the new height, and depending on the value of the `::vScaling` property, the pixel values are either scaled or truncated, with or without resampling.

**lineSize INTEGER**

A read-only property, returning the length of a row of pixels in bytes, as represented internally in memory. Data returned by the `::data` property are aligned to `::lineSize` bytes per row. Setting the `::data` property expects the input scalar to be aligned to this value unless the `lineSize` field is set together with `data` to indicate another alignment. See the *Data access* entry for more.

**mean**

Returns the mean value of pixels. The mean value is a `::sum` of pixel values, divided by the number of pixels.

**palette [ @PALETTE ]**

The color palette is used for representing 1, 4, and 8-bit bitmaps when the image object is to be visualized. @PALETTE contains individual color component (R,G,B) triplets as 8-bit integers. For example, the palette values for a typical black-and-white monochrome image can be `[0,0,0,255,255,255]`.

See also `colormap`.

**pixel ( X_OFFSET, Y_OFFSET ) PIXEL**

Provides per-pixel access to the image data when the image object is in the disabled paint state.

Pixel values for grayscale 1-, 4-, and 8-bit images are treated uniformly, their values range from 0 to 255. For example, values for grayscale 1-bit images are 0 and 255, not 0 and 1.

In the paint state behaves in the same way as `Prima::Drawable::pixel`.

**preserveType BOOLEAN**

If 1, reverts the image type and eventual palette to their old values whenever an implicit pixel format change is needed, for example during `end_paint()`. This option can be expensive, and repetitive conversions can drastically degrade image quality; use with care.

Default: false

See also: `conversion`

**rangeHi**

Returns the maximum pixel value in the image data.

**rangeLo**

Returns the minimum pixel value in the image data.

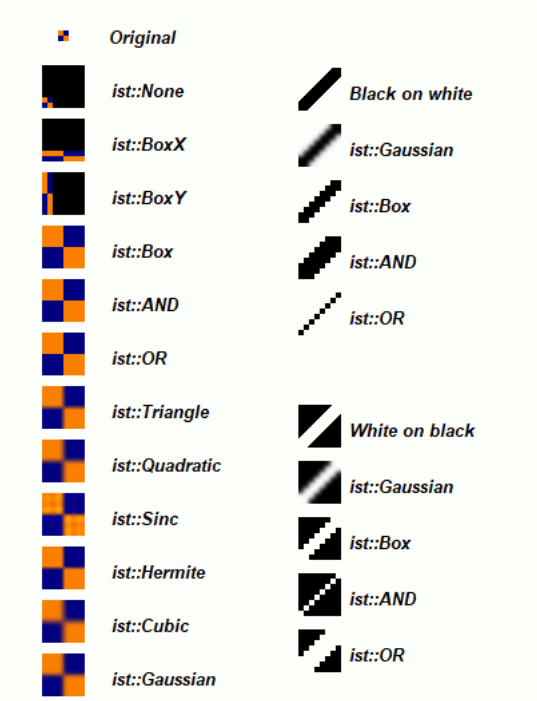
**scaling INT**

Declares the scaling strategy when the image is resized. Strategies `ist::None` through `ist::Box` are very fast scalers, while the others are slower.

Can be one of `ist:::XXX` constants:

```
ist::None      - the image will be either stripped (when downsizing)
                 or padded (when upsizing) with zeros
ist::Box       - the image will be scaled using a simple box transform
ist::BoxX      - columns will behave the same as in ist::None,
                 rows will behave the same as in ist::Box
ist::BoxY      - rows will behave the same as in ist::None,
                 columns will behave the same as in ist::Box
ist::AND       - when rows or columns are to be shrunk, leftover pixels
```

```
                  will be AND-end together (for black-on-white images)
                  ( does not work for floating point pixels )
ist::OR         - when rows or columns are to be shrunk, leftover pixels
                  will be OR-end together (for white-on-black images)
                  ( does not work for floating point pixels )
ist::Triangle   - bilinear interpolation
ist::Quadratic  - 2nd order (quadratic) B-Spline approximation of the Gaussian
ist::Sinc       - sine function
ist::Hermite    - B-Spline interpolation
ist::Cubic      - 3rd order (cubic) B-Spline approximation of the Gaussian
ist::Gaussian   - Gaussian transform with gamma=0.5
```




Note: Resampling scaling algorithms (those greater than `ist::Box`), when applied to Icons with a 1-bit icon mask will silently upgrade the mask to 8 bits and apply the same scaling algorithm to it. This will have a great smoothing effect on mask edges if the system supports ARGB layering (see the *Layering* entry ).

**size WIDTH, HEIGHT**

Manages the dimensions of the image data. On the set-call, the image content is changed to adapt to the new height, and depending on the value of the `::vScaling` property, the pixel values are either scaled or truncated, with or without resampling.

**stats ( INDEX ) VALUE**

Returns one of the calculated statistics addressed by INDEX, which can be one of the following `is::XXX` constants:

```
is::RangeLo   - minimum pixel value
is::RangeHi   - maximum pixel value
is::Mean      - mean value
is::Variance  - variance
```

```
is::StdDev   - standard deviation
is::Sum      - the sum of pixel values
is::Sum2     - the sum of squares of pixel values
```

The values are re-calculated on request and cached. On the set-call VALUE is stored in the cache, and is returned on the next get-call. The cached values are discarded every time the image data changes.

These values are also accessible via a set of alias properties: `::rangeLo`, `::rangeHi`, `::mean`, `::variance`, `::stdDev`, `::sum`, and `::sum2`.

**stdDev**

Returns the standard deviation of the image data. The standard deviation is the square root of `::variance`.

**sum**

Returns the sum of pixel values of the image data

**sum2**

Returns the sum of squares of pixel values of the image data

**type TYPE**

Manages the image pixel format type. TYPE is a combination of the `im::XXX` constants. The constants are collected in groups:

Bit-depth constants provide the size of pixels in bits. Their actual value is the same as the number of bits, so the value of the `im::bpp1` constant is 1, `im::bpp4` is 4, etc. The supported constants represent the bit depths from 1 to 128:

```
im::bpp1
im::bpp4
im::bpp8
im::bpp16
im::bpp24
im::bpp32
im::bpp64
im::bpp128
```

The following values reflect the pixel format category:

```
im::Color
im::GrayScale
im::RealNumber
im::ComplexNumber
im::TrigComplexNumber
im::SignedInt
```

The value of the `im::Color` constant is 0, whereas other category constants are represented by unique bit values, so a combination of `im::RealNumber` and `im::ComplexNumber` becomes possible (although not all of the combinations are supported).

There are also several mnemonic constants defined:

```
im::Mono          - im::bpp1
im::BW            - im::bpp1 | im::GrayScale
im::16            - im::bpp4
im::Nibble        - im::bpp4
im::256           - im::bpp8
im::RGB           - im::bpp24
im::Triple        - im::bpp24
im::Byte          - gray 8-bit unsigned integer
im::Short         - gray 16-bit unsigned integer
im::Long          - gray 32-bit unsigned integer
im::Float         - float
im::Double        - double
im::Complex       - dual float
im::DComplex      - dual double
im::TrigComplex   - dual float
im::TrigDComplex  - dual double
```

The bit depths of the float- and double-derived pixel formats depend on the platform.

These values can be isolated using the mask values:

```
im::BPP      - bit depth constants
im::Category - category constants
im::FMT      - extra format constants
```

The extra formats are the pixel formats, not supported by the `::type` property, but recognized in the combined set-call, for example like this:

```
$image-> set(
   type => im::fmtBGRI,
   data => 'BGR-BGR-',
);
```

The data, supplied with the extra image format specification will be converted to the closest supported format. Currently, the following extra pixel formats are recognized:

```
im::fmtBGR
im::fmtRGBI
im::fmtIRGB
im::fmtBGRI
im::fmtIBGR
```

**variance**

Returns the variance of pixel values of the image data. The variance is `::sum2`, divided by the number of pixels minus the square of `::sum` of pixel values.

**width INTEGER**

Manages the horizontal dimension of the image data. On the set-call, the image content is changed to adapt to the new height, and depending on the value of the `::vScaling` property, the pixel values are either scaled or truncated, with or without resampling.

### Prima::Icon properties

#### autoMasking TYPE

Selects if the mask information should be updated automatically after `::data` is changed. Every `::data` change is mirrored in `::mask`, using TYPE, one of the `am::XXX` constants:

```
am::None          - no mask update performed
am::MaskColor     - mask update based on ::maskColor property
am::MaskIndex     - mask update based on ::maskIndex property
am::Auto          - mask update based on corner pixel values
```

The `::maskColor` color value is used as a transparent color if TYPE is `am::MaskColor`. The transparency mask generation algorithm turned on by `am::Auto` checks corner pixel values, assuming that the majority of the corner pixels represent a transparent color. Once such color is found, the mask is generated as in the `am::MaskColor` case.

`::maskIndex` is the same as `::maskColor`, except that it points to a specific color index in the palette.

When image `::data` is stretched, `::mask` is stretched accordingly, disregarding the `::autoMasking` value.

#### mask SCALAR

Provides access to the transparency pixels. On the get-call, returns all mask pixels, aligned to a 4-byte boundary. On the set-call, stores the provided transparency data with the same alignment. If the SCALAR is an image object, copies its pixels as a new mask. In that case, copies the pixels as is if the format matches (i e 1-bit icon mask receives 1-bit pixels from the image). or the image data is converted to 8 bits and the mask is converted to the 8-bit format as well.

#### maskColor COLOR

When the `::autoMasking` property is set to `am::MaskColor`, COLOR is used as the transparency value.

#### maskIndex INDEX

When the `::autoMasking` property is set to `am::MaskIndex`, the INDEXth color in the current palette is used as the transparency value.

#### maskLineSize INTEGER

A read-only property, returning the length of the mask row in bytes, as represented internally in memory. Data returned by the `::mask` property is aligned with `::maskLineSize` bytes per row.

#### maskPixel ( X_OFFSET, Y_OFFSET ) PIXEL

Provides per-pixel access to the icon mask.

In the disabled mode, gets and sets the value directly from the mask memory. In the paint mode, and if (and only if) the mask depth is 8 bits, queries the alpha pixel value from the system paint surface. Pixel values for all mask depths are treated uniformly, their values range from 0 to 255. For example, values for 1-bit mask pixels are 0 and 255, not 0 and 1.

#### maskType INTEGER

Is either `im::bpp1` (1) or `im::bpp8` (8). The latter can be used as a layered (argb) source surface to draw with blending effects.

Note: if a mask with depth 8 is downgraded to depth 1, the image pixels that correspond to alpha values lesser than 255 will be reset to 0.

### Prima::DeviceBitmap properties

**maskPixel ( X_OFFSET, Y_OFFSET ) PIXEL**

Provides per-pixel access to the alpha component of the layered device bitmap. If the bitmap is not layered, the property does not do anything.

**type INTEGER**

A read-only property that can only be set during creation, reflects whether the system bitmap is a black-and-white 1-bit (`dbt::Bitmap`), a colored drawable that is compatible with widgets (`dbt::Pixmap`), or is a colored drawable with an alpha channel that is compatible with layered widgets (`dbt::Layered`).

The bit depth of the bitmap pixel type can be read via the `get_bpp()` method; monochrome bitmaps always have a bit depth of 1, and layered bitmaps have a bit depth of 32.

### Prima::Image methods

The following properties are same as in the `Prima::Drawable` clear class, but can be called also outside of the paint state: `bar`, `bar_alpha`, `bars`, `chord`, `clear`, `ellipse`, `fill_chord`, `fill_ellipse`, `fill_sector`, `fill_spline`, `flood_fill`, `line`, `lines`, `pixel`, `polyline`, `put_image`, `put_image_indirect`, `rectangle`, `sector`, `spline`, `stretch_image`.

These drawing primitives are executed using the core Prima functionality, without involving the system backend.

**bitmap**

Returns a newly created `Prima::DeviceBitmap` object with the same image dimensions and pixel content.

**clone %properties**

Creates a copy of the image and applies `%properties`. An easy way to create a down-sampled copy, for example.

**codecs**

Returns an array of hashes, each describing the supported image format.

See the *Prima::image-load* section for details.

This method can be called without object instance:

```
perl -MData::Dumper=Dumper -MPrima::noX11 -MPrima -le 'print Dumper(Prima::Image->codecs)
```

**dup**

Returns a copy of the object, a newly created `Prima::Image`, with all properties copied. Does not preserve the graphical properties though (color etc).

**extract X_OFFSET, Y_OFFSET, WIDTH, HEIGHT**

Returns a newly created image object with dimensions equal to or less than WIDTH and HEIGHT, initialized with pixel data from X_OFFSET and Y_OFFSET in the bitmap. The dimensions could be less than requested if they extend past the original image dimensions.

Same as the `Drawable::` functions but can be used also outside of the paint state.

**get_bpp**

Returns the bit depth of the pixel format. Same as `::type & im::BPP`.

**get_handle**

Returns the system handle of the image object.

**has_codec $MATCH**

Returns true if Prima supports the codec. Can be called on a package.

**load (FILENAME or FILEGLOB) [ %PARAMETERS ]**

Loads an image from file FILENAME or stream FILEGLOB into an object, and returns the success flag. The method features different semantics, depending on the PARAMETERS hash. The `load()` method can be called either in the context of the existing object, then a boolean success flag is returned. Or in the class context, then a newly created object ( or `undef` ) is returned. If an error occurs, the `$@` variable contains the error string. These two invocation semantics are equivalent:

```
my $x = Prima::Image-> new();
die "$@" unless $x-> load( ... );
```

and

```
my $x = Prima::Image-> load( ... );
die "$@" unless $x;
```

See the *Prima::image-load* section for details and the *Prima::Image::Loader* section for more functionality.

Note: when loading from streams on win32, mind the `binmode`.

**load_stream BASE64_STRING, %OPTIONS**

Decodes BASE64_STRING and tries to load an image from it. Returns image reference(s) on success, or `undef` on failure; also `$@` is set in this case.

**map COLOR**

Performs iterative mapping of bitmap pixels, setting every pixel to the `::color` property with respect to the `::rop` type if a pixel equals to COLOR, and to the `::backColor` property with respect to the `::rop2` type otherwise.

The `rop::NoOper` type can be used for color masking.

Examples:

```
width => 4, height => 1, data => [ 1, 2, 3, 4]
color => 10, backColor => 20, rop => rop::CopyPut

rop2 => rop::CopyPut
input: map(2) output: [ 20, 10, 20, 20 ]

rop2 => rop::NoOper
input: map(2) output: [ 1, 10, 3, 4 ]
```

**mirror VERTICAL**

Mirrors the image either vertically or horizontally depending on the boolean flag VERTICAL

**premultiply_alpha CONSTANT_OR_IMAGE**

Applies premultiplication formula to each pixel

```
pixel = int( pixel * alpha / 255 + 0.5 )
```

where the alpha either is a constant or the corresponding pixel value in the image

**put_image, put_image_indirect, stretch_image**

Same as the `Drawable::` functions but can be used also outside of the paint state.

Extends raster functionality to access alpha channel either using constant alpha values or `Prima::Icon` as sources. See the explanation of the `rop::` constants in the **Raster operations** entry in the *Prima::Drawable* section.

**resample SRC_LOW, SRC_HIGH, DEST_LOW, DEST_HIGH**

Performs linear scaling of gray pixel values from range (SRC_LOW - SRC_HIGH) to the new range (DEST_LOW - DEST_HIGH). Can be used to visualize gray non-8-bit pixel values, by the code:

```
$image-> resample( $image-> rangeLo, $image-> rangeHi, 0, 255);
$image-> type(im::Byte);
```

**rotate DEGREES [,FILL_COLOR]**

Rotates the image. Where the angle is 90, 180, or 270 degrees, fast pixel flipping is used, otherwise fast Paeth rotation is used. Eventual resampling can be controlled by the `scaling` property ( probably not worth it for functions with a support range of more than 1 pixel).

Fills empty pixels with an optional FILL_COLOR.

The resulting images can be 1 pixel too wide due to horizontal shearing applied twice, where in worst cases 1 pixel from the original image can take 3 horizontal pixels in the resulting image.

**save (FILENAME or FILEGLOB), [ %PARAMETERS ]**

Stores image data into image file FILENAME or stream FILEGLOB, returns the success flag. The method features different semantics, depending on the PARAMETERS hash. If an error occurs, the `$@` variable contains the error string.

Note that when saving to a stream, `codecID` must be explicitly given in `%PARAMETERS`.

See the *Prima::image-load* section for details and the **Prima::Image::Saver** entry in the *Prima::Image::Loader* section for more functionality.

Note: when saving to streams on win32, mind the `binmode`.

**save_stream BASE64_STRING, %OPTIONS**

Saves the image into an internal stream. Unless `$OPTIONS{codecID}` or `$image-{extras}->{codecID}>` is set, tries to find the best codec for the job. Returns the base64-encoded content on success, or `undef` on failure; `$@` is set in the latter case.

**scanline Y**

Returns a scanline from the Y offset in the same raw format as `data`

**shear X, Y**

Applies the shearing transformation to the image. If the shearing is needed only for one axis, set the shearing factor for the other one to zero.

**convert_to_icon $MASK_DEPTH, $MASK_TEMPLATE**

Creates an icon from the image, with `$MASK_DEPTH` integer (can be either 1 or 8), and `$$MASK_TEMPLATE` scalar used for the newly created mask.

**to_colormask COLOR**

Creates a new icon with bit depth 24 filled with COLOR, where the mask bits are copied from the caller image object and upgraded to bit depth 8 if needed.

**to_rgba TYPE=undef**

Creates a new icon with type set to 24 or 8 gray bits and mask type to 8 bits. If TYPE is set, uses that type instead.

**to_region**

Creates a new the *Prima::Region* section object with the image as the data source. The image is expected to be of 1-bit depth.

**transform matrix => [a,b,c,d,x,y], [ fill => color ]**

Applies a generic 2D transform matrix to the image and fills empty pixels with an optional fill color.

The required option `matrix` should point to an array of 6 float numbers, where these represent a standard 3x2 matrix for 2D transformation, f ex a `Prima::matrix` object.

Tries first to split the matrix into a series of shear and scale transforms using the LDU decomposition; if an interim image is calculated to be too large, fails and returns `false`.

The last two matrix members (X and Y translation) only use the mantissa and ignore the integer part, so setting these f ex to 10.5 will not produce an image 11 pixels larger, but only 1. The translation is thus effectively sub-pixel.

The rotation matrices can be applied too, however, when angles are close to 90 or 270 degrees, either interim images become too big, or defects introduced by the shearing become too visible. Therefore the method specifically detects rotation cases and uses the Paeth rotation algorithm instead, which yields better results. Also, if the angle is detected to be 90, 180, or 270 degrees, fast pixel flipping is used.

Eventual resampling can be controlled by the `scaling` property.

**ui_scale %OPTIONS**

Resizes the image with smooth scaling. Understands `zoom` and `scaling` options. The `zoom` default value is the one in `$::application->uiScaling`, and the `scaling` default value is `ist::Quadratic` .

See also: the **uiScaling** entry in the *Application* section

### Prima::Image events

`Prima::Image`-specific events occur only from inside the `load` call, to report the loading progress. Not all codecs (currently JPEG,PNG,TIFF only) can report the progress to the caller. See the **Loading with progress indicator** entry in the *Prima::image-load* section for details, the **watch_load_progress** entry in the *Prima::ImageViewer* section and the **load** entry in the *Prima::Dialog::ImageDialog* section for suggested use.

**HeaderReady EXTRAS**

Called whenever the image header is read, and image dimensions and pixel type are changed accordingly to accommodate the image data.

`EXTRAS` is the hash to be stored later in the {`extras`} field on the object.

**DataReady X, Y, WIDTH, HEIGHT**

Called whenever image data that covers an area defined by the X,Y,WIDTH,HEIGHT rectangle is ready. Use the `load` option `eventDelay` to limit the rate of `DataReady` events.

**Prima::Icon methods**

**bar_alpha ALPHA <X1, Y1, X2, Y2>**

Same as `Drawable::bar_alpha` but can be used also outside of the paint state.

**combine DATA, MASK**

Copies information from the DATA and MASK images into the `::data` and the `::mask` properties. DATA and MASK are expected to be images of the same dimension.

**create_combined DATA, MASK, %SET**

Same as `combine`, except can be called without an object, and applies the `%SET` hash to the corresponding properties of the newly created icon.

**image %opt**

Renders the icon on a newly created `Prima::Image` object instance using the black background. If `$opt{background}` is given, this color is used instead.

**maskline Y**

Returns the mask scanline from the Y offset in the same raw format as `mask`

**maskImage**

Return an image created from the mask

**premultiply_alpha CONSTANT_OR_IMAGE = undef**

Applies the premultiplication formula to each pixel

```
pixel = pixel * alpha / 255
```

where alpha is the corresponding alpha value for each coordinate. If the value passed is `undef`, premultiplies the data pixels with the corresponding mask pixels.

Only applicable when `maskType` is <im::bpp8>.

**rotate, transform**

Applies the transformation to both color and mask pixels. Ignores fill color, fills with zeros in both planes.

**split**

Returns two new `Prima::Image` objects of the same dimension. Pixels in the first image are copied from the `::data` storage, and in the second one - from the `::mask` storage.

**translate matrix => [a,b,c,d,x,y]**

Same as the `translate` method in the `Prima::Image` class except that it also rotates the mask, and ignores the `fill` option - all new pixels are filled with zeros.

**ui_scale %OPTIONS**

Same as `ui_scale` from `Prima::Image`, but with few exceptions: It tries to use `ist::Quadratic` only when the system supports ARGB layering. Otherwise, falls back on the `ist::Box` scaling algorithm, and also limits the zoom factor to integers (2x, 3x, etc) only, because when displayed, the smooth-scaled color plane will not match the mask plane downgraded to 0/1 mask, and also because the box-scaling with non-integer zooms looks ugly.

### Prima::DeviceBitmap methods

**dup**

Returns a duplicate of the object, a newly created `Prima::DeviceBitmap`, with all information copied to it. Does not preserve graphical properties (color etc).

**icon**

Returns a newly created `Prima::Icon` object instance, with the pixel information copied from the object. If the bitmap is layered, returns icons with maskType set to `im::bpp8`.

**image**

Returns a newly created `Prima::Image` object instance, with the pixel information copied from the object.

**get_handle**

Returns the system handle for the system bitmap object.

## 3.7 Prima::image-load

Using the image subsystem

### Description

This document describes using the Prima image subsystem for loading and saving images

### Loading

#### Simple loading

In the simplest case, loading a single image would look like this:

```
my $x = Prima::Image-> load( 'filename.jpg');
die "$@" unless $x;
```

Image functions can be invoked either as package functions or as Prima::Image object methods. The code above could be also written as

```
my $x = Prima::Image-> new;
die "$@" unless $x-> load( 'filename.jpg');
```

In both cases, `$x` contains loaded image data upon success. If an error occurs, it is returned in the `$@` variable ( see *perlvar* ).

#### Loading from stream

`Prima::Image` can also load images by reading from a stream:

```
open FILE, 'a.jpeg' or die "Cannot open:$!";
binmode FILE;
my $x = Prima::Image-> load( \*FILE);
die "$@" unless $x;
```

#### Multiframe loading

Multiframe load calls can be issued in two ways:

```
my @x = Prima::Image-> load( 'filename.gif', loadAll => 1);
die "$@" unless $x[-1];

my $x = Prima::Image-> new;
my @x = $x-> load( 'filename.gif', loadAll => 1);
die "$@" unless $x[-1];
```

In the second case, the content of the first frame is stored in `$x` and `$x[0]`. To check if the error has occurred during the loading, inspect the last item of the returned array; it is undefined if the error indeed occurred. This check works also if an empty array is returned. Only this last item can be undefined, others are guaranteed to be valid objects.

Prima can load more than one image from a file, assuming the image format allows that. The `load` function recognizes such *multiframe* semantics when certain extra hash keys are used. These keys are:

**loadAll**

Requests to load all frames that can be read from the file:

```
loadAll => 1
```

**index**

If present, returns a single frame with the index given:

```
index => 8
```

**map**

Contains an anonymous array of the frame indices to load. The indices must be integers that are greater or equal to zero:

```
map => [0, 10, 15..20]
```

### Querying extra information

By default, Prima loads only image pixels and palette. For any other information that can be loaded, use the hash `'extras'` bound to the image object. To notify the image loader that this extra information is expected, the `loadExtras` boolean value is used:

```
my $x = Prima::Image-> load( $f, loadExtras => 1);
die "$@" unless $x;
for ( keys %{$x-> {extras}}) {
   print " $_ : $x->{extras}->{$_}\n";
}
```

The code above loads and prints extra information read from a file. Typical output, for example, from a *gif* codec based on the *libgif* library would look like this:

```
codecID : 1
transparentColorIndex : 1
comment : created by GIMP
frames : 18
```

`codecID` is a Prima-defined integer field, an internal index of the codec which had previously loaded the file. This value is useful for the explicit selection of the codec to be used for saving an image.

`frames` is also a Prima-defined field, with its integer value set to the number of frames in the image. It might be set to -1 signaling that the codec is incapable of quick reading of the frame count. If, however, it is necessary to get the actual frame count, the `wantFrames` boolean value should be set to 1 - then the `frames` field is guaranteed to be set to a 0 or a positive value. Such a request may take longer though, especially on large files with sequential access. A real-life example is a gif file with more than a thousand frames. The `wantFrames` flag is also useful in null load requests (see below).

### Multiprofile loading requests

The parameters that are accepted by the `load` function are divided into two groups - first, those that apply to the whole loading process, and then those that apply only to a particular frame. Some fields that were already mentioned (`wantFrames` and `loadAll`) belong to the first group because they affect the whole loading session. Some other parameters (`loadExtras`, `noImageData`, `noIncomplete`, `iconUnmask`) can be applied to each loaded frame, individually. A codec may as well define its own parameters, however, it is not possible to tell what parameter belongs to what group - this information is to be found in the codec documentation.

The parameters that apply to a frame, can be specified separately for every frame in a single call. For that purpose, the special parameter `profiles` is defined. The `profiles` value is expected to be an anonymous array of hashes, where each hash corresponds to a request number. For example:

```
$x-> load( $f, loadAll => 1, profiles => [
     {loadExtras => 0},
     {loadExtras => 1},
]);
```

The first hash there applies to the frame index 0, second - to the frame index 1. Note that in the code below

```
$x-> load( $f,
   map => [ 5, 10],
   profiles => [
     {loadExtras => 0},
     {loadExtras => 1},
]);
```

first hash applies to the frame index 5, and second - to the frame index 10.

### Null load requests

If it is desired to quickly peek into an image, reading only its pixel type and dimensions, one should set the `noImageData` boolean value to 1. Using `noImageData`, empty image objects are returned, that would have the `type` property set to the image type (as the codec would translate it to the Prima image type), and with the extras `width` and `height` set to the image dimensions. Example:

```
$x-> load( $f, noImageData => 1);
die "$@" unless $x;
print $x-> {extras}-> {width} , 'x' , $x-> {extras}-> {height}, 'x',
   $x-> type & im::BPP, "\n";
```

Some image information can be loaded even without frame loading - if the codec provides such a functionality. This is the only request that cannot be issued with the package syntax, an existing image object is required:

```
my $x = Prima::Image->new;
$x-> load( $f, map => [], loadExtras => 1);
```

Since no frames are required to load, an empty array is returned on success and an array with one undefined value on failure.

### Using Prima::Image descendants

If Prima needs to create a storage object, it uses by default either the class name of the caller object, or the package the request was issued on, or the `Prima::Image` class. This behavior can be altered using the parameter `className`, which defines the class to be used for each frame:

```
my @x = Prima::Image-> load( $f,
    map => [ 1..3],
    className => 'Prima::Icon',
    profiles => [
        {},
        { className => 'Prima::Image' },
        {}
    ],
```

In this example, `@x` will contain (Icon, Image, Icon) upon success.

When loading to an Icon object, the default toolkit action is to build the transparency mask based on the image data. When this is not desired, f.ex., there is no explicit knowledge of the image to be loaded, while the image may or may not contain transparency information, the `iconUnmask` boolean parameter can be used. When set to the `true` value, and the object is a `Prima::Icon` descendant, `Prima::Icon::autoMasking` is set to `am::None` before the file loading which effectively disables any attempt to generate the icon mask. By default, this option is turned off.

### Loading with the progress indicator

Some codecs (PNG,TIFF,JPEG) can signal their progress as they read image data. For this purpose, the `Prima::Image` class defines two events, `onHeaderReady` and `onDataReady`. If either (or both) are present on the image object issuing the `load` call, and the codec supports progressive loading, then these events will be called. The `onHeaderReady` event is called when the image header data is acquired, and an empty image with the required pixel dimensions and type is allocated. The `onDataReady` notification is called whenever a part of the image is ready and is loaded in the memory of the object; the position and dimensions of the loaded area are reported also. The format of the events is as follows:

```
onHeaderReady $OBJECT
onDataReady   $OBJECT, $X, $Y, $WIDTH, $HEIGHT
```

The `onHeaderReady` event is called only once while `onDataReady` is called as soon as new image data is available. To reduce the frequency of these calls, which otherwise would be issued after every scanline loaded, `load` has the parameter `eventDelay`, the minimum number of seconds that need to pass between two consecutive `onDataReady` calls. The default `eventDelay` is 0.1 .

The handling of the `onDataReady` event must be performed with care. First, the image must be accessed read-only, i e no transformations of any kind are allowed. Currently, there is no protection for such actions ( because the codec must also perform these itself), so a crash will most surely issue. Second, loading and saving of images is not in general reentrant, and although some codecs are reentrant, loading and saving images inside image events is not recommended.

There are two techniques to display the image progressively. Both of them require overloading the `onHeaderReady` and `onDataReady` callbacks. The simpler case is to call the `put_image` method from inside `onDataReady`:

```
$i = Prima::Image-> new(
        onDataReady => sub {
                $progress_widget-> put_image( 0, 0, $i);
        },
);
```

but that will most probably load heavily underlying OS-dependent conversion of the image data to native display bitmap data. A smarter, but more complex solution is to copy loaded (and only loaded) bits to a preexisting bitmap or image:

```
$i = Prima::Image-> new(
        onHeaderReady => sub {
                $bitmap = Prima::DeviceBitmap-> new(
                        width    => $i-> width,
                        height   => $i-> height,
                ));
        },
        onDataReady => sub {
                my ( $i, $x, $y, $w, $h) = @_;
```

```
                $bitmap-> put_image( $x, $y, $i-> extract( $x, $y, $w, $h));
        },
);
```

The latter technique is used by the `Prima::ImageViewer` widget class when it is ordered to monitor the image loading progress. See the **watch_load_progress** entry in the *Prima::ImageViewer* section for details.

### Truncated files

By default, codecs are not told whether they would fail on the premature end of the file or omit the error and return a truncated image. The `noIncomplete` boolean parameter tells that a codec must always fail if the image cannot be read in full. It is off by default. If indeed the codec detects that the file is incomplete, it sets the `truncated` field in the `extras` profile, if `loadExtras` was requested; the field is a string and contains the error message that occurred when the codec tried to load the truncated field.

### Inline files

Using the `Prima::Image::base64` module it is possible to convert images into the base64 format and embed the result directly into the source code. Assuming an appropriate codec was compiled in, the following would work:

```
my $icon = Prima::Icon->load_stream(<<~'ICON');
        R0lGODdhIAAgAIAAAAAAAP///ywAAAAAIAAgAIAAAAD///8CT4SPqcvtD6OctNqLcwogcK91nEhq
        3gim2Umm4+W2IBzX0fvl8jTr9SeZiU5E4a1XLHZ4yaal6XwFoSwMVUVzhoZSaQW6ZXjD5LL5jE6r
        DQUAOw==
        ICON
print $icon->save_stream;
```

### Reading one frame at a time

When one needs to load all frames from an image that contains too many frames, or there is a constraint on memory, Prima provides a way to load images one by one, without needing to allocate space for all frames in the file.

This section describes the lower-level API that allows for that functionality, however, an easier-to-use higher level API is documented in the *Prima::Image::Loader* section .

In order to read one frame at a time, the programmer needs to open a loading session, by adding the `session => 1` option to the `load` call; that call can only be made on an existing object, not on the package. The call would return the success flag and an eventual error in `$@`, as usual. No frames are loaded yet, though the `extras` hash on the caller image object may be filled, depending on the `loadExtras` option. The options supplied to the session opening call would apply to all subsequent frames, but these settings may be overridden later.

Having the session successfully opened, the subsequent calls to `load` with the `session => 1` option but with the first parameter set to undef will load the next frame. Each of those `load` call will recognize the options supplied and will apply them to indifidual frames. The session-based loading will recognize all of the function options, except the `map`, `profiles` and `loadAll` options. The loading session is closed automatically after either a first loading failure or after the end of file is reached.

## Saving

### Simple saving

The typical saving code is

```
$x-> save( 'filename.jpg') or die $@;
```

The function returns 1 on success and 0 on failure. Save requests also can be performed with the package syntax:

```
die "$@" unless Prima::Image-> save( 'filename.jpg',
    images => [$x]);
```

### Saving to a stream

Saving to a stream requires the explicit `codecID` integer value to be supplied. When the image is loaded with `loadExtras`, this field is always present on the image object and is the integer that selects the image file format.

```
my ($png_id) =
   map  { $_-> {codecID} }
   grep { $_-> {fileShortType} =~ /^png$/i }
   @{ Prima::Image-> codecs };
die "No png codec installed" unless $png_id;

open FILE, ">", "a.png" or die "Cannot save:$!";
binmode FILE;
$image-> save( \*FILE, codecID => $png_id)
   or die "Cannot save:$@";
```

### Multiframe saving

When saving more than one image object into a single file the method returns the number of successfully saved frames. The saved image file is erased though, if an error occurs, even after some successfully written frames.

```
 die "$@" if scalar(@images) > Prima::Image-> save( $f,
    images => \@images);
```

### Saving extras information

All information that is found in the object hash reference `extras`, is assumed to be saved in the image file too. It is a codec's own business how it reacts to invalid and/or unacceptable information - but a typical behavior is that keys that were not recognized by the codec get ignored, while invalid values raise an error.

```
    $x-> {extras}-> {comments} = 'Created by Prima';
    $x-> save( $f);
```

### Saving one frame at a time

Similar to the session-based loading, Prima provides the functionality to save a multi-frame image with one frame at a time, using the similar API calls.

This section describes the lower-level API that allows for that functionality, however, an easier-to-use higher level API is documented in the **Prima::Image::Saver** entry in the *Prima::Image::Loader* section .

In order to save one frame at a time, the programmer needs to open a saving session, by adding the `session => 1` option to the `save` call, and the `frames` options that signals how many frames are to be saved in total; that call can only be made on an existing object, not on the package. The call would return the success flag and an eventual error in `$@`, as usual. The options supplied to the

session opening call would apply to all subsequent frames, but these settings may be overridden later.

Having the session successfully opened, the subsequent calls to `save` with the `session => 1` and the image as the first option option would save the next frame. Each of those `save` call will recognize the options supplied and will apply them to indifidual frames. The session-based saving will recognize all of the function options, except the `images` option. The saving session is closed automatically after a first failure.

#### Selecting a codec

The integer field `codecID`, the same field that is defined after successful load requests, explicitly selects the codec for saving the image. If the codec is incapable of saving then an error is returned. Selecting a codec is only possible with the object-driven syntax, and this information is never extracted from the objects but is passed in the `images` array instead.

```
$x-> {extras}-> {codecID} = 1;
$x-> save( $f);
```

The actual relation between codecs and their indices is described below.

Note: if `codecID` is not specified, Prima tries to deduce the codec by the file extension.

#### Type conversion

Codecs usually are incapable of saving images in all possible pixel formats, so whenever Prima needs to save an image with an unsupported pixel type it either converts the image to an appropriate pixel format or signals an error. This behavior is managed by the parameter `autoConvert`, which is 1 by default. With the `autoConvert` set, it is guaranteed that the image will be saved, but original image information may be lost. With the `autoConvert` field set to 0, no information will be lost, but Prima may signal an error. Therefore general-purpose saving routines should be planned carefully. As an example, the `Prima::Dialog::ImageDialog::SaveImageDialog` code might be useful.

When the conversion takes place, the Image property `conversion` is used for the selection of the error distribution algorithm if down-sampling is required.

### Managing the codecs

Prima provides the `Prima::Image-> codecs` function that returns an anonymous array of hashes where every hash entry corresponds to a registered codec. The `codecID` parameter on load and save requests is the index in this array. Indexes for the codecs registered once never change, so it is safe to manipulate these numbers within a single run of the program

Codec information that is contained in these hashes contains the following fields:

**codecID**

A unique integer value for a codec, the same as the index of the codec entry in the result of the `Prima::Image->codecs` method

**name**

The full codec name, string

**vendor**

The codec vendor, string

**versionMajor and versionMinor**

Usually underlying library versions, integers

**fileExtensions**

An array of strings, with file extensions that are typical to the codec. Example: ['tif', 'tiff']

**fileType**

Description of the type of file that the codec is designed to work with. A string.

**fileShortType**

Short description of the type of file that the codec is designed to work with ( short means 3-4 characters ). A string.

**featuresSupported**

An array of strings containing string description of the features that the codec supports - usually they implement only a part of file format specification, so it is always interesting to know, which part

**module and package**

Specifies the perl module, usually inside the *Prima/Image* directory in the Prima distribution, and the package name inside the module. The package contains some specific functions for working with codec-specific parameters. The current implementation defines the only `save_dialog` function that creates a dialog that allows to change these parameters. See `Prima::Dialog::ImageDialog::SaveImageDialog` for details. Strings or `undef`.

**canLoad**

1 if the codec can load images, 0 if not

**canLoadStream**

1 if the codec can load images from streams, 0 otherwise

**canLoadMultiple**

1 if the codec can handle multiframe load requests and load frames with an index more than zero, 0 otherwise

**canSave**

1 if the codec can save images, 0 if not.

**canSaveStream**

1 if the codec can save images to streams, 0 otherwise

**canSaveMultiple**

Is set if the codec can save more than one frame

**types**

An array of integers - each is a combination of the `im::XXX` flags, the image type, that the codec can save. The first type in the list is the default type; if the type of the image to be saved is not in that list, the image will be converted to this default type.

**loadInput**

A hash, where the keys are those that are accepted by the `Prima::Image-> load` method and the values are the default values for these keys.

**loadOutput**

An array of strings, each is the name of an extra information entry in the `extras` hash.

**saveInput**

A hash, where the keys are those that are accepted by the `Prima::Image-> save` method and the values are the default values for these keys.

**mime**

An array of strings, containing the mime identifiers specific to the image format. An example: ['image/xbm', 'image/x-bitmap']

## API

This section describes the parameters accepted and the data returned by the `Prima::Image::load` method

### Common

### Loading parameters

**blending BOOLEAN = 1**

Affects how to treat the alpha channel bits, if present.

If set, mixes the alpha channel with the background color in case when loading to an image, or premultiplies color bits (either data or palette) with alpha, when loading to an icon. Note that saving the object will result in different image pixel values, but the object will be ready to be displayed immediately.

If unset, the color and eventual alpha bits, if loaded to an icon, will not be affected in any way. Note that saving the object will result in the same image, but the object will not be ready to be displayed immediately. See also: the **premultiply_alpha** entry in the *Prima::Image* section.

**className STRING**

When loading more than one image, this string is used to create instances of image containers. By default, the calling class is used (i.e. `Prima::Image` or `Prima::Icon`).

**eventDelay INT**

Specifies `onDataReady` event granularity in seconds, when the codec is capable of triggering this event.

Default: 0.1

**iconUnmask BOOL**

If set, `Prima::Icon::autoMasking` is set to `am::None` before the file loading which disables any attempt to deduce the mask pixels based on the data pixels.

Default: false. Only actual for `Prima::Icon` loading.

**index INT**

When loading from a multiframe file, selects the frame index to load.

Default: 0

**loadExtras BOOL**

If set, all available extra information will be stored in the `extras` hash on the loaded object.

Default: false

**loadAll BOOL**

When loading from a multiframe file, selects that all frames are to be loaded

Default: false

**map [INT]**

When loading from a multiframe file, selects the set of the frame indexes to load.

Default: undef

**noImageData BOOL**

If set, neither image data is loaded, nor image dimensions are changed (newly created images have a size of 1x1). Instead, the {`extras`} hash contains `width` and `height` integers.

Default: false

**noIncomplete BOOL**

Affects the action when the image is incomplete, truncated, etc. If set, signals an error. Otherwise, no error is signaled and whatever data could be recovered from the image is returned. In the latter case, the `truncated` field contains the error string.

Default: false

**profiles [HASH]**

An array of hashes passed down to each frame in multiframe loading sessions. Each frame loading request will be provided with an individual hash, a result of the hash join of all profiles passed to `Image::load` and the nth hash in the array.

**wantFrames BOOL**

Affects how the number of frames in a file is reported in the `frames` field. If set, Prima always scans the file for the exact number of frames. Otherwise, it is up to the codec to do that.

Default: false

See also: the *frames* entry.

**Load output**

**codecID INT**

Indicates the internal codec ID used to load the image. Can be used for `Image::save`.

**frames INT**

If set to a positive integer, indicates the number of frames in the file. Otherwise signals that there are frames, but the codec needs an expensive scan to calculate the number (and the `wantFrames` parameter being set).

**height INT**

When the `noImageData` parameter is set, contains the image height.

**truncated BOOL**

When the `noIncomplete` parameter is set, will be set if the image was truncated. The value is the error string.

**width INT**

When the `noImageData` parameter is set, contains the image width.

**Saving parameters**

**autoConvert BOOL**

Affects the action when the image cannot be stored in a file in its existing pixel format. If set, the system tries to convert the image into one of the pixel formats supported by the selected codec. Fails otherwise.

Default: true

**codecID INT**

Disables the algorithm where the codec is selected based on the filename extension. Uses the codec number `codecID` instead. Note that when saving images into streams this option must always be present.

Default: undef

**BMP codec**

BMP, the bitmap codec is not dependent on external libraries and is always available.

**BitDepth INT**

Original bit depth may differ from `Image::bpp`.

Not valid as a saving parameter.

**Compression STRING**

Bitmap compression method.

Not valid as a saving parameter.

**HotSpotX, HotSpotY INT**

If loading from a cursor file, contains pointer hotspot coordinates

**ImportantColors INT**

The minimal number of colors is needed to display the image

**OS2 BOOL**

Is set when loading an OS/2 bitmap

**XResolution, YResolution INT**

Image resolution in PPM

**X11 codec**

X11, the X Consortium data file codec is implemented internally and is always available.

**hotSpotX, hotSpotY INT**

Contains pointer hotspot coordinates, if any

**XPM codec**

**extensions HASH**

A set of xpm-specific extension strings. Cannot be used for saving.

**hintsComment, colorsComment, pixelsComment STRING**

Contains comments on different sections

**hotSpotX, hotSpotY INT**

Contains the pointer hotspot coordinates

**transparentColors [COLOR]**

An array of transparent colors. Cannot be used for saving.

### JPEG codec

#### Load parameters

**exifTransform none|auto|wipe**

If set to `auto` or `wipe`, tries to detect whether there are any exif tags hinting that the image has to be rotated and/or mirrored. If found, applies the transformation accordingly.

When set to `wipe`, in addition to that, removes the exif tags so that subsequent image save won't result in transformed images with exifs tags still present.

This parameter requires a `loadExtras` parameter set because exif tags are stored in extra JPEG data.

#### Output fields of the loader and input parameters for the saver

**appdata [STRING]**

An array of raw binary strings found in extra JPEG data.

**comment STRING**

Any comment text found in the file.

**progressive BOOL**

If set, produces a progressively encoded JPEG file.

Default: 0

Only used for saving.

**quality INT**

JPEG quality, 1-100.

Default: 75

Only used for saving.

### PNG codec

#### Load input

**background COLOR**

When a PNG file contains an alpha channel, and `alpha` is set to `blend`, this color is used to blend the background. If set to `clInvalid`, the default PNG library background color is used.

Default: clInvalid

Not applicable to `Prima::Icon`.

**gamma REAL**

Override gamma value applied to the loaded image

Default: 0.45455

**screen_gamma REAL**

Current gamma value for the operating system, if specified.

Default: 2.2

**Load output and save input**

**background COLOR**

Default PNG library background color

Default: clInvalid, which means PNG library default

**blendMethod blend|no_blend|unknown**

Signals whether the new frame is to be blended over the existing animation, or it should replace that.

**delayTime $milliseconds**

Delay between frames

**default_frame BOOLEAN**

When set, means that the first image is the "default" frame, a special backward-compatibility image that is supposed to be excluded from the animation sequence, to be displayed only when all animation frames cannot be loaded for whatever reason.

**disposalMethod none|background|restore|unknown**

Signals whether the frame, before being replaced, is to be erased by the background color, or by the previous frame, or not touched at all.

**gamma REAL**

Gamma value found in the file.

Default: 0.45455

**hasAlpha BOOLEAN**

If set, the image contains an alpha channel

**iccp_name, iccp_profile STRING**

Embedded ICC color profiles in raw format

Default: `unspecified` and `""`.

**interlaced BOOL**

If set, the PNG file is interlaced

Default: 0

**left INTEGER**

The horizontal offset of the frame from the screen

**loopCount INTEGER**

How many times the animation sequence should run, or 0 for forever

**mng_datastream BOOL**

If set, the file contains an MNG datastream

Default: 0

**offset_x, offset_y INT**

A positive offset from the left edge of the screen to offset_x and the positive offset from the left edge of the screen to offset_y

Default: 0

**offset_dimension pixel|micrometer**

Offset units

Default: pixel

**render_intent none|saturation|perceptual|relative|absolute**

See the PNG docs ( the *http:www.libpng.orgpubpngspec1.1PNG-Chunks.html* entry, 4.2.9. sRGB Standard RGB color space).

Default: none

**resolution_x, resolution_y INT**

Image resolution

Default: 0

**resolution_dimension meter|unknown**

Image resolution units

Default: meter

**scale_x, scale_y**

Image scale factors

Default: 1

**scale_unit meter|radian|unknown**

Image scale factor units

Default: unknown

**screenWidth, screenHeight INTEGER**

**text HASH**

Free-text comments found in the file

Default: {}

**top INTEGER**

The vertical offset of the frame from the screen

**transparency_table [INT]**

When a paletted image contains transparent colors, returns an array of palette indexes (`transparency_table`) in the 0-255 range, where each number is an alpha value.

Default value: empty array

**transparent_color COLOR**

One transparent color value for 24-bit PNG images.

Default value: clInvalid (i.e. none)

**transparent_color_index INT**

One transparent color value, as the palette index for 8- or less-bit PNG images.

Default value: -1 (i.e. none)

Not applicable for load.

**TIFF codec**

**Load input**

**MinIsWhite BOOL**

Automatically invert `PHOTOMETRIC_MINISWHITE` images

Default: 1

**Fax BOOL**

If set, converts 1-bit grayscale with ratio 2:1 into 2-bit grayscale (algorithm also known as *faxpect*).

Default: 0

**Load output**

**Photometric STRING**

TIFF `PHOTOMETRIC_XXX` constant. One of:

```
MinIsWhite
MinIsBlack
Palette
YCbCr
RGB
LogL
LogLUV
Separated
MASK
CIELAB
DEPTH
Unknown
```

**BitsPerSample INT**

Bits used to represent a single sample, 1-64

**SamplesPerPixel INT**

Number of samples per pixel, 1-4. Most images have 1 sample. Planar TIFFs may split low and high bytes in 2 samples. RGB has 3 samples, and YCbCr and RGBA have 4.

**PlanarConfig contiguous|separate**

`separate` images split individual samples or components (f.ex. R and G and B) into individual planes. `contiguous` mix sample bytes one after another.

**SampleFormat STRING**

Pixel sample format, one of:

```
unsigned integer
signed integer
floating point
untyped data
complex signed int
complex floating point
```

**Tiled BOOL**

If set, TIFF is tiled

**Faxpect BOOL**

When the `Fax` option was set to `true`, and indeed the image was converted from 1 to 2 bits, this parameter will be set to signal this.

**CompressionType STRING**

The compression algorithm used for reading. One of:

```
NONE
CCITTRLE
CCITTFAX3
CCITTFAX4
LZW
OJPEG
JPEG
NEXT
CCITTRLEW
PACKBITS
THUNDERSCAN
IT8CTPAD
IT8LW
IT8MP
IT8BL
PIXARFILM
PIXARLOG
DEFLATE
ADOBE_DEFLATE
DCS
JBIG
SGILOG
SGILOG24
```

**Save input**

**Compression STRING**

Same values as in `CompressionType`. Different names are used to avoid implicit but impossible compression selection because tibtiff can decompress many types, but compress only a few.

**Load output and save input**

**generic strings**

The following keys have no specific meanings for Prima, but are both recognized for loading and saving:

```
Artist
Copyright
DateTime
DocumentName
HostComputer
ImageDescription
```

```
Make
Model
PageName
PageNumber
PageNumber2
```

**PageNumber, PageNumber2 INT**

Default: 1

**ResolutionUnit inch|centimeter|none**

Default: none

**Software**

Default: Prima

**XPosition, YPosition INT**

Default: 0

**XResolution, YResolution INT**

Default: 1200

### GIF codec

For GIF animation see the *Prima::Image::Animate* section.
The following load output and save input keys are recognized:

**comment STRING**

GIF comment text

**delayTime INT**

Delay in a hundredth of a second between frames

Default: 1

**disposalMethod INT**

Animation frame disposal method

```
DISPOSE_NOT_SPECIFIED    = 0; # Leave frame, let new frame draw on top
DISPOSE_KEEP             = 1; # Leave frame, let new frame draw on top
DISPOSE_CLEAR            = 2; # Clear the frame's area, revealing bg
DISPOSE_RESTORE_PREVIOUS = 3; # Restore the previous (composited) frame
```

Default: 0

**interlaced BOOL**

If set, the GIF image is interlaced

Default: 0

**left, top INT**

Frame offset in pixels

Default: 0

**loopCount INT**

How many times do the GIF animation loops. 0 means indefinite.

Default: 1

**screenBackGroundColor COLOR**

GIF screen background color

Default: 0

**screenColorResolution INT**

Default: 256

**screenWidth, screenHeight INT**

Default: -1, i.e. use image width and height

**screenPalette [INT]**

Default: 0,0,0,255,255,255

**transparentColorIndex INT**

Index of the GIF transparent color

Default: 0

**userInput INT**

User input flag

Default: 0

**WebP codec**

**Load output**

**background $ARGB_color**

An integer constant encoded as 32-bit ARGB, hints the background color to be used

**blendMethod blend|no_blend|unknown**

Signals whether the new animation frame is to be blended over the existing animation, or it should replace that.

**delayTime $milliseconds**

Delay time between frames

**disposalMethod none|background|unknown**

Signals whether the frame, before being replaced, is to be erased by the background color or not.

**hasAlpha BOOLEAN**

If set, the image contains an alpha channel

**left INTEGER**

The horizontal offset of the frame from the screen

**loopCount INTEGER**

How many times the animation sequence should run, or 0 for forever.

**screenWidth INTEGER**

**screenHeight INTEGER**

**top INTEGER**

The vertical offset of the frame from the screen

**Save input** WebP requires all images to have the same dimensions. Also, saving the webp loading result might fail because loaded frames might only contain parts to be superimposed on each other while saving always requires full frames. To convert the loaded frames to something that can be saved later more-or-less identically, use the `Prima::Image::webp::animation_to_frames` converter:

```
use Prima qw(Image::webp);
my @i = Prima::Icon->load('source.webp', loadAll => 1, loadExtras => 1) or die $@;
@i = Prima::Image::webp::animation_to_frames(@i);
die $@ if @i != Prima::Icon->save('target.webp', images => \@i);
```

**background $ARGB_color**

An integer constant encoded as 32-bit ARGB, hints the background to be used

**compression lossless (default)|lossy|mixed**

**delay $milliseconds**

**filter_strength INTEGER**

The value is between 0 and 100, where 0 means off.

**kmax INTEGER**

Min distance between keyframes. The default is 9 for the lossless compression and 3 for the lossy

**kmin INTEGER**

Max distance between keyframes. The default is 17 for the lossless compression and 5 for the lossy

**loopCount 0**

How many times the animation sequence should run, or 0 for forever.

**method INTEGER**

Compression method vs size, 0 (fast) to 6 (slow)

**minimize_size BOOLEAN**

Minimize the output size (off by default)

**quality INTEGER**

Quality factor (0:small..100:big)

**thread_level BOOLEAN**

Use multi-threading if available (off by default)

**HEIF codec**

**Load output**

**chroma_bits_per_pixel**

**depth_images**

Number of depth images available for the frame

**has_alpha**

**ispe_height, ispe_width**

Original image size before transformations (crop, rotation, etc) are applied

**is_primary**

Set if this is the primary image

**luma_bits_per_pixel**

**premultiplied_alpha**

Is set if the values of the alpha channel are premultiplied

**thumbnails**

An array of hashes with keys *type*, *content_type*, and *content*.

**aux**

**metadata**

**thumbnail_of INDEX**

Set if this frame is the thumbnail of the INDEXth top-level frame

**Save input**

**quality**

Integer, 0-100

**compression**

HEIC,AV1,AVC

**is_primary**

The first frame (#0) gets to be the primary by default, but this can be changed explicitly.

**premultiplied_alpha**

Trueset if the values of the alpha channel are premultiplied

**metadata**

An array of hashes with keys *type*, *content_type*, and *content*.

**thumbnail_of INDEX**

Sets this image as the thumbnail of the INDEXth top-level frame

## 3.8 Prima::Widget

Window management

### Synopsis

```
# create a widget
my $widget = Prima::Widget-> new(
    size    => [ 200, 200],
    color   => cl::Green,
    visible => 0,
    onPaint => sub {
       my ($self,$canvas) = @_;
       $canvas-> clear;
       $canvas-> text_out( "Hello world!", 10, 10);
    },
);

# manipulate the widget
$widget-> origin( 10, 10);
$widget-> show;
```

### Description

Prima::Widget is a descendant of the Prima::Component class, that provides comprehensive management of system-dependent windows. Objects of the Prima::Widget class are mapped to the screen space as a rectangular area, with distinct boundaries, a pointer and sometimes a cursor, and a user-selectable input focus.

### Usage

The Prima::Widget class and its descendants are used widely throughout the toolkit and are at the center of almost all its user interaction and input-output functions. The notification system, explained in the *Prima::Object* section, is heavily used in the Prima::Widget class, providing the programmer with unified access to the system-generated events that occur when for example the user moves a window, clicks the mouse, types on the keyboard, etc.

### Creation and destruction

The widget creation syntax is the same as for creating other Prima objects:

```
Prima::Widget-> new(
   name => 'Widget',
   size => [ 20, 10],
   onMouseClick => sub { print "click\n"; },
   owner => $owner,
);
```

A widget must almost always be explicitly assigned an owner. The owner object is either a Prima::Widget descendant, in which case the widget is drawn inside its inferior, or the application object so that the widget becomes a top-level screen object. This is the reason why the `insert` syntax is preferred, as it is more illustrative and is more convenient for creating several widgets in one call ( see the *Prima::Object* section ).

```
$owner-> insert( 'Prima::Widget',
   name => 'Widget',
   size => [ 20, 10],
   onMouseClick => sub { print "click\n"; },
);
```

These two examples produce identical results.

As a descendant of the Prima::Component class, Prima::Widget objects also send the `Create` notification while being created ( more precisely, after its init stage is finished. See the *Prima::Object* section for details). This notification is called and processed within the `new()` method. Another notification `Setup` is sent after the widget finishes the creation process. This message is *posted* though, i e it is invoked within the `new()` method but is processed inside the application event loop. This means that the moment when the `Setup` event is executed is uncertain, as it is with all posted messages. Its delivery is system-dependent, so its use must be considered with care.

After the widget is created, it is usually asked to render its visual content by the system, provided that the widget is visible. This request is delivered by the `Paint` notification.

When the lifetime of the widget is over, its method `destroy()` should be called. In some circumstances, the method can be also called implicitly by the toolkit. If the widget gets destroyed because its owner also gets destroyed, it is guaranteed that its widget children will be destroyed first, and the owner only afterward. In such situation the widget can still operate but with limited functionality ( see the *Prima::Object* section, the **Creation** section ).

## Graphic content

There are two ways graphics can be displayed in a widget. The first is the event-driven method when the `Paint` notification arrives, notifying the widget that it must re-paint itself. The second is the 'direct' method when the program itself begins drawing on the widget.

### Event-driven rendering

When the system decides that a widget needs to update its graphics it sends the `Paint` notification to the program. The notification has a single ( besides the widget itself ) parameter, referred to as *canvas*, the object where the drawing is performed. During the event-driven call initiated by the system, it is always the widget itself. Other callers of the `Paint` notification though can provide another object to draw on:

```
$widget-> notify('Paint', $some_other_widget);
```

When programming this notification use this parameter, not the widget itself, for the painting. An example of the `Paint` notification handler could be a simple like this:

```
Prima::Widget-> new(
    ...
    onPaint => sub {
       my ( $self, $canvas) = @_;
       $canvas-> clear;
       $canvas-> text_out("Clicked $self->{clicked} times", 10, 10);
    },
    onMouseClick => sub {
       $_[0]-> {clicked}++;
       $_[0]-> repaint;
    },
);
```

The example shows several important features of the event-driven mechanism. First, no `begin_paint()`/`end_paint()` brackets are used within the callback - these are called implicitly. Second, when the widget graphics need to be changed (after a mouse click, for example), no code like `notify(q(Paint))` is needed - the `repaint()` method is used instead. Note that the execution of `Paint` callbacks may or may not occur inside the `repaint()` call. This behavior is managed by the `::syncPaint` property. A call to `repaint()` marks the whole widget's area to be refreshed, or *invalidates* the area. For the finer access to the area that should be repainted, the functions `invalidate_rect()` and `validate_rect()` are used. Thus the call

```
$x-> repaint()
```

is identical to the

```
$x-> invalidate_rect( 0, 0, $x-> size);
```

call.

The area passed to the `invalidate_rect()` method will be accessible as the clipping rectangle inside the `Paint` notification. However, the interaction between the program logic and the system logic can result in situations where the system may request repainting of the other parts of the widget, not only those that were requested by the `invalidate_rect` call. This can happen for example when windows from another program move over the widget. In these cases, the clipping rectangle might not be exactly the same. Moreover, the clipping rectangle can even become empty as a result of these interactions, and the notification won't be called at all.

The clipping rectangle is represented differently inside and outside the drawing mode. To access the rectangle, the `::clipRect` property is used. Inside the `Paint` call (or, strictly speaking, inside the `begin_paint`/`end_paint` brackets) the rectangle is measured in the inclusive-inclusive coordinates, whereas the `invalidate_rect()`, `validate_rect()`, and `get_invalid_rect()` method use the inclusive-exclusive coordinates. Assuming the clipping rectangle is not changed by the system, the example below illustrates the difference:

```
$x-> onPaint( sub {
   my @c = $_[0]-> clipRect;
   print "clip rect:@c\n";
});
$x-> invalidate_rect( 10, 10, 20, 20);
...
clip rect: 10 10 19 19
```

The notification handler can use the `::clipRect` property to optimize the painting code, drawing only the parts that are necessary to draw.

In the drawing mode it is possible to change the `::clipRect` property, however, increasing the clipping rectangle in such a way won't make it possible to draw on the screen area that lies outside the original clipping region. This is part of the same mechanism that doesn't allow drawing outside the widget's geometric boundaries.

### Direct rendering

The direct rendering, contrary to the event-driven, is initiated by the program, not by the system. If a programmer wishes to paint over a widget immediately, then the `begin_paint()` method should be called first, and, if it is successful, the part of the screen occupied by the widget is accessible for drawing.

This method is useful, for example, for graphic demonstration programs, that draw continuously without any input. Another use of this method is the drawing directly on the screen, which is performed by entering the drawing mode on the `$::application` object, that does not have the `Paint` notification. The application's graphic canvas represents the whole screen, allowing drawing the windows that also belong to other programs.

The majority of the widget rendering code is using the event-driven drawing. Sometimes, however, the changes needed to be made to the widget's graphic context are so insignificant, so the direct rendering method is preferable, because of the cleaner and terser code. Below is an example of a simple progress bar that draws a simple colored stripe. The event-driven code would be ( in short, omitting many details ) like this:

```
$bar = Widget-> new(
  width => 100,
  onPaint => sub {
     my ( $self, $canvas) = @_;
     $canvas-> color( cl::Blue);
     $canvas-> bar( 0, 0, $self-> {progress}, $self-> height);
     $canvas-> color( cl::Back);
     $canvas-> bar( $self-> {progress}, 0, $self-> size);
  },
);
...
$bar-> {progress} += 10;
$bar-> repaint;
# or, more efficiently,
# $bar-> invalidate_rect( $bar->{progress}-10, 0,
#                 $bar->{progress}, $bar-> height);
```

While the version with the direct drawing would be

```
$bar = Widget-> new( width => 100 );
...
$bar-> begin_paint;
$bar-> color( cl::Blue);
$bar-> bar( $progress, 0, $progress + 10, $bar-> height);
$bar-> end_paint;
$progress += 10;
```

The pros and the contras are obvious: the event-driven rendered widget correctly represents the status after an eventual repaint, for example when the user sweeps a window over the progress bar widget. The direct method is not that smart, but if the status bar is an insignificant part of the program it can be used instead.

Both methods can be effectively disabled by using the locking mechanism. The `lock()` and `unlock()` methods can be called several times, counting the requests. This feature is useful because many properties implicitly call `repaint()`, and if several of these properties are called in a row or within each other, the unnecessary redrawing of the widget can be avoided by wrapping such calls in the lock/unlock brackets. The drawback is that the last `unlock()` call triggers the `repaint()` method unconditionally.

## Geometry

### Basic properties

A widget always has its position and size determined, even when it is not visible on the screen. Prima::Widget provides several properties with overlapping functionality that manage the geometry of widgets. The base properties are `::origin` and `::size`, and the derived ones are `::left`, `::bottom`, `::right`, `::top`, `::width`, `::height`, and `::rect`. `::origin` and `::size` operate on two integers, `::rect` on four, others on one integer value.

The Prima toolkit coordinate space begins in the lower bottom corner, so the combination of `::left` and `::bottom` is the same as `::origin`, and the combination of `::left`, `::bottom`, `::right` and `::top` - same as the `::rect` property.

When widgets are moved or resized, two notifications may occur, correspondingly, `Move` and `Size`. The parameters for both are the old and the new position and size. The notifications occur irrespective of whether the geometry change was issued by the program itself or by the user.

### Implicit size regulations

There exist two other properties that regulate widget size, `::sizeMin` and `::sizeMax`. They keep the minimum and the maximum sizes the widget may have. A call that would try to assign the widget size outside the `::sizeMin` and `::sizeMax` limits will fail; the widget size will always be adjusted to the limits' values.

Changes to the widget's position and size can also occur automatically if the widget's owner changes its size. The toolkit contains several implicit rules that define how exactly these changes should occur. For example, the `::growMode` property accepts a set of `gm::XXX` flags that encode this behavior. The exact meaning of the `gm::XXX` flags is not given here ( see the description to `::growMode` in the API section ), but in short, it is possible using fairly simple means to program changes to widget size and position when its owner is resized. By default, the value of the `::growMode` property is 0, so widgets don't change either their size or position on these occasions. A widget with `::growMode` set to 0 stays always in the left bottom corner of its owner. When, for example, a widget is expected to stay in the right bottom corner, or the left top corner, the `gm::GrowLoX` and `gm::GrowLoY` values must be used, correspondingly. If a widget is expected to cover its owner's lower part and change its width following the owner's, ( a horizontal scroll bar in an editor window is a good example of this behavior), the `gm::GrowHiX` value must be used.

When such implicit size change occurs, the `::sizeMin` and `::sizeMax` properties still play their part - they still do not allow the widget's size to exceed these limits. However, this algorithm has a problem, that is illustrated by the following example. Imagine a widget with the size-dependent `::growMode` set to `gm::GrowHiX`, which means that the increase or decrease of the owner width would result in a similar change in the widget. If the implicit width change would match verbatim the change of the owner's width, then the widget's size (and probably its position) will be incorrect after an attempt is made to change the widget's size to values outside the size limits.

Here's the example: let's assume that the child widget has width of a 100 pixels, its growMode property is set to `gm::GrowHiX`, and its sizeMin property is set to (95, 95). The widget's owner has a width of 200 pixels. If the owner widget changes its width from 200 to 195 and then to 190 pixels, and then back again, then one naively could expect that the child widget's width would undergo the following changes:

```
                  Owner        Child
Initial state      200          100
Shrink             195   -5      95
Shrink             190   -5      95 - as it can not be less than 95.
Grow               195   +5     100
Grow               200   +5     105
```

The situation here is fixed by introducing the *virtual size* . The `::size` property is derived from the virtual size, but while `::size` cannot exceed the size limits, the virtual size can. Moreover, it can even accept negative values. This algorithm produces the correct sizes:

```
                  Owner        Child's       Child's
                                virtual width  width
Initial state      200          100           100
Shrink             195   -5      95            95
Shrink             190   -5      90            95
Grow               195   +5      95            95
Grow               200   +5     100           100
```

### Geometry managers

The concept of geometry managers is imported from the Tk, which in turn is a port of the Tcl-Tk. The idea behind it is that the widget size and position are governed by one of the *managers*, and each manager has its own set of properties. One can select the manager by assigning the `::geometry` property one the of `gt::XXX` constants. The native ( and the default ) geometry manager is the described above grow-mode algorithm ( `gt::GrowMode`). The currently implemented Tk managers are packer ( `gt::Pack` ) and placer ( `gt::Place`). Each has its own set of options and methods, and their manuals are provided separately in the *Prima::Widget::pack* section and the *Prima::Widget::place* section ( the manpages are also imported from the Tk ).

Another concept that comes along with geometry managers is the 'geometry request size'. It is realized as a two-integer property `::geomSize`, which reflects the size deduced by some intrinsic widget knowledge. The idea is that `::geomSize` is merely a request to a geometry manager, whereas the latter changes `::size` accordingly. For example, a button might set its 'intrinsic' width in accord with the width of the text string displayed in it. If the default width for such a button is not overridden, it is assigned with such a width. By default, under the `gt::GrowMode` geometry manager, setting `::geomSize` ( and its two semi-alias properties `::geomWidth` and `::geomHeight` ) also changes the actual widget size. Moreover, when the size is passed to the Widget initialization code, the `::size` property is used to initialize `::geomSize`. Such design minimizes the confusion between the two properties, and also minimizes the direct usage of `::geomSize`, limiting it to selecting the advisory size in the internal code.

The geometry request size is useless under the `gt::GrowMode` geometry manager, but Tk managers use it extensively.

### Relative coordinates

Another geometry issue, or rather a programming technique, must be mentioned - the *relative coordinates*. It is a well-known problem, when a dialog window, developed with one font looks garbled on another system with another font. The relative coordinates technique solves this problem by introducing the `::designScale` two-integer property, the width and height of the font that was used when the dialog window was designed. With this property supplied, the position and size supplied when the widget is created on another setup using another font, are adjusted proportionally to the actual font metrics.

The relative coordinates can only be used when passing the geometry properties values, and only before the creation stage, before the widget is created. This is because the scaling calculations are made in the Prima::Widget::`profile_check_in()` method.

To use the relative coordinates technique the owner ( or the *dialog* ) widget must set its `::designScale` property to the font metrics, and the `::scaleChildren` property to 1. Widgets created with an owner that meets these requirements automatically participate in the relative coordinates scheme. If a widget must be excluded from the relative geometry applications, either the owner's property `::scaleChildren` must be set to 0, or the widget's `::designScale` must be set to `undef`. As the default `::designScale` value is `undef`, no implicit relative geometry schemes are applied by default.

The `::designScale` property is automatically propagated to the children widgets, unless the explicit `::designScale` overrides it. This is useful when a child widget is a complex widget container, and was designed on yet another setup with different font sizes.

Note: it is advised to test your applications with the Prima::Stress module that assigns a random font as the default. See the *Prima::Stress* section for more.

## Z-order

When two widgets overlap on the screen, one of these is drawn in full whereas the other only partly. Prima::Widget provides management of the *Z-axis* ordering, with these four methods: `first()`, `last()`, `next()`, and `prev()`. The methods return, correspondingly, the first and the

last widgets in the Z-order stack, and the direct neighbors of the widget ( $widget-> next-> prev always is the $widget itself given that $widget-> next exists ). If a widget is *last* that means that it is not obscured by its sibling widgets, i e the topmost one.

The Z-order can also be changed at runtime ( but not during the widget's creation). Three methods that change the Z-order: `bring_to_front()` sends the widget to the top of the stack, `send_to_back()` sends it to the bottom, and `insert_behind()` sets a widget behind another widget

Changes to Z-order trigger the `ZOrderChanged` notification.

## Parent-child relationship

By default, if a widget is the child of another widget or a window, it is clipped by its owner's boundaries and is moved together with its owner if the latter changes its position. In this case, the child's owner is also its parent.

A widget must always have an owner, however, not necessarily a parent. The `::clipOwner` which is 1 by default, is set to 0, switches the widget into the parent-less mode. That means that the widget is neither clipped nor moved together with its parent. The widget becomes parent-less, or, more strictly speaking, the screen becomes its parent. Moreover, in this mode, the widget's origin offset is calculated not from the owner's coordinates but from the screen, and clicking on the widget does not bring its owner's top-level window to the front.

The same result can be also achieved if a widget is inserted in the application object which does not have any screen visualization. A widget that belongs to the application object, has its `::clipOwner` value set to 0, and it cannot be changed.

The `::clipOwner` property opens a possibility for the toolkit widgets to live inside other programs' windows. The `::parentHandle` property can be assigned a valid system window handle, so the widget becomes a child of this window. This option has a caveat, because normal widgets are never destroyed for no reason, and likewise, top-level windows are never destroyed before their `Close` notification agrees to their destruction. When a widget is inserted into another application it must be prepared to be destroyed at any moment. It is recommended to use prior knowledge about such an application, and, even better, use one or another inter-process communication scheme to interact with it.

A widget doesn't need to do any special action to become an 'owner'. A widget that was referred to in the `::owner` property of another widget, becomes an owner of the latter automatically. The `get_widgets()` method returns the list of these children widgets, similar to the `Prima::Component::get_components()` method, but returns only Prima::Widget descendant objects.

Widgets can change their owner at any moment. The `::owner` property is both readable and writable, and if a widget is visible during the owner change, it immediately appears under different coordinates and different clipping conditions after the property change, given that its `::clipOwner` property is set to 1.

## Visibility

Widgets are created visible by default. The visibility status is managed by the `::visible` property, and its two convenience alias methods, `show()` and `hide()`.

When a widget gets hidden its geometry is not discarded; the widget retains its position and size and is still subject to all previously discussed implicit sizing issues. When the change to the `::visible` property is made, the screen is not updated immediately but in the next event loop invocation because uncovering the underlying area of a hidden widget, and repainting a newly-shown widget, both depend on the event-driven rendering functionality. If the graphic content must be updated immediately, the `update_view()` method must be called, but there's a caveat. It is obvious that if a widget is shown, the only content to be updated is its own. When a widget becomes invisible, it may uncover more than one underlying widget, and even if the uncovered widgets belong to the same program, it is unclear what widgets must be updated and when. For

practical reasons, it is enough to get one event loop passed, by calling the `yield()` method on the `$::application` object. The other notifications may pass here as well, however.

There are other kinds of visibility. A widget might be visible, but one of its owners might not. Or, a widget and its owners might be visible, but they might be overshadowed by the other windows. These conditions are returned by `showing()` and `exposed()` functions, correspondingly. So, if a widget is 'exposed', it is 'showing' and 'visible'; the `exposed()` method always returns 0 if the widget is either not 'showing' or not 'visible'. If a widget is 'showing', then it is always 'visible'. `showing()` always returns 0 if a widget is invisible.

Change to the visibility status trigger the `Hide` and `Show` notifications.

## Focus

One of the key points of any GUI system is that only one window at a time can possess a *focus*. The widget is *focused* if the keyboard input is directed to it.

Prima::Widget property `::focused` manages the focused state of the widget. It is often too powerful to be used directly, however. Its wrappers, the `::selected` and the `::current` properties are usually more convenient to operate.

The `::selected` property sets focus to a widget only if it is allowed to be focused, by consulting with the value of the `::selectable` property. When the widget is selectable, the focus may be passed to either the widget itself or to one of its ( grand-) children. For example, when 'selecting' a window with a text field by clicking on a window, one does not expect the window itself to be focused, but the text field. To achieve this and reduce unnecessary coding, the `::current` property is introduced. The 'current' widget gets precedence in getting selected over widgets that are not 'current'.

De-selecting, in turn, leaves the system in such a state when no window has input focus. There are two convenience shortcuts `select()` and `deselect()` defined, aliased to selected(1) and selected(0), correspondingly.

Within the whole GUI space, there can be only one focused widget, and in the same fashion there can be only one current widget for each owner widget. A widget can be marked as current by calling either its `::current` property or the owner widget's `::currentWidget` property. When a widget gets focused, the reassignments of the current widgets happen automatically. The reverse is also true: if a widget becomes current while it belongs to the widget tree with the focus in one of its widgets, then the focus is automatically passed to it, or down to its hierarchy if the widget itself is not selectable.

These relations between the current widget pointer and focus allow the toolkit to implement the focusing hierarchy easily. The focused widget is always on the top of the chain of its owner widgets, where each is the current widget. If, for example, a window that contains a widget that contains a focused button, becomes un-focused, and then the user selects the window again, then the button will become focused automatically.

Changes to the focus produce the `Enter` and `Leave` notifications.

The next section discusses the mouse- and keyboard-driven focusing schemes. Note that all of these work via the `::selected` property, and do not allow to focus the widgets with their `::selectable` property set to 0.

### Mouse-aided focusing

Typically when the user clicks the left mouse button on a widget, the latter becomes focused. One can note that not all widgets become focused after the mouse click - scroll bars for example. Another behavior is the one described above the window with the text field - clicking the mouse on the window focuses the text field, not the window.

Prima::Widget has also the `::selectingButtons` property, a combination of the mb::XXX ( mouse buttons ) flags. If the bits corresponding to the buttons are present there then the click of this mouse button will automatically call `::selected(1)` on the widget that received the mouse click.

Another boolean property `::firstClick` determines the behavior when the mouse button action is about to focus a widget, but the widget's top-level window is not active. The default value of `::firstClick` is 1, but if it is set otherwise, the user must click twice on the widget to get it focused. The property does not influence anything if the top-level window was already active when the click event occurred.

Due to different GUI designs, it is hardly possible to force the selection of a top-level window when the user clicked another window. The window manager or the OS can interfere, although this does not always happen, and the results may be different depending on the system. Since the primary goal of the toolkit is portability, such functionality must be considered with care. Moreover, when the user selects a window by clicking not on the toolkit-created widgets, but on the top-level window decorations, it is not possible to discern the case from any other kind of focusing.

#### Keyboard focusing

The Prima has a built-in way to navigate between the widgets using the tab and arrow keys. The tab ( and its reverse, shift-tab ) key combinations move the focus between the widgets in the same top-level group ( but not inside the same owner widget group ). The arrow keys, if the focused widget is not interested in these keystrokes, move the focus in the specified direction, if it is possible. The methods that calculate the widget to be focused depending on the keystroke are `next_tab()` and `next_positional()` ( see API for the details).

The `next_positional()` method uses the geometry of the widgets to calculate which widget is the best candidate when the user presses an arrow key. The `next_tab()` method uses the `::tabStop` and `::tabOrder` properties for this. The boolean property `::tabStop` is set to 1 by default and is used to check whether the widget is willing to participate in the tab-aided focus circulation or not. If it doesn't the `next_tab()` never returns that widget as a candidate to be selected. The value of the `::tabOrder` property value is an integer that is unique within the sibling widgets ( i e those that share the same owner ) list. That integer value is used as a simple tag when the next tab-focus candidate is considered. The default `::tabOrder` value is -1, which changes to a unique value automatically after the widget creation.

## User input

The toolkit responds to the two basic means of user input - the keyboard and the mouse. Below are the three aspects of the input handling - the event-driven, the polling, and the simulated input.

The event-driven input is the more or less natural way of communicating with the user; when the user presses the key or moves the mouse, a system event occurs and triggers the notification in one or more widgets. Polling provides the immediate state of the input devices. The polling technique is rarely chosen, primarily because of its limited usability, and because the information it provides is passed to the notification callbacks anyway. The simulated input is little more than a `notify()` call with specifically crafted parameters. It interacts with the system, by sending the event through the system API so that the input emulation can be more similar to the user actions. The simulated input functions allow the notifications to be called right away, or to be *post*'ed, delaying the notification until the next invocation of the event loop.

#### Keyboard

##### Event-driven

Keyboard input generates several notifications, where the most important are the `KeyDown` and `KeyUp`. Both have almost the same list of parameters ( see the API ) that contain the keycode, the modifier keys ( if any ) that were pressed, and an eventual character code. The algorithms that extract the meaning of the key, for example, discern between the character and functional keys, etc are not described here. The reader is advised to look at the Prima::KeySelector module which provides some convenience functions for various

transformations of the keyboard input values. And to the Prima::Edit and Prima::InputLine modules, the classes that use extensively the keyboard input. But in short, the keycode is one of the `kb::XXX` ( like, kb::F10, kb::Esc ) constants and the key modifier value is a combination of the `km::XXX` ( km::Ctrl, km::Shift) constants. The notable exception is the kb::None constant that hints that there is a character code present in the event. Some other `kb::XXX`-marked keys have the character code as well, and it is up to a programmer to decide how to treat these combinations. It is advised, however, to look at the keycode first, and then at the character code later after to decide what type of key combination the user pressed.

The `KeyDown` event has also the *repeat* integer parameter that shows the count of how many times the key was repeatedly pressed. Usually, it is set to 1, but if a widget is not able to get its portion of events between the key presses, its value can be higher. If the code doesn't check for this parameter, some keyboard input may be lost. If the code will be too complicated by introducing the repeat-value, one may consider setting the `::briefKeys` property to 0. `::briefKeys`, the boolean property, is 1 by default. If it is set to 0, it guarantees that the repeat value will always be 1, but that comes with the price of certain under-optimization. If the core `KeyDown` processing code sees a repeat value greater than 1, it simply calls the notification again.

Along with these two notifications, the `TranslateAccel` event is generated after `KeyDown`, if the focused widget is not interested in the key event. Its usage covers the eventual needs of the other widgets to read the user input, even while being out of focus. A notable example can be a button with a hotkey, that reacts on the key press when the focus is elsewhere within its top-level window. `TranslateAccel` has the same parameters as `KeyDown`, except the REPEAT parameter.

Such an out-of-focus input scheme is also used when Prima checks if a keypress event should trigger a menu item because the menu items API allows to declare hotkeys in the Menu definitions. Thus, if a descendant of the Prima::AbstractMenu class is in the widget's children tree hierarchy, then it is checked whether it contains some hotkeys that match the user input. See the *Prima::Menu* section for the details. In particular, Prima::Widget has the `::accelTable` property, a mere slot for an object that contains a table of hotkeys mapped to the custom subroutines.

### Polling

The keyboard can only be polled for the states of the modifier keys. The `get_shift_state()` method returns the state of the modifier keys as a combination of the `km::XXX` constants.

### Simulated input

There are two methods `key_up()` and `key_down()` that generate the simulated keyboard input. They accept the same parameters as the `KeyUp` and `KeyDown` notifications plus the POST boolean flag. See the *API* entry for details. These methods are convenience wrappers for the `key_event()` method, which is never used directly.

## Mouse

### Event-driven

The mouse notifications are sent when the user moves the mouse, presses the mouse buttons, or releases them. The notifications are grouped in two sets, after their function. The first set consists of the <MouseDown>, `MouseUp`, `MouseClick`, and `MouseWheel` notifications, and the second of `MouseMove`, `MouseEnter`, end `MouseLeave`.

The notifications from the first set respond to the mouse button actions. Pressing, depressing, clicking ( and double-clicking ), and turning the mouse wheel, all these actions result in the generation of the four notifications from the group. The notifications are sent

together with the mouse pointer coordinates, the button that the user was operating on, and the eventual modifier keys that were pressed, if any. In addition, the `MouseClick` notification provides an integer parameter of how many clicks occurred on the same button; that one can distinguish between the single, double, triple, etc mouse clicks. The `MouseWheel` notification provides the numeric argument that reflects how far the mouse wheel was turned. All of these notifications occur when the user operates the mouse while the mouse pointer is within the geometrical bounds of a widget. If the widget is in the *capture* mode, then these events are sent to it even if the mouse pointer is outside the widget's boundaries, and are not sent to the widgets and windows that reside under the mouse pointer.

The second set of notifications responds to the mouse pointer movements. When the pointer passes over a widget, it receives first the `MouseEnter` event, then a series of `MouseMove` events, and finally the `MouseLeave` event. The `MouseMove` and `MouseEnter` notifications provide the X,Y-coordinates and the eventual modifier keys; `MouseLeave` provides no parameters.

**Polling**

The `get_mouse_state()` method returns a combination of the `mb::XXX` constants. The `::pointerPos` two-integer property reflects the current position of the mouse pointer.

**Simulated input**

There are five methods, - `mouse_up()`, `mouse_down()`, `mouse_click()`, `mouse_wheel()`, and `mouse_move()`, that accept the same parameters as their event counterparts do, plus the POST boolean flag. See the *API* entry for details.

These methods are convenience wrappers for the `mouse_event()` method that is never used directly.

**Drag and drop**

Widgets can participate in drag-and-drop sessions, interacting with other applications as well as with themselves, with very few restrictions. See below how to use this functionality.

**Data exchange**

Prima defines a special clipboard object that serves as an exchange agent whenever data is to be either sent or received in a DND session. To either offer to or choose from many formats that another DND client can work with, use this clipboard (see more in the *Prima::Clipboard* section). The DND clipboard can be accessed at any time by calling the `$::application->get_dnd_clipboard` method.

To successfully exchange data with other applications, one should investigate the results of a `$clipboard-> get_formats(1)` call to see what types of data the selected application can send or accept. Programs can often exchange text and images in the system-dependent format, and other data in the formats named after the MIME type of the data. For example, Prima supports image formats like `image/bmp` out of the box, and `text/plain` on X11, which are selected automatically when operating with pseudo-formats `Text` or `Image`. Other MIME formats like f.ex. `text/html` are not known to Prima, but can be exchanged quite easily; the program only needs to register these formats by calling the `Clipboard::register_format` method at the start of the program.

**Dragging**

To begin a dragging session first fill the DND clipboard with data to be exchanged, using one or more formats, then call the the *start_dnd* entry method. Alternatively, call the *begin_drag* entry, a wrapper method that can set up the necessary clipboard data itself. See the documentation on these methods for more details.

During the dragging, the sender will receive the the *DragQuery* entry and the *DragResponse* entry events, to decide whether the drag session must continue or stop depending on the

user interactions, and reflect that decision to the user. Traditionally, mouse pointers are changed to show whether an application will receive dropped data, and if yes, what action (copy, move, or link) it will recognize. Prima will try its best to either use system pointers or synthesize ones that are informative enough; if that is not sufficient, one may present its own pointer schema (see f.ex how `begin_drag` is implemented).

**Dropping**

To register a widget as a drop target, set its the *dndAware* entry property to either 1, to mark that it will answer to every format, or to a string, in which case drop events will only be delivered if the DND clipboard contains a format with that string as its name.

When the user initiates a DND session and moves the mouse pointer over the widget, it receives the related events: first a the *DragBegin* entry event, then a series of the the *DragOver* entry events, and finally a the *DragEnd* entry event with a flag telling whether the user chose to drop the data to the widget or cancel the session.

The `DragOver` and `DragEnd` callbacks provide the possibility to either allow or deny data and select an action (if there is more than one allowed by the other application) to proceed with. To do so, set appropriate values to the {`allow`} and the {`action`} fields in the last hashref parameter that is sent to these event handlers. Additionally, the program can respond to the `DragOver` by setting the {`pad`} rectangle that will cache the last answer and tell the system to not send repeated events with the same input while the mouse pointer stays in the rectangle.

**Portability**

X11 and Win32 are rather identical in how they handle DND sessions from the user perspective. The only difference that is significant to Prima here is whether the sender or the receiver is responsible for selecting an action for the available list of actions when the user presses the modifier keys, like CTRL or SHIFT.

On X11, it is the sender that controls that aspect, and tells the receiver what action at any given moment the user chose, by responding to a `DragQuery` event. On Win32, it is the receiver that selects an action from the list on each `DragOver` event, depending on the modifier keys pressed by the user; Win32 recommends adhering to the standard scheme where the CTRL key means the `dnd::Move` action, and the SHIFT key the `dnd::Link` action, but that is up to the receiver.

Thus, to write a robust and portable program, assume that it may control the actions both as the sender and as the receiver. The toolkit's system-dependent code will make sure that there will be no ambiguities in the input. F.ex. the sender on Win32 will never be presented with combination of several `dnd::` constants inside a `DragQuery` event, and the X11 receiver will similarly never be presented with such combination inside `DragOver`. Nevertheless, a portable program must be prepared to select and return a DND action in either callback.

Additionally, the X11 DND protocol describes the situation when the receiver is presented with the choice of actions, and may also ask the user what action to select, or cancel the session altogether. This is okay and is expected by the user, and it is up to your program to use that possibility or not.

## Colors

Prima::Widget extends the functionality of `::color` and `::backColor`, the properties inherited from the Prima::Drawable class. Their values are the widget's 'foreground' and 'background' colors, in addition to their function as template values. Moreover, their dynamic change induces the repainting of the widget. The values of these properties can be inherited from the owner; the inheritance is managed by the properties `::ownerColor` and `::ownerBackColor`. If these are set to 1 then changes to the owner's properties `::color` or `::backColor` are copied automatically to

the widget. Once the widget's `::color` or `::backColor` is explicitly set, the owner link breaks automatically by setting `::ownerColor` or `::ownerBackColor` to 0.

In addition to these two existing color properties, Prima::Widget introduces six others. These are: `::disabledColor`, `::disabledBackColor`, `::hiliteColor`, `::hiliteBackColor`, `::light3DColor`, and `::dark3DColor`. The 'disabled' color pair contains the values that are expected to be used as the foreground and the background when the widget is in the disabled state ( see API, `::enabled` property ). The 'hilite' values serve as colors painting a selection inside of the widget. Selection may be of any kind, and some widgets do not provide any. But for those that do, the 'hilite' color values provide distinct alternative colors. The examples are the selections in the text widgets or the list boxes. The last pair, `::light3DColor` and `::dark3DColor` is used for drawing 3D-bevelled outlines on the widget. The purpose of all these properties is to respect the system colors and draw the Prima GUI as close as possible to the native system look.

There are eight additional `cl::` constants that can be used to access these system colors. These named correspondingly, cl::NormalText, cl::Normal, cl::HiliteText, cl::Hilite, cl::DisabledText, cl::Disabled, cl::Light3DColor, and cl::Dark3DColor. The cl::NormalText constant is an alias to cl::Fore, and cl::Normal - to the cl::Back constant. Another constant set, `ci::` can be used with the `::colorIndex` property, a multiplexer method for all the eight color properties. `ci::` constants mimic their non-RGB `cl::` counterparts, so that a call to `hiliteBackColor(cl::Red)` is equal to `colorIndex(ci::Hilite, cl::Red)`.

The `map_color` translates these special constants to the 24-bit RGB integer values. The `cl::` constants alone are sufficient for acquiring the default values, but the toolkit provides even wider functionality to address default colors for different types of widgets. The `cl::` constants can be combined with the `wc::` constants, that represent the standard widget classes. If the color property was assigned with a `cl::` constant not combined with a `wc::` constant, the widget class value is read from the `::widgetClass` property. Thus a call to, for example, `backColor( cl::Back)` on a button and an input line may result in different colors because the `cl::Back` is translated in the first case to `cl::Back|wc::Button`, and in another to `cl::Back|wc::InputLine`. The `wc::` constants are described in the *API* entry.

Dynamic changes of the color properties result in the `ColorChanged` notification.

## Fonts

The default font can be automatically inherited from the owner if the `::ownerFont` property is set to 1. If it is set to 0, then the font returned by the `get_default_font` method is used instead. The method may return different fonts depending on the widget class, name, and user preferences ( see the *Additional resources* entry ). A similar method `get_default_popup_font` is used to query the default popup font and the `::popupFont` property for accessing it. The Prima::Window class has also similar functions, the `get_default_menu_font` method and the `::menuFont` property.

Dynamic changes to the font property result in the `FontChanged` notification.

## Additional resources

The resources operated via the Prima::Widget class but not that strictly bound to the widget concept are gathered in this section. The section includes an overview of pointer, cursor, hint, menus, and user-specified resources.

### Markup text

The `Prima::Drawable::Markup` class provides text-like objects that can draw rich text with various fonts and colors and has primitive support for painting images. The methods of `Prima::Drawable` that handle text output such as `text_out`, and `get_text_width`, etc can detect if the text passed is a blessed object, and make a corresponding call on it. The markup objects can employ this mechanism to be used transparently in the `text` and the `hint` properties.

There are two ways to construct a markup object: either directly:

```
Prima::Drawable::Markup->new( ... )
```

or using an imported method `M`,

```
use Prima::Drawable::Markup q(M);
M '...';
```

where the results of both calls can be directly set to almost any textual property throughout the whole toolkit, provided that the classes are not peeking inside the object but only calling the drawing methods on them.

In addition to that, the `Prima::Widget` class and its descendants recognize the third syntax:

```
Widget->new( text => \ 'markup' )
```

treating a scalar reference to a text string as a sign that this is the text to be compiled into a markup object.

### Pointer

The mouse pointer is the shared resource that can change its visual representation when it hovers over different kinds of widgets. It is usually a good practice for a text field, for example, to set the pointer icon to a vertical line, or indicate a moving window with a cross-arrow pointer.

A widget can select any of the predefined system pointers mapped by the `cr::XXX` constant set, or supply own pointer icon of arbitrary size and color depth.

NB: Not all systems support colored pointer icons. The system value sv::ColorPointer index contains a boolean value, whether the colored icons are allowed or not. Also, the pointer icon size may have a limit: check if sv::FixedPointerSize is non-zero, in which case the pointer size will be forcibly reduced to the system limits.

In general, the `::pointer` property is enough to access these functions of the mouse pointer. The property can deduce whether it is an icon or a constant passed and set the appropriate system properties. These properties are also accessible separately, although their usage is not encouraged, primarily because of the tangled relationship between them. These properties are: `::pointerType`, `::pointerIcon`, and `::pointerHotSpot`. See the details in the the *API* entry sections.

Another property called `Prima::Application::pointerVisible` manages the visibility of the mouse pointer for all widgets at once.

### Cursor

The cursor is a blinking rectangular area that signals that the widget has the input focus. There can be only one active cursor or no active cursor at all. The Prima::Widget class provides several cursor properties: `::cursorVisible`, `::cursorPos`, and `::cursorSize`. There are also two methods, `show_cursor()` and `hide_cursor()` that govern the cursor visibility. Note: If the `hide_cursor()` method was called three times, then `show_cursor()` must be called three times as well for the cursor to become visible.

### Hints

`::hint` is a text string that describes the widget's purpose to the user in a terse manner. If the mouse pointer hovers over the widget longer than some timeout ( see Prima::Application::hintPause ), then a small *tooltip* window appears with the hint text, which stays on the screen until the pointer is drawn away. The hint behavior is managed by Prima::Application, but a widget can do two additional things about its hint: it can enable and disable it by setting the `::showHint` property, and it can inherit the owner's `::hint` and `::showHint` properties using the `::ownerHint` and `::ownerShowHint` properties. If, for example, the widgets' `::ownerHint` property is set to 1, then the `::hint` value is automatically copied from the widget's owner when it changes. If, however,

the widget's `::hint` or `::showHint` are explicitly set, the owner link breaks automatically by setting `::ownerHint` or `::ownerShowHint` to 0.

The widget can also operate the `::hintVisible` property that shows or hides the hint label immediately if the mouse pointer is inside the widget's boundaries.

### Menu objects

Prima::Widget objects may have a special relationship with registered object instances of the Prima::AccelTable and Prima::Popup class ( for Prima::Window this is also valid for Prima::Menu objects ). The registration and/or automatic creation of these objects can happen by using the `::accelTable`, `::popup`, and `::menu` properties. Also the `::items` property of these objects can also be accessed via the `::accelItems`, `::popupItems`, and `::menuItems` properties. As mentioned in the *User input* entry, these objects intercept the user keyboard input and call the programmer-defined callback subroutine if the keystroke matches one of their key definitions. `::popup` provides access to a context pop-up menu, which can be invoked by either right-clicking the mouse or pressing a system-dependent key combination.

The widget also provides the `::popupColorIndex` and `::popupFont` properties. The multiplexer method `::popupColorIndex` can be also used to access the `::popupColor`, `::popupHiliteColor`, `::popupHiliteBackColor`, etc properties exactly like the `::colorIndex` property. The Prima::Window class provides equivalent methods for the menu bar, introducing `::menu`, `::menuItems`, `::menuColorIndex`, and `::menuFont` properties.

Win32 doesn't support custom font and color of the menu and popup objects. Check the the *Prima::Menu* section for the implementation of the menu widgets without using the system menu objects.

### User-specified resources

It is considered a good idea to incorporate user preferences into the toolkit look and feel. Prima::Widget relies on the system-specific code that tries to map these preferences as closely as possible to the toolkit.

The X11 backend uses XRDB ( X resource database ) which is the natural (but mostly obsolete as of now) way for the user to tell the preferences with fine granularity. Win32 reads the setting that the user has to set interactively, using system tools. Nevertheless, the toolkit can not emulate all user settings that are available on the supported platforms; it rather takes the 'least common denominator', which is colors and fonts only. The `fetch_resource()` method is capable of accessing such settings, in font, color, or a generic text format. The method is rarely called directly.

A somewhat appealing idea of making every widget property adjustable via the user-specified resources is not implemented in full. It can be accomplished up to a certain degree using the `fetch_resource()` method, but it is believed that calling the method for every property on every widget is prohibitively expensive.

## API

### Properties

**accelItems [ ITEM_LIST ]**

Manages items of a Prima::AccelTable object associated with a widget. The ITEM_LIST format is the same as `Prima::AbstractMenu::items` and is described in the *Prima::Menu* section.

See also: `accelTable`

**accelTable OBJECT**

Manages a Prima::AccelTable object associated with a widget. The sole purpose of the accelTable object is to provide convenience mapping of key combinations to anonymous subroutines. Instead of writing an interface specifically for Prima::Widget, the existing interface of Prima::AbstractMenu was taken.

The accelTable object can be destroyed safely; its cancellation can be done either via `accelTable(undef)` or `destroy()` call.

Default value: undef

See also: `accelItems`

**autoEnableChildren BOOLEAN**

If TRUE, all immediate children widgets maintain the same `enabled` state as the widget. This property is useful for group-like widgets ( ComboBox,

Default value: 0

**backColor COLOR**

In the paint state, manages the background color of the graphic context. In the normal state, manages the background color property. When changed initiates the `ColorChanged` notification and repaints the widget.

See also: `color`, `colorIndex`, `ColorChanged`

**bottom INTEGER**

Maintains the lower boundary of the widget. If changed does not affect the widget height, however does so if called in the `set()` method together with the `::top` property.

See also: `left`, `right`, `top`, `origin`, `rect`, `growMode`, `Move`

**briefKeys BOOLEAN**

If 1 compresses the repetitive `KeyDown` events into a single event and reports the number of the events compressed in the REPEAT parameter. If 0 the REPEAT parameter is always 1.

Default value: 1

See also: `KeyDown`

**buffered BOOLEAN**

If 1, request the system to allocate a memory buffer for painting the widget. The memory content is copied to the screen then. Used when complex drawing methods are used, or if output smoothness is desired.

This behavior can not be always granted, however. If there is not enough memory then the widget draws in the usual manner. One can check whether the buffering request is granted by calling the `is_surface_buffered` method.

Default value: 0

See also: `Paint`, the *is_surface_buffered* entry.

**capture BOOLEAN, CLIP_OBJECT = undef**

Manipulates capturing of the mouse events. If 1, the mouse events are not passed to the widget the mouse pointer is over but is redirected to the caller widget. The call for capture might not be always granted due to the race conditions between programs.

If the CLIP_OBJECT widget is defined in the set-mode call, the pointer movements are confined to CLIP_OBJECT inferior.

See also: `MouseDown`, `MouseUp`, `MouseMove`, `MouseWheel`, `MouseClick`.

**centered BOOLEAN**

A write-only property. Once set the widget is centered by X and Y axis relative to its owner.

See also: `x_centered`, `y_centered`, `growMode`, `origin`, `Move`.

**clipChildren BOOLEAN**

Affects the drawing mode when the children widgets are present and obscuring the drawing area. If set, the children widgets are automatically added to the clipping area, and drawing over them will not happen. If unset, the painting can be done over the children widgets.

Default: 1

**clipOwner BOOLEAN**

If 1, the widget is clipped by its owner boundaries. It is the default and expected behavior. If clipOwner is 0, the widget behaves differently: it does not get clipped by the owner, it is not moved together with the parent, the origin offset is calculated not from the owner's coordinates but from the screen, and mouse events in the widget do not transgress to the top-level window decorations. In short, it becomes a top-level window, that, contrary to the one created from the Prima::Window class, does not have any interference with the system-dependent window stacking and positioning ( and any other ) policy, and it is neither equipped with the window manager decorations.

Default value: 1

See the *Parent-child relationship* entry

See also: `Prima::Object` owner section, `parentHandle`

**color COLOR**

In the paint state manages, the foreground color of the graphic context. In the normal state, manages the basic foreground color property. When changed initiates `ColorChanged` notification and repaints the widget.

See also: `backColor`, `colorIndex`, `ColorChanged`

**colorIndex INDEX, COLOR**

Manages the basic color properties indirectly by accessing them via the `ci::XXX` constants. Is a complete alias for `::color`, `::backColor`, `::hiliteColor`, `::hiliteBackColor`, `::disabledColor`, `::disabledBackColor`, `::light3DColor`, and `::dark3DColor` properties. The `ci::XXX` constants are:

```
ci::NormalText or ci::Fore
ci::Normal or ci::Back
ci::HiliteText
ci::Hilite
ci::DisabledText
ci::Disabled
ci::Light3DColor
ci::Dark3DColor
```

The non-RGB `cl::` constants, specific to the Prima::Widget color usage are identical to their `ci::` counterparts:

```
cl::NormalText or cl::Fore
cl::Normal or cl::Back
cl::HiliteText
cl::Hilite
```

```
cl::DisabledText
cl::Disabled
cl::Light3DColor
cl::Dark3DColor
```

See also: `color`, `backColor`, `ColorChanged`

**current BOOLEAN**

If 1, the widget (or one of its children) is marked as the one to be selected and possibly focused when the owner widget receives the `select()` call. Only one children widget can be current, or none at all.

See also: `currentWidget`, `selectable`, `selected`, `selectedWidget`, `focused`

**currentWidget OBJECT**

Points to the children widget that is to be selected and possibly focused when the owner widget receives the `select()` call.

See also: `current`, `selectable`, `selected`, `selectedWidget`, `focused`

**cursorPos X_OFFSET Y_OFFSET**

Specifies the lower left corner of the cursor

See also: `cursorSize`, `cursorVisible`

**cursorSize WIDTH HEIGHT**

Specifies width and height of the cursor

See also: `cursorPos`, `cursorVisible`

**cursorVisible BOOLEAN**

Specifies the cursor visibility flag. The default value is 0.

See also: `cursorSize`, `cursorPos`

**dark3DColor COLOR**

The color used to draw dark shades.

See also: `light3DColor`, `colorIndex`, `ColorChanged`

**designScale X_SCALE Y_SCALE**

The width and height of the font that was used when the widget ( usually a dialog or a grouping widget ) was designed.

See also: `scaleChildren`, `width`, `height`, `size`, `font`

**disabledBackColor COLOR**

The color to be used instead of the value of the `::backColor` property when the widget is in the disabled state.

See also: `disabledColor`, `colorIndex`, `ColorChanged`

**disabledColor COLOR**

The color to be used instead of the value of the `::color` property when the widget is in the disabled state.

See also: `disabledBackColor`, `colorIndex`, `ColorChanged`

**dndAware 0 | 1 | Format**

To register the widget as a drop target, set its the *dndAware* entry property to either 1, to mark that it will answer to all formats, or to a text string, in which case the drop events will only be delivered if the DND clipboard contains the data of the type Format.

Default: 0

See also: `Drag and Drop`

**enabled BOOLEAN**

Specifies if the widget can accept focus, the keyboard, and the mouse events. The default value is 1, however, being 'enabled' does not automatically allow the widget to become focused. Only the reverse is true - if enabled is 0, focusing can never happen.

See also: `responsive`, `visible`, `Enable`, `Disable`

**font %FONT**

Manages the font context. Same syntax as in Prima::Drawable. When changed initiates `FontChanged` notification and repaints the widget.

See also: `designScale`, `FontChanged`, `ColorChanged`

**geometry INTEGER**

Selects one of the available geometry managers. The corresponding integer constants are:

```
gt::GrowMode, gt::Default - the default grow-mode algorithm
gt::Pack                  - Tk packer
gt::Place                 - Tk placer
```

See `growMode`, the *Prima::Widget::pack* section, the *Prima::Widget::place* section.

**growMode MODE**

Specifies the widget's behavior, when its owner is resized or moved. MODE can be 0 ( default ) or a combination of the following constants:

**Basic constants**

| | |
|---|---|
| `gm::GrowLoX` | `the widget's left side is kept in constant distance from its owner's right side` |
| `gm::GrowLoY` | `the widget's bottom side is kept in constant distance from its owner's top side` |
| `gm::GrowHiX` | `the widget's right side is kept in constant distance from its owner's right side` |
| `gm::GrowHiY` | `the widget's top side is kept in constant distance from its owner's top side` |
| `gm::XCenter` | `the widget is kept in the center of its owner's horizontal axis` |
| `gm::YCenter` | `the widget is kept in the center of its owner's vertical axis` |
| `gm::DontCare` | `widgets origin is constant relative to the screen` |

**Derived or aliased constants**

| | |
|---|---|
| `gm::GrowAll` | `gm::GrowLoX\|gm::GrowLoY\|gm::GrowHiX\|gm::GrowHiY` |
| `gm::Center` | `gm::XCenter\|gm::YCenter` |
| `gm::Client` | `gm::GrowHiX\|gm::GrowHiY` |
| `gm::Right` | `gm::GrowLoX\|gm::GrowHiY` |
| `gm::Left` | `gm::GrowHiY` |
| `gm::Floor` | `gm::GrowHiX` |

See also: `Move`, `origin`

**firstClick BOOLEAN**

If 0, the widget ignores the first mouse click if the top-level window it belongs to was not activated, so selecting such a widget with a mouse must take two clicks.

Default: 1

See also: `MouseDown`, `selectable`, `selected`, `focused`, `selectingButtons`

**focused BOOLEAN**

On the get-call returns whether the widget possesses the input focus or not. On the set-call sets the focus to the widget, ignoring the `::selectable` property.

See also: `selectable`, `selected`, `selectedWidget`, `KeyDown`

**geomWidth, geomHeight, geomSize**

The three properties that manage the geometry request size. Writing and reading to the `::geomWidth` and `::geomHeight` properties is equivalent to doing the same to the `::geomSize` property. The properties are run-time only, and behave differently under different circumstances:

- The properties can only be used after widget creation, they can not be set in the creation profile, and their initial value is fetched from the `::size` property. Thus, setting the explicit size additionally sets the advised size in case the widget is to be used with the Tk geometry managers.
- Setting the properties under the `gt::GrowMode` geometry manager also sets the corresponding `::width`, `::height`, or `::size` properties. When the properties are read though, the widget size properties are not accessed, their values are kept separately.
- Setting the properties under Tk geometry managers causes the widget's size and position to change according to the geometry manager policy.

**height**

Maintains the height of the widget.

See also: `width`, `growMode`, `Move`, `Size`, `get_virtual_size`, `sizeMax`, `sizeMin`

**helpContext STRING**

A text string that binds the widget to the interactive help topic. STRING format is defined as a POD link ( see *perlpod* ) - "manpage/section", where 'manpage' is the file with POD content and 'section' is the topic inside the manpage.

See also: `help`

**hiliteBackColor COLOR**

The color to be used to draw the alternate background areas with a higher contrast.

See also: `hiliteColor`, `colorIndex`, `ColorChanged`

**hiliteColor COLOR**

The color to be used to draw the alternate foreground areas with a higher contrast.

See also: `hiliteBackColor`, `colorIndex`, `ColorChanged`

**hint TEXT**

A text string is shown under the mouse pointer if it is hovered over the widget longer than the `Prima::Application::hintPause` timeout. The text appears only if the `::showHint` is 1.

TEXT can also be a `Prima::Drawable::Markup` object

See also: `hintVisible`, `showHint`, `ownerHint`, `ownerShowHint`

**hintVisible BOOLEAN**

Returns the hint visibility status when called in the get-form. When called in the set-form immediately turns on or off the hint label, disregarding the timeouts. It does regard the mouse pointer location though and does not turn on the hint label if the pointer is not immediately over the widget.

See also: `hint`, `showHint`, `ownerHint`, `ownerShowHint`

**layered BOOLEAN**

If set, requests the system to use the alpha transparency. Depending on the system and its configuration this request may or may not be granted. The actual status of the request success is returned by the `is_surface_layered` method. See the **Layering** entry in the *Prima::Image* section for more details.

Default: false

Note: On Windows mouse events will not be delivered to the layered widget if the pixel under the mouse pointer is fully transparent.

On X11 you need to run a composition manager, f.ex. *compiz* or *xcompmgr*.

On Darwin/XQuartz the alpha transparency is unavailable (2023).

**left INTEGER**

Maintains the left boundary of the widget. If changed does not affect the widget width, however does so if called in the `set()` method together with the `::right` property.

See also: `bottom`, `right`, `top`, `origin`, `rect`, `growMode`, `Move`

**light3DColor COLOR**

The color to draw light shades.

See also: `dark3DColor`, `colorIndex`, `ColorChanged`

**ownerBackColor BOOLEAN**

If 1, the background color is synchronized with the owner's. Automatically resets to 0 if the `::backColor` property is set explicitly.

See also: `ownerColor`, `backColor`, `colorIndex`

**ownerColor BOOLEAN**

If 1, the foreground color is synchronized with the owner's. Automatically resets to 0 if the `::color` property is set explicitly.

See also: `ownerBackColor`, `color`, `colorIndex`

**ownerFont BOOLEAN**

If 1, the font is synchronized with the owner's. Automatically resets to 0 if the `::font` property is set explicitly.

See also: `font`, `FontChanged`

**ownerHint BOOLEAN**

If 1, the hint is synchronized with the owner's. Automatically resets to 0 if the `::hint` property is set explicitly.

See also: `hint`, `showHint`, `hintVisible`, `ownerShowHint`

**ownerShowHint BOOLEAN**

If 1, the show hint flag is synchronized with the owner's. Automatically resets to 0 if the `::showHint` property is set explicitly.

See also: `hint`, `showHint`, `hintVisible`, `ownerHint`

**ownerPalette BOOLEAN**

If 1, the palette array is synchronized with the owner's. Automatically resets to 0 if the `::palette` property is set explicitly.

See also: `palette`

**ownerSkin BOOLEAN**

If 1, the `skin` property is set to undef and thus will be synchronized with the owner's. Automatically resets to 0 if the `::skin` property is set explicitly.

See also: `skin`

**origin X Y**

Maintains the left and bottom boundaries of the widget relative to its owner ( or to the screen if the `::clipOwner` property is 0 ).

See also: `bottom`, `right`, `top`, `left`, `rect`, `growMode`, `Move`

**packInfo %OPTIONS**

See the *Prima::Widget::pack* section

**palette [ @PALETTE ]**

Manages the array of colors that are desired to be present in the system palette, as close to the PALETTE as possible. This property works only if the graphic device allows palette operations. See the **palette** entry in the *Prima::Drawable* section.

See also: `ownerPalette`

**parentHandle SYSTEM_WINDOW**

If the SYSTEM_WINDOW is a valid system-dependent window handle then a widget becomes the child of the window specified, given the widget's `::clipOwner` is 0. The parent window may belong to another application.

Default value is undef.

See also: `clipOwner`

**placeInfo %OPTIONS**

See the *Prima::Widget::place* section

**pointer cr::XXX or ICON**

Specifies the pointer icon by either one of the `cr::XXX` constants or an icon. If the icon contains a hash variable `__pointerHotSpot` with an array of two integers, these integers will be treated as the pointer hot spot. In the get-mode call, this variable is automatically assigned to an icon if the result is an icon object.

See also: `pointerHotSpot`, `pointerIcon`, `pointerType`

**pointerHotSpot X_OFFSET Y_OFFSET**

Specifies the hot spot coordinates of the pointer icon associated with the widget.

See also: `pointer`, `pointerIcon`, `pointerType`

**pointerIcon ICON**

Specifies the pointer icon associated with the widget.

See also: `pointerHotSpot`, `pointer`, `pointerType`

**pointerPos X_OFFSET Y_OFFSET**

Specifies the mouse pointer coordinates relative to the widget's coordinates.

See also: `get_mouse_state`, `screen_to_client`, `client_to_screen`

**pointerType TYPE**

Specifies the type of the pointer associated with the widget. The TYPE can accept one constant of the `cr::XXX` constants:

```
cr::Default                 same pointer type as owner's
cr::Arrow                   arrow pointer
cr::Text                    text entry cursor-like pointer
cr::Wait                    hourglass
cr::Size                    general size action pointer
cr::Move                    general move action pointer
cr::SizeWest, cr::SizeW     right-move action pointer
cr::SizeEast, cr::SizeE     left-move action pointer
cr::SizeWE                  general horizontal-move action pointer
cr::SizeNorth, cr::SizeN    up-move action pointer
cr::SizeSouth, cr::SizeS    down-move action pointer
cr::SizeNS                  general vertical-move action pointer
cr::SizeNW                  up-right move action pointer
cr::SizeSE                  down-left move action pointer
cr::SizeNE                  up-left move action pointer
cr::SizeSW                  down-right move action pointer
cr::Invalid                 invalid action pointer
cr::DragNone                pointer for an invalid dragging target
cr::DragCopy                pointer to indicate that a dnd::Copy action can be accepted
cr::DragMove                pointer to indicate that a dnd::Move action can be accepted
cr::DragLink                pointer to indicate that a dnd::Link action can be accepted
cr::Crosshair               the crosshair pointer
cr::UpArrow                 arrow directed upwards
cr::QuestionArrow           question mark pointer
cr::User                    user-defined icon
```

All constants except the `cr::User` and the `cr::Default` represent the system-defined pointers, their icons, and hot spot offsets. `cr::User` is a constant that tells the system that an icon object was specified explicitly via the `::pointerIcon` property. The `cr::Default` constant tells that the widget inherits its owner pointer type, no matter if is it a system-defined pointer or a custom icon.

See also: `pointerHotSpot`, `pointerIcon`, `pointer`

**popup OBJECT**

Manages a Prima::Popup object associated with the widget. The purpose of the popup object is to show the context menu when the user right-clicks or selects the corresponding keyboard combination. Prima::Widget can host many popup objects but only the one that is registered in the `::popup` property will be activated automatically.

The popup object can be destroyed safely; can be done either via a `popup(undef)` or a `destroy()` call.

See also: `Prima::Menu`, `Popup`, `Menu`, `popupItems`, `popupColorIndex`, `popupFont`

**popupColorIndex INDEX, COLOR**

Maintains eight color properties of the pop-up context menu, associated with the widget. INDEX must be one of the `ci::XXX` constants ( see `::colorIndex` property ).

See also: `popupItems`, `popupFont`, `popup`

**popupColor COLOR**

Basic foreground color in the popup context menu color.

See also: `popupItems`, `popupColorIndex`, `popupFont`, `popup`

**popupBackColor COLOR**

Basic background color in the popup context menu color.

See also: `popupItems`, `popupColorIndex`, `popupFont`, `popup`

**popupDark3DColor COLOR**

The color for drawing dark shades in the popup context menu.

See also: `popupItems`, `popupColorIndex`, `popupFont`, `popup`

**popupDisabledColor COLOR**

The foreground color for the disabled items in the popup context menu.

See also: `popupItems`, `popupColorIndex`, `popupFont`, `popup`

**popupDisabledBackColor COLOR**

The background color for the disabled items in the popup context menu.

See also: `popupItems`, `popupColorIndex`, `popupFont`, `popup`

**popupFont %FONT**

Maintains the font of the pop-up context menu associated with the widget.

See also: `popupItems`, `popupColorIndex`, `popup`

**popupHiliteColor COLOR**

The foreground color for the selected items in the popup context menu.

See also: `popupItems`, `popupColorIndex`, `popupFont`, `popup`

**popupHiliteBackColor COLOR**

The background color for the selected items in the popup context menu.

See also: `popupItems`, `popupColorIndex`, `popupFont`, `popup`

**popupItems [ ITEM_LIST ]**

Manages items of the Prima::Popup object associated with the widget. The ITEM_LIST format is the same as `Prima::AbstractMenu::items` and is described in the *Prima::Menu* section.

See also: `popup`, `popupColorIndex`, `popupFont`

**popupLight3DColor COLOR**

The color for drawing light shades in the popup context menu.

See also: `popupItems`, `popupColorIndex`, `popupFont`, `popup`

**rect X_LEFT_OFFSET Y_BOTTOM_OFFSET X_RIGHT_OFFSET Y_TOP_OFFSET**

Maintains the rectangular boundaries of the widget relative to its owner ( or to the screen if `::clipOwner` is 0 ).

See also: `bottom`, `right`, `top`, `left`, `origin`, `width`, `height`, `size` `growMode`, `Move`, `Size`, `get_virtual_size`, `sizeMax`, `sizeMin`

**right INTEGER**

Maintains the right boundary of the widget. If changed does not affect the widget width, however does so if called in the `set()` method together with the `::left` property.

See also: `left`, `bottom`, `top`, `origin`, `rect`, `growMode`, `Move`

**scaleChildren BOOLEAN**

If the widget has the `::scaleChildren` property set to 1, then the newly-created children widgets inserted in it will be scaled corresponding to the value of its owner's `::designScale` property, given that widget's `::designScale` is not `undef` and the owner's is not [0,0].

Default: 1

See also: `designScale`

**selectable BOOLEAN**

If 1, the widget can be granted focus by the toolkit or by the user. The `select()` method checks if this property is set, and does not focus a widget that has `::selectable` set to 0.

Default: 0

See also: `current`, `currentWidget`, `selected`, `selectedWidget`, `focused`

**selected BOOLEAN**

In the get-mode returns true if either the widget or one of its (grand-) children is focused. In the set-mode either turns the system with no-focus state ( if a value of 0 is given ) or re-sends input focus to itself or one of the (grand-) children widgets down the `::currentWidget` chain.

See also: `current`, `currentWidget`, `selectable`, `selectedWidget`, `focused`

**selectedWidget OBJECT**

Points to the immediate child widget that has the value of the property `::selected` set to 1.

See also: `current`, `currentWidget`, `selectable`, `selected`, `focused`

**selectingButtons FLAGS**

The FLAGS is a combination of the `mb::XXX` ( mouse button ) flags. If the widget receives a mouse click with the button that has the corresponding bit set in `::selectingButtons` then the `select()` method is called.

Default: `mb::Left`

See also: `MouseDown`, `firstClick`, `selectable`, `selected`, `focused`

**shape REGION**

Maintains the non-rectangular shape of the widget. In the set-mode REGION is either a Prima::Image object with its 0 bits treated as transparent pixels and 1 bits as opaque pixels, or a Prima::Region object. In the get-mode, it is either undef or a Prima::Region object.

Successive only if the `sv::ShapeExtension` value is true.

**showHint BOOLEAN**

If 1, the toolkit is allowed to show the hint label over the widget. The `::hint` property must contain a non-empty string text if the hint label is to be shown.

The default value is 1.

See also: `hint`, `ownerShowHint`, `hintVisible`, `ownerHint`

**size WIDTH HEIGHT**

Maintains the width and height of the widget.

See also: `width`, `height` `growMode`, `Move`, `Size`, `get_virtual_size`, `sizeMax`, `sizeMin`

**sizeMax WIDTH HEIGHT**

Specifies the maximal size for the widget.

See also: `width`, `height`, `size` `growMode`, `Move`, `Size`, `get_virtual_size`, `sizeMin`

**sizeMin WIDTH HEIGHT**

Specifies the minimal size for the widget.

See also: `width`, `height`, `size` `growMode`, `Move`, `Size`, `get_virtual_size`, `sizeMax`

**skin SCALAR**

A generic scalar, is not used in the `Prima::Widget` class implementation but is designed to select the visual style of a widget, where the interpretation of the property value will be up to the widget class itself. Many of the toolkit widgets implement two skins, `classic` and `flat`.

Does not repaint the widget on the property change, however many of the toolkit widgets do that.

If the `ownerSkin` property value is 1 returns the skin of the owner. When is undef, sets the `ownerSkin` property to 1, otherwise resets it to 0.

Note: this is not a symmetric property, as a `$self->skin($self->skin)` call is not idempotent.

**syncPaint BOOLEAN**

If 0, the `Paint` request notifications are stacked until the event loop is called. If 1, every time the widget surface gets invalidated the `Paint` notification is called.

Default: 0

See also: `invalidate_rect`, `repaint`, `validate_rect`, `Paint`

**tabOrder INTEGER**

Maintains the order in which the tab- and shift-tab key navigation algorithms focus the sibling widgets. INTEGER is unique among the sibling widgets. In the set-mode, if INTEGER the value is already taken by another widget, the latter is assigned another unique value, but without the destruction of the internal queue - the widgets with ::tabOrder greater than of the widget in question receive new values too. The special value -1 is accepted as 'the end of list' request in the set-call. A negative value is never returned in the get-call.

See also: `tabStop`, `next_tab`, `selectable`, `selected`, `focused`

**tabStop BOOLEAN**

Specifies whether the widget is interested in the tab- and shift-tab key navigation or not.

Default value is 1.

See also: `tabOrder`, `next_tab`, `selectable`, `selected`, `focused`

**text TEXT**

A text string for generic purposes. Many Prima::Widget descendants use this property heavily - buttons, labels, input lines, etc, but Prima::Widget itself does not.

If the `TEXT` is a reference to a string, it is treated as a markup string and is compiled into a `Prima::Drawable::Markup` object internally.

See the *Prima::Drawable::Markup* section, *examples/markup.pl*

**top INTEGER**

Maintains the upper boundary of the widget. If changed does not affect the widget height, however does so if called in the `set()` method together with the `::bottom` property.

See also: `left`, `right`, `bottom`, `origin`, `rect`, `growMode`, `Move`

**transparent BOOLEAN**

Specifies whether the background of the widget before it starts painting is of any importance. If 1, the widget can gain a certain emulated transparency look if it does not clear the background during the `Paint` event.

Default value is 0

See also: `Paint`, `buffered`.

**visible BOOLEAN**

Specifies whether the widget is visible or not. See the *Visibility* entry.

See also: `Show`, `Hide`, `showing`, `exposed`

**widgetClass CLASS**

Maintains the integer value, designating the color class that is defined by the system and is associated with Prima::Widget's eight basic color properties. The CLASS can be one of the `wc::XXX` constants:

```
wc::Undef
wc::Button
wc::CheckBox
wc::Combo
wc::Dialog
wc::Edit
wc::InputLine
wc::Label
wc::ListBox
wc::Menu
wc::Popup
wc::Radio
wc::ScrollBar
wc::Slider
wc::Widget or wc::Custom
wc::Window
wc::Application
```

These constants are not associated with the toolkit classes but rather are a wide shot to any possible native classes or widgets that the system may implement and have different color defaults for. Any Prima class can use any of these constants in its `::widgetClass` property.

See also: `map_color`, `colorIndex`

**widgets @WIDGETS**

In the get-mode returns the list of immediate children widgets (identical to `get_widgets`). In the set-mode accepts the set of widget profiles, as `insert` does, as a list or an array. This way it is possible to create a widget hierarchy in a single call.

**width WIDTH**

Maintains the width of the widget.

See also: `height growMode`, `Move`, `Size`, `get_virtual_size`, `sizeMax`, `sizeMin`

**x_centered BOOLEAN**

A write-only property. Once set, the widget is centered on the horizontal axis relative to its owner.

See also: `centered`, `y_centered`, `growMode`, `origin`, `Move`.

**y_centered BOOLEAN**

A write-only property. Once set, the widget is centered on the vertical axis relative to its owner.

See also: `x_centered`, `centered`, `growMode`, `origin`, `Move`.

### Methods

**begin_drag [ DATA | %OPTIONS ]**

Wrapper over the `dnd_start` method that adds several aspects to the DND session that the system doesn't offer. All of the input is contained in the `%OPTIONS` hash except when the case of a single-parameter call, when the DATA scalar is treated either as `text => DATA` or `image => DATA` depending on the `DATA` type.

Returns -1 if a DND session cannot start, `dnd::None` if it was canceled by the user, or any other `dnd::` constant when the DND receiver has selected and successfully performed that action. For example, after a call to the `dnd_start` method that returned the `dnd::Move` value the caller may remove the data the user selected to move (`Prima::InputLine` and `Prima::Edit` do exactly this).

In the `wantarray` context also returns the widget that accepted the drop, if that is a Prima widget. Check this before handling `dnd::Move` actions that require data to be deleted on the source, to not occasionally delete the freshly transferred data. The `begin_drag` method uses a special precaution for this scenario and by default won't let the widget be both the sender and the receiver ( see `self_aware` below ).

The following input is recognized:

**actions INTEGER = dnd::Copy**

A combination of the `dnd::` constants, to tell a DND receiver if copying, moving, and/or linking the data is allowed. The method fails on the invalid `actions` input.

**format Format, data INPUT**

If set, the DND clipboard will contain a single entry of the `INPUT` in the `Format` format, where the format is either the standard `Text` or `Image`, or one of the formats registered by the `Clipboard::register_format` method.

If not set, the caller needs to fill the clipboard in advance, f.ex. to offer data in more than one format.

**image INPUT**

Shortcut for `format => 'Image', data => $INPUT, preview => $INPUT`

**preview INPUT**

If set, the mouse pointers sending feedback to the user will be visually combined with either text or image, depending on whether `INPUT` is a text scalar or an image reference.

**self_aware BOOLEAN = 1**

If unset, the widget's `dndAware` will be temporarily set to 0 to exclude a possibility of an operation that may end in sending data to itself.

**text INPUT**

Shortcut for `format => 'Text', data => $INPUT, preview => $INPUT`

**track INTEGER = 5**

If set, waits to start the DND process until the user moves the mouse pointer away from the starting point further than `track` pixels, which makes sense if the method is to be called directly from a `MouseDown` event handler.

If the drag did not happen because the user released the button or otherwise marked that this is not a drag, -1 is returned. In that case, the caller should continue to handle the `MouseDown` event as if no drag session was ever started.

**bring_to_front**

Sends the widget on top of all other sibling widgets

See also: `insert_behind`, `send_to_back`, `ZOrderChanged` ,`first`, `next`, `prev`, `last`

**can_close**

Sends the `Close` event and returns its flag value. Windows that need to abort a potential closing, for example when an editor asks the user if a document needs to be saved, need to call the `clear_event` method in the `Close` event handler.

See also: `Close`, `close`

**client_to_screen @OFFSETS**

Maps an array of X and Y integer offsets from the widget to the screen coordinates. Returns the mapped OFFSETS.

See also: `screen_to_client`, `clipOwner`

**close**

Calls `can_close()`, and if successful, destroys the widget. Returns the `can_close()` result.

See also: `can_close`, `Close`

**defocus**

Alias for the `focused(0)` call

See also: `focus`, `focused`, `Enter`, `Leave`

**deselect**

Alias for the `selected(0)` call

See also: `select`, `selected`, `Enter`, `Leave`

**dnd_start ACTIONS = dnd::Copy, USE_DEFAULT_POINTERS = 1**

Starts a drag-and-drop session with a combination of the `ACTIONS` allowed. It is expected that the DND clipboard will be filled with the data that are prepared to be sent to a DND receiver.

Returns -1 if a DND session cannot start, the `dnd::None` constant if it was canceled by the user or any other `dnd::` constant when the DND receiver has selected and successfully performed that action. For example, after a call to the `dnd_start` method returning `dnd::Move`, the caller may remove the data the user selected to move (`Prima::InputLine` and `Prima::Edit` do exactly this).

Also returns the widget that accepted the drop, if that was the Prima widget within the same program.

If the USE_DEFAULT_POINTERS flag is set the system will use default drag pointers. Otherwise, it is expected that a `DragResponse` action will change the mouse pointers according to the current action, to give the user the visual feedback.

See `begin_drag` for the wrapper over this method that extends this functionality.

See also: `Drag and Drop`, `DragQuery`, `DragResponse`.

**exposed**

Returns the boolean value indicating whether the widget is at least partly visible on the screen. Never returns 1 if the widget's `::visible` value is 0.

See also: `visible`, `showing`, `Show`, `Hide`

**fetch_resource CLASS_NAME, NAME, CLASS_RESOURCE, RESOURCE, OWNER, RESOURCE_TYPE = fr::String**

Returns a system-defined scalar of the resource, defined by the widget hierarchy, its class, name, and owner. RESOURCE_TYPE can be one of the following type constants:

```
fr::Color  - color resource
fr::Font   - font resource
fs::String - text string resource
```

These parameters are used to address the widget in its hierarchy before it is created. The CLASS_NAME is the widget class string, NAME is the widget name. CLASS_RESOURCE is the class of the resource, and RESOURCE is the resource name.

For example, resources 'color' and 'disabledColor' belong to the resource class 'Foreground'.

**first**

Returns the first ( from bottom ) sibling widget in Z-order.

See also: `last`, `next`, `prev`

**focus**

Alias for `focused(1)` call

See also: `defocus`, `focused`, `Enter`, `Leave`

**hide**

Sets widget `::visible` to 0.

See also: `hide`, `visible`, `Show`, `Hide`, `showing`, `exposed`

**hide_cursor**

Hides the cursor. If the `hide_cursor()` method was called more than once then the `show_cursor` should also be called as many times to show the cursor back.

See also: `show_cursor`, `cursorVisible`

**help**

Starts the interactive help viewer session and requests it to open the link in the `::helpContext` string value. The string value is combined from the widget's owner `::helpContext` strings if the latter is empty or begins with a slash. A special meaning is assigned to the empty string " " - the help() call fails when such value is found to be the section component. This feature can be useful when a window or a dialog presents a standalone functionality in a separate module, and the documentation is related more to the module than to an embedding program. In such case the grouping widget holds `::helpContext` as a pod manpage name with the trailing slash, and its children widgets are assigned `::helpContext` to the topics without the manpage but with the leading slash instead. If the grouping widget has an empty string " " as the `::helpContext` then the help is unavailable for all the children widgets.

See also: `helpContext`

**insert CLASS, %PROFILE [[ CLASS, %PROFILE], ... ]**

Creates one or more widgets with their `owner` properties set to the caller widget, and returns the list of the references to the newly created widgets.

Has two calling formats:

**Single widget**

```
$parent-> insert( 'Child::Class',
   name => 'child',
   ....
);
```

**Multiple widgets**

```
$parent-> insert(
  [
     'Child::Class1',
        name => 'child1',
        ....
  ],
  [
     'Child::Class2',
        name => 'child2',
        ....
  ],
);
```

**insert_behind OBJECT**

Sends the widget behind the OBJECT on Z-axis given that the OBJECT is a sibling to the widget.

See also: `bring_to_front`, `send_to_back`, `ZOrderChanged` ,`first`, `next`, `prev`, `last`

**invalidate_rect X_LEFT_OFFSET Y_BOTTOM_OFFSET X_RIGHT_OFFSET Y_TOP_OFFSET**

Marks the rectangular area of the widget as 'invalid', triggering the re-painting of the area. See the *Graphic content* entry.

See also: `validate_rect`, `get_invalid_rect`, `repaint`, `Paint`, `syncPaint`, `update_view`

**is_surface_buffered**

Returns true if the `buffered` property is set and the buffering request was granted. The value is only valid inside the `begin_paint/end_paint` bracket and is always false otherwise.

See also: the *buffered* entry

**is_surface_layered**

Returns true if both the widget and its top-most parent are layered. If the widget itself is top-most, i.e. a window, a non-clipOwner widget, or a child to the application, then is the same as `layered`.

See also: the *layered* entry

**key_down CODE, KEY = kb::NoKey, MOD = 0, REPEAT = 1, POST = 0**

The method sends or posts ( POST flag ) the simulated `KeyDown` event to the system. CODE, KEY, MOD, and REPEAT are the parameters to be passed to the notification callbacks.

See also: `key_up`, `key_event`, `KeyDown`

**key_event COMMAND, CODE, KEY = kb::NoKey, MOD = 0, REPEAT = 1, POST = 0**

The method sends or posts ( POST flag ) the simulated keyboard event to the system. CODE, KEY, MOD and REPEAT are the parameters to be passed to an eventual `KeyDown` or `KeyUp` notification. COMMAND is allowed to be either `cm::KeyDown` or `cm::KeyUp`.

See also: `key_down`, `key_up`, `KeyDown`, `KeyUp`

**key_up CODE, KEY = kb::NoKey, MOD = 0, POST = 0**

The method sends or posts ( POST flag ) the simulated `KeyUp` event to the system. CODE, KEY and MOD are the parameters to be passed to the notification callbacks.

See also: `key_down`, `key_event`, `KeyUp`

**last**

Returns the last ( the topmost ) sibling widget in Z-order.

See also: `first`, `next`, `prev`

**lock**

Turns off the ability of the widget to re-paint itself. If the `lock` method was called more than once, then the `unlock` method should be called as many times to re-enable the painting. Returns the boolean success flag.

See also: `unlock`, `repaint`, `Paint`, `get_locked`

**map_color COLOR**

Translated combinations of the `cl::XXX` and `ci::XXX` constants to a 24-bit RGB integer color value. If the COLOR is already in the RGB format, returns the same value.

See also: `colorIndex`

**mouse_click BUTTON = mb::Left, MOD = 0, X = 0, Y = 0, NTH = 0, POST = 0**

The method sends or posts ( POST flag ) the simulated `MouseClick` event to the system. BUTTON, MOD, X, Y, and NTH are the parameters to be passed to the notification callbacks.

See also: `MouseDown`, `MouseUp`, `MouseWheel`, `MouseMove`, `MouseEnter`, `MouseLeave`

**mouse_down BUTTON = mb::Left, MOD = 0, X = 0, Y = 0, POST = 0**

The method sends or posts ( POST flag ) the simulated `MouseDown` event to the system. BUTTON, MOD, X, and Y are the parameters to be passed to the notification callbacks.

See also: `MouseUp`, `MouseWheel`, `MouseClick`, `MouseMove`, `MouseEnter`, `MouseLeave`

**mouse_enter MOD = 0, X = 0, Y = 0, POST = 0**

The method sends or posts ( POST flag ) the simulated `MouseEnter` event to the system. MOD, X, and Y are the parameters to be passed to the notification callbacks.

See also: `MouseDown`, `MouseUp`, `MouseWheel`, `MouseClick`, `MouseMove`, `MouseLeave`

**mouse_event COMMAND = cm::MouseDown, BUTTON = mb::Left, MOD = 0, X = 0, Y = 0, NTH = 0, POST = 0**

The method sends or posts ( POST flag ) the simulated mouse event to the system. BUTTON, MOD, X, Y, and NTH are the parameters to be passed to an eventual mouse notification. COMMAND is allowed to be one of the `cm::MouseDown`, `cm::MouseUp`, `cm::MouseWheel`, `cm::MouseClick`, `cm::MouseMove`, `cm::MouseEnter`, `cm::MouseLeave` constants.

See also: `mouse_down`, `mouse_up`, `mouse_wheel`, `mouse_click`, `mouse_move`, `mouse_enter`, `mouse_leave`, `MouseDown`, `MouseUp`, `MouseWheel`, `MouseClick`, `MouseMove`, `MouseEnter`, `MouseLeave`

**mouse_leave**

The method sends or posts ( POST flag ) the simulated `MouseLeave` event to the system.

See also: `MouseDown`, `MouseUp`, `MouseWheel`, `MouseClick`, `MouseMove`, `MouseEnter`, `MouseLeave`

**mouse_move MOD = 0, X = 0, Y = 0, POST = 0**

The method sends or posts ( POST flag ) the simulated `MouseMove` event to the system. MOD, X, and Y are the parameters to be passed to the notification callbacks.

See also: `MouseDown`, `MouseUp`, `MouseWheel`, `MouseClick`, `MouseEnter`, `MouseLeave`

**mouse_up BUTTON = mb::Left, MOD = 0, X = 0, Y = 0, POST = 0**

The method sends or posts ( POST flag ) the simulated `MouseUp` event to the system. BUTTON, MOD, X, and Y are the parameters to be passed to the notification callbacks.

See also: `MouseDown`, `MouseWheel`, `MouseClick`, `MouseMove`, `MouseEnter`, `MouseLeave`

**mouse_wheel MOD = 0, X = 0, Y = 0, INCR = 0, POST = 0**

The method sends or posts ( POST flag ) the simulated `MouseUp` event to the system. MOD, X, Y, and INCR are the parameters to be passed to the notification callbacks.

See also: `MouseDown`, `MouseUp`, `MouseClick`, `MouseMove`, `MouseEnter`, `MouseLeave`

**next**

Returns the neighbor sibling widget, next ( above ) in the Z-order. If none is found, undef is returned.

See also: `first`, `last`, `prev`

**next_tab FORWARD = 1**

Returns the next widget in the list of the sibling widgets sorted by `::tabOrder`. FORWARD is the boolean lookup direction flag. If none is found, the first ( or the last, depending on the FORWARD flag ) widget is returned. Only widgets with the `::tabStop` value set to 1 participate in the scanning.

Also used by the internal keyboard navigation code.

See also: `next_positional`, `tabOrder`, `tabStop`, `selectable`

**next_positional DELTA_X DELTA_Y**

Returns the sibling, the (grand-)child of a sibling, or the (grand-)child widget that matched best the direction specified by the DELTA_X and DELTA_Y integers. Only one of these parameters may be zero; another parameter must be either 1 or -1.

Also used by the internal keyboard navigation code.

See also: `next_tab`, `origin`

**pack, packForget, packSlaves**

See the *Prima::Widget::pack* section

**place, placeForget, placeSlaves**

See the *Prima::Widget::place* section

**prev**

Returns the neighbor sibling widget, previous ( below ) in the Z-order. If none is found, undef is returned.

See also: `first`, `last`, `next`

**repaint**

Marks the whole widget area as 'invalid', triggering the re-painting of the widget. See the *Graphic content* entry.

See also: `validate_rect`, `get_invalid_rect`, `invalidate_rect`, `Paint`, `update_view`, `syncPaint`

**rect_bevel $CANVAS, @RECT, %OPTIONS**

Draws a rectangular area, similar to one produced by the `rect3d` method, over `@RECT` which is a 4-integer tuple (X1,Y1,X2,Y2). Uses the values of the widget's `light3DColor` and `dark3DColor` properties. The following options are recognized:

**fill COLOR**

If set, the area is filled with COLOR, otherwise is left intact.

**width INTEGER**

The width of the border in pixels

**concave BOOLEAN**

If 1, draws a concave area, or a bulged area otherwise

**responsive**

Returns the boolean flag indicating whether the widget and its owners have all the `::enabled` property value set 1 or not. Useful for the fast check if the widget should respond to the user's actions.

See also: `enabled`

**screen_to_client @OFFSETS**

Maps array of X and Y integer offsets from screen to widget coordinates. Returns the mapped OFFSETS.

See also: `client_to_screen`

**scroll DELTA_X DELTA_Y %OPTIONS**

Scrolls the graphic context area by DELTA_X and DELTA_Y pixels. The OPTIONS is a hash that may contain the following optional parameters:

**clipRect [X1, Y1, X2, Y2]**

The clipping area is confined by the X1, Y1, X2, Y2 rectangle. If not specified, the clipping area covers the whole widget. Only the bits covered by the clipRect are affected. The bits scrolled from the outside of the rectangle to the inside are invalidated; the bits scrolled from the inside of the rectangle to the outside are not invalidated.

**confineRect [X1, Y1, X2, Y2]**

The scrolling area is confined by the X1, Y1, X2, Y2 rectangle. If not specified, the scrolling area covers the whole widget.

**withChildren BOOLEAN**

If 1, the scrolling affects the eventual children widgets so that they also change their positions to DELTA_X and DELTA_Y.

Returns one of the following constants:

```
scr::Error          - failure
scr::NoExpose       - call resulted in no new exposed areas
scr::Expose         - call resulted in new exposed areas, expect a repaint
```

Cannot be used inside the paint state.

See also: `Paint`, `get_invalid_rect`

**select**

Alias for `selected(1)` call

See also: `deselect`, `selected`, `Enter`, `Leave`

**send_to_back**

Sends the widget to the bottom of all other sibling widgets

See also: `insert_behind`, `bring_to_front`, `ZOrderChanged` ,`first`, `next`, `prev`, `last`

**show**

Sets the widget's `::visible` property to 1.

See also: `hide`, `visible`, `Show`, `Hide`, `showing`, `exposed`

**show_cursor**

Shows the cursor. If the `hide_cursor()` method was called more than once then the `show_cursor` should also be called as many times to show the cursor back.

See also: `hide_cursor`, `cursorVisible`

**showing**

Returns the boolean value indicating whether the widget and its owners have all `::visible` property set to 1 or not.

**unlock**

Turns on the ability of a widget to re-paint itself. As many times the `lock()` method was called, as many times its counterpart, the `unlock()` method must be called to enable re-painting again. After the last `unlock()` is called an implicit `repaint()` call is issued. Returns the boolean success flag.

See also: `lock`, `repaint`, `Paint`, `get_locked`

**update_view**

If any parts of the widget were marked as 'invalid' by either the `invalidate_rect()`, `scroll`, or `repaint()` calls, or by the exposure caused by the window movements, then the `Paint` notification is immediately called. If no parts are invalid, no action is performed. If the widget has the `::syncPaint` property set to 1 the `update_view()` is always a no-op call.

See also: `invalidate_rect`, `get_invalid_rect`, `repaint`, `Paint`, `syncPaint`, `update_view`

**validate_rect X_LEFT_OFFSET Y_BOTTOM_OFFSET X_RIGHT_OFFSET Y_TOP_OFFSET**

Reverses the effect of `invalidate_rect()`, restoring the original, 'valid' state of the widget area covered by the rectangular area passed. If the widget with previously invalid areas was wholly validated by this method, no `Paint` notifications occur.

See also: `invalidate_rect`, `get_invalid_rect`, `repaint`, `Paint`, `syncPaint`, `update_view`

**Get-methods**

**get_default_font**

Returns the default font for the Prima::Widget class.

See also: `font`

**get_default_popup_font**

Returns the default font for the Prima::Popup class.

See also: `font`

**get_invalid_rect**

Returns the rectangle encompassing the actual invalid region on the widget. If the widget doesn't need to be repainted, the (0,0,0,0) tuple is returned.

See also: `validate_rect`, `invalidate_rect`, `repaint`, `Paint`, `syncPaint`, `update_view`

**get_handle**

Returns the system handle for the widget

See also: `get_parent_handle`, `Window::get_client_handle`

**get_locked**

Returns 1 if the `lock()` was called and all repaints are effectively blocked.

See also: `lock`, `unlock`

**get_mouse_state**

Returns a combination of the `mb::XXX` constants that reflects the currently pressed mouse buttons.

See also: `pointerPos`, `get_shift_state`

**get_parent**

Returns the widget that the caller widget boundaries get clipped to, or the application object if the caller widget is top-level or has the clipOwner property set to 1.

See also: `clipOwner`

**get_parent_handle**

Returns the system handle for the parent of the widget, the window that belongs to another program. Returns 0 if the widget's owner and parent are in the same application and process space.

See also: `get_handle`, `clipOwner`

**get_pointer_size**

Returns two integers, the width and height of the icon, that the system accepts as valid for the mouse pointer. If the sizes of the icon exceed or are inferior to the size the icon is then truncated or padded with transparency bits ( but not stretched ). Can be called with the class syntax as it returns the system-wide value.

**get_shift_state**

Returns a combination of the `km::XXX` constants that reflects the currently pressed keyboard modifier buttons.

See also: `get_shift_state`

**get_virtual_size**

Returns the virtual width and height of the widget. See the *Geometry* entry, Implicit size regulations.

See also: `width`, `height`, `size` `growMode`, `Move`, `Size`, `sizeMax`, `sizeMin`

**get_widgets**

Returns the list of the children widgets.

### Events

**Change**

A generic notification, is used for the Prima::Widget's descendants; the Prima::Widget class itself neither calls nor uses the event. Designed to be called when an arbitrary major state of the widget is changed.

**Click**

A generic notification, is used for the Prima::Widget's descendants; Prima::Widget itself neither calls nor uses the event. Designed to be called when an arbitrary major action for the widget is called.

**Close**

Triggered by the `can_close()` and `close()` functions. If the event flag is cleared during execution, these functions return the false value.

See also: `close`, `can_close`

**ColorChanged INDEX**

Called when one of the widget's color properties is changed, either by a direct property change or by the system. INDEX is one of the `ci::XXX` constants.

See also: `colorIndex`

**Disable**

Triggered by a successful `enabled(0)` call

See also: `Enable`, `enabled`, `responsive`

**DragBegin CLIPBOARD, ACTION, MOD, X, Y, COUNTERPART**

Triggered on the receiver widget when the mouse pointer with a DND object enters its screen boundaries. `CLIPBOARD` contains the DND data, `ACTION` is a combination of the `dnd::` constants that reflect the actions the sender is ready to offer, `MOD` is a combination of the modifier keys (`kb::`), and `X` and `Y` are the coordinates where the mouse pointer has entered the widget. This event and the following `DragOver` and `DragEnd` events occur only if the property `dndAware` is set either to 1 or if it matches the clipboard format that exists in the `CLIPBOARD`.

`COUNTERPART` is set to the Prima DND sender widget if the session was initiated within the same program; is undef otherwise.

See also: the *Drag and Drop* entry, `DragOver`, `DragEnd`

**DragEnd CLIPBOARD, ACTION, MOD, X, Y, COUNTERPART, ANSWER**

Triggered on the receiver widget when the user either drops or cancels the DND session. In case of the canceled drop, `CLIPBOARD` is set to `undef` and `ACTION` to the `dnd::None` constant. On a successful drop, the input data are the same as in `DragBegin` while the output data are expected to be stored in the hashref `ANSWER`, if any. The following answers can be stored:

**allow BOOLEAN**

Is pre-set to 1. If changed to 0, a signal will be sent to the sender that the drop request is not accepted.

**action INTEGER**

A `dnd::` constant (not a combination) to be returned to the sender with the action the receiver has accepted, if any.

`COUNTERPART` is set to the Prima DND sender widget if the session was initiated within the same program; is undef otherwise.

See also: the *Drag and Drop* entry, `DragBegin`, `DragOver`

**DragOver CLIPBOARD, ACTION, MOD, X, Y, COUNTERPART, ANSWER**

Triggered on the received widget during the DND session. The event is sent repeatedly while the user drags the mouse pointer over the widget. The input data are same as in `DragBegin`, and output data are to be stored in hashref `ANSWER`, if any. The following answers can be stored:

**allow BOOLEAN**

Is pre-set to 1. If the event handler changes it to 0, a response will be sent to the sender that a drop action cannot happen with the input or location provided.

**action INTEGER**

A `dnd::` constant (not a combination) to be returned to the sender with the action the receiver is ready to accept, if any.

**pad X, Y, WIDTH, HEIGHT**

If set, instructs the sender not to repeat `DragOver` events that contain the same input data, while the mouse pointer is within these geometrical limits.

`COUNTERPART` is the Prima DND sender widget, if the session is initiated within the same program.

**DragQuery MOD, COUNTERPART, ANSWER**

Triggered on a sender DND widget when there was detected a change in the mouse or modifier buttons, or the user pressed the `Escape` key to cancel the DND session. The combination of the mouse and modifier buttons is stored in the `MOD` integer parameter. The `km::Escape` bit is set if the `Escape` key is pressed.

It is up to the event handler to decide whether to continue the drag session or not. If it is decided not to continue, the `$ANSWER->{allow}` flag must be set to 0.

Additionally, the `$ANSWER->{action}` flag can be assigned a single `dnd::` constant to counter-propose the action to the sender. The proposal will be typically based on the `MOD` value, f.ex. `dnd::Move` if the CTRL key was pressed.

Note: This action will only forward the change to the receiver on X11, but it is advised to implement it anyway for the sake of portability.

`COUNTERPART` is the Prima DND receiver widget, if the session is initiated within the same program.

See also: the *Drag and Drop* entry, `DragResponse`

**DragResponse ALLOW, ACTION, COUNTERPART**

Triggered on the sender DND widget when there was detected a change in the mouse or modifier buttons, or when the mouse was moved from one DND target to another. The sender event handler is then presented with the new input, collected from the interaction with the new target. There, the `ALLOW` integer parameter is set to a boolean value that shows

whether the sender is allowed to drop data or not. The `ACTION` is the `dnd::` constant with the action the receiver has earlier agreed to accept, if any.

If the DND session was started without the option to update mouse pointers on this event, the event handler should update the pointer itself. It is not needed though to save and restore the mouse pointers before and after the DND session, the `begin_drag` method manages this.

`COUNTERPART` is the Prima DND receiver widget, if the session is initiated within the same program.

See also: the *Drag and Drop* entry, `dnd_start`, `begin_drag`.

**Enable**

Triggered by a successful `enabled(1)` call

See also: `Disable`, `enabled`, `responsive`

**Enter**

Called when the widget receives the input focus.

See also: `Leave`, `focused`, `selected`

**FontChanged**

Called when the widget font is changed either by the direct property change call or by the system.

See also: `font`, `ColorChanged`

**Hide**

Triggered by a successful `visible(0)` call

See also: `Show`, `visible`, `showing`, `exposed`

**Hint SHOW_FLAG**

Called when the hint label is about to show or hide, depending on the SHOW_FLAG parameter. The show or hide action is not executed if the event flag is cleared in the event handler.

See also: `showHint`, `ownerShowHint`, `hintVisible`, `ownerHint`

**KeyDown CODE, KEY, MOD, REPEAT**

Sent to the focused widget when the user presses a key. CODE contains an eventual character code, KEY is one of the `kb::XXX` constants, and MOD is a combination of the modifier keys pressed when the event occurred ( the `km::XXX` constants ). REPEAT is an integer with the number of how many times the key was pressed; usually, it is 1. ( see `::briefKeys` ).

The valid `km::` constants are:

```
km::Shift
km::Ctrl
km::Alt
km::KeyPad
km::DeadKey
km::Unicode
```

The valid `kb::` constants are grouped in several sets. Some codes are aliased, for example, `kb::PgDn` and `kb::PageDown` have the same value.

**Modifier keys**

```
kb::ShiftL   kb::ShiftR   kb::CtrlL      kb::CtrlR
kb::AltL     kb::AltR     kb::MetaL      kb::MetaR
kb::SuperL   kb::SuperR   kb::HyperL     kb::HyperR
kb::CapsLock kb::NumLock  kb::ScrollLock kb::ShiftLock
```

**Keys with character code defined**

```
kb::Backspace  kb::Tab    kb::Linefeed   kb::Enter
kb::Return     kb::Escape kb::Esc        kb::Space
```

**Function keys**

```
kb::F1 .. kb::F30
kb::L1 .. kb::L10
kb::R1 .. kb::R10
```

**Other**

```
kb::Clear    kb::Pause   kb::SysRq  kb::SysReq
kb::Delete   kb::Home    kb::Left   kb::Up
kb::Right    kb::Down    kb::PgUp   kb::Prior
kb::PageUp   kb::PgDn    kb::Next   kb::PageDown
kb::End      kb::Begin   kb::Select kb::Print
kb::PrintScr kb::Execute kb::Insert kb::Undo
kb::Redo     kb::Menu    kb::Find   kb::Cancel
kb::Help     kb::Break   kb::BackTab
```

See also: `KeyUp`, `briefKeys`, `key_down`, `help`, `popup`, `tabOrder`, `tabStop`, `accelTable`

### KeyUp CODE, KEY, MOD

Sent to the focused widget when the user releases a key. CODE contains an eventual character code, KEY is one of the `kb::XXX` constants, and MOD is a combination of the modifier keys pressed when the event occurred ( `km::XXX`).

See also: `KeyDown`, `key_up`

### Leave

Called when the input focus is removed from the widget

See also: `Enter`, `focused`, `selected`

### Menu MENU VAR_NAME

Called before the user-navigated menu ( pop-up or pull-down ) is about to show another level of submenu on the screen. MENU is a Prima::AbstractMenu descendant, that is also a direct child to the widget. VAR_NAME is the name of the menu item that is about to be shown.

Can be used for making dynamic changes in the menu structures, f.ex. enabling or disabling clipboard commands if there is data in the clipboard that can be pasted.

See also: `popupItems`

### MouseClick BUTTON, MOD, X, Y, NTH

Called when the mouse click ( a button is pressed, then released, within the system-defined interval of time ) occurs in the widget area. BUTTON is one of the `mb::XXX` constants, MOD is a combination of the `km::XXX` constants that reflects the pressed modifier keys during the event, and X and Y are the mouse pointer coordinates. NTH is an integer, set to 0 if it was a single click, and to 2 and up if it was a double (triple, etc etc) click.

`mb::XXX` constants are:

```
mb::b1 or mb::Left
mb::b2 or mb::Middle
mb::b3 or mb::Right
mb::b4
mb::b5
mb::b6
mb::b7
mb::b8
```

See also: `MouseDown`, `MouseUp`, `MouseWheel`, `MouseMove`, `MouseEnter`, `MouseLeave`

**MouseDown BUTTON, MOD, X, Y**

Occurs when the user presses a mouse button on the widget. BUTTON is one of the `mb::XXX` constants, MOD is a combination of the `km::XXX` constants that reflects the pressed modifier keys during the event, and X and Y are the mouse pointer coordinates.

See also: `MouseUp`, `MouseClick`, `MouseWheel`, `MouseMove`, `MouseEnter`, `MouseLeave`

**MouseEnter MOD, X, Y**

Occurs when the mouse pointer enters the area occupied by the widget. MOD is a combination of the `km::XXX` constants that reflects the pressed modifier keys during the event, and X and Y are the mouse pointer coordinates.

See also: `MouseDown`, `MouseUp`, `MouseClick`, `MouseWheel`, `MouseMove`, `MouseLeave`

**MouseLeave**

Occurs when the mouse pointer leaves the area occupied by the widget.

See also: `MouseDown`, `MouseUp`, `MouseClick`, `MouseWheel`, `MouseMove`, `MouseEnter`

**MouseMove MOD, X, Y**

Occurs when the mouse pointer moves over the widget. MOD is a combination of the `km::XXX` constants that reflects the pressed modifier keys during the event, and X and Y are the mouse pointer coordinates.

See also: `MouseDown`, `MouseUp`, `MouseClick`, `MouseWheel`, `MouseEnter`, `MouseLeave`

**MouseUp BUTTON, MOD, X, Y**

Occurs when the user depresses a mouse button on the widget. BUTTON is one of the `mb::XXX` constants, MOD is a combination of the `km::XXX` constants that reflects the pressed modifier keys during the event, X and Y are the mouse pointer coordinates.

See also: `MouseDown`, `MouseClick`, `MouseWheel`, `MouseMove`, `MouseEnter`, `MouseLeave`

**MouseWheel MOD, X, Y, INCR**

Occurs when the user rotates the mouse wheel on the widget. MOD is a combination of the `km::XXX` constants that reflects the pressed modifier keys during the event, INCR is the wheel movement, scaled by 120. +120 is a step upwards, and -120 is a step downwards. Many of the consumer mice report the wheel moves with a resolution of 120, the gamer mice may report a better resolution. An event handler should treat the scroll values as INCR/120 per unit, for whatever the unit of movement might be, for example as lines of text, slider ticks, etc.

The event handle may use different units if some MOD keys are pressed. For example, the `Prima::SpinEdit` class has two different `step` and `pageStep` properties, and it uses the value of the `pageStep` property when the CTRL key is pressed and the value of the `step` property otherwise (see the *Prima::Sliders* section ).

See also: `MouseDown`, `MouseUp`, `MouseClick`, `MouseMove`, `MouseEnter`, `MouseLeave`

**Move OLD_X, OLD_Y, NEW_X, NEW_Y**

Triggered when the widget changes its position relative to its parent, either by one of the Prima::Widget methods or by the user. OLD_X and OLD_Y are the old coordinates of the widget, NEW_X and NEW_Y are the new ones.

See also: `Size`, `origin`, `growMode`, `centered`, `clipOwner`

**Paint CANVAS**

Caused when the system calls for the refresh of the widget's graphic content. CANVAS is the widget itself, use it to draw on ( see the *Graphic content* entry ).

See also: `repaint`, `syncPaint`, `get_invalid_rect`, `scroll`, `colorIndex`, `font`

**Popup BY_MOUSE, X, Y**

Called by the system when the user presses the key or the mouse combination defined for the execution of a context the pop-up menu. By default executes the associated Prima::Popup object if it is present. If the event flag is cleared in the event handler then pop-up menu request is denied and the popup is not shown.

See also: `popup`

**Setup**

This message is posted right after the `Create` notification and is delivered to widgets from inside the event loop. Prima::Widget does not use it for anything.

**Show**

Triggered by a successful `visible(1)` call

See also: `Show`, `visible`, `showing`, `exposed`

**Size OLD_WIDTH, OLD_HEIGHT, NEW_WIDTH, NEW_HEIGHT**

Triggered when the widget changes its size, either by Prima::Widget methods or by the user. OLD_WIDTH and OLD_HEIGHT are the old sizes of the widget, and NEW_WIDTH and NEW_HEIGHT are the new ones.

See also: `Move`, `origin`, `size`, `growMode`, `sizeMax`, `sizeMin`, `rect`, `clipOwner`

**SysHandle**

Same as in `Component` except that the following `Widget` properties can also trigger it:

the *clipOwner* entry, the *syncPaint* entry, the *layered* entry, the *transparent* entry

Handling of this event is generally needed only if the program relies on the widget's system handle that is returned by the `get_handle` method.

**TranslateAccel CODE, KEY, MOD**

A distributed version of the `KeyDown` event. The event traverses all of the object tree that the widget that received the original `KeyDown` event belongs to. Once the event flag is cleared, the iteration stops.

Used by the widgets that need to react to the keyboard input even if not focused.

See also: `KeyDown`

**ZOrderChanged**

Triggered when the widget's stacking order ( Z-order ) is changed either by one of the Prima::Widget methods or by the user.

See also: `bring_to_front`, `insert_behind`, `send_to_back`

## 3.9 Prima::Widget::pack

Geometry manager that packs around edges of cavity

### Synopsis

```
$widget-> pack( args);

$widget-> packInfo( args);
$widget-> geometry( gt::Pack);
```

### Description

The **pack** method is used to communicate with the packer, a geometry manager that arranges the children of a owner by packing them in order around the edges of the owner.

In this port of **Tk::pack** it is normal to pack widgets one-at-a-time using the widget object to be packed to invoke a method call. This is a slight distortion of the original Tcl-Tk interface (which can handle lists of windows to one pack method call) but Tk reports that it has proven effective in practice.

The **pack** method can have any of several forms, depending on *Option*:

**pack %OPTIONS**

The options consist of pairs of arguments that specify how to manage the slave. See the *The packer algorithm* entry below for details on how the options are used by the packer. The following options are supported:

**after => *$other***

*$other* must be another window. Use its master as the master for the slave, and insert the slave just after *$other* in the packing order.

**anchor => *anchor***

*Anchor* must be a valid anchor position such as **n** or **sw**; it specifies where to position each slave in its parcel. Defaults to **center**.

**before => *$other***

*$other* must be another window. Use its master as the master for the slave, and insert the slave just before *$other* in the packing order.

**expand => *boolean***

Specifies whether the slave should be expanded to consume extra space in their master. *Boolean* may have any proper boolean value, such as **1** or **no**. Defaults to 0.

**fill => *style***

If a slave's parcel is larger than its requested dimensions, this option may be used to stretch the slave. *Style* must have one of the following values:

**none**

Give the slave its requested dimensions plus any internal padding requested with **-ipadx** or **-ipady**. This is the default.

**x**

Stretch the slave horizontally to fill the entire width of its parcel (except leave external padding as specified by **-padx**).

**y**

Stretch the slave vertically to fill the entire height of its parcel (except leave external padding as specified by **-pady**).

**both**

Stretch the slave both horizontally and vertically.

**in => *$master***

Insert the slave(s) at the end of the packing order for the master window given by *$master*. Currently, only the immediate owner can be accepted as master.

**ipad => *amount***

*Amount* specifies how much both horizontal and vertical internal padding to leave on each side of the slave(s). *Amount* must be a valid screen distance, such as **2** or **.5c**. It defaults to 0.

**ipadx => *amount***

*Amount* specifies how much horizontal internal padding to leave on each side of the slave(s). *Amount* must be a valid screen distance, such as **2** or **.5c**. It defaults to 0.

**ipady => *amount***

*Amount* specifies how much vertical internal padding to leave on each side of the slave(s). *Amount* defaults to 0.

**pad => *amount***

*Amount* specifies how much horizontal and vertical external padding to leave on each side of the slave(s). *Amount* defaults to 0.

**padx => *amount***

*Amount* specifies how much horizontal external padding to leave on each side of the slave(s). *Amount* defaults to 0.

**pady => *amount***

*Amount* specifies how much vertical external padding to leave on each side of the slave(s). *Amount* defaults to 0.

**side => *side***

Specifies which side of the master the slave(s) will be packed against. Must be **left**, **right**, **top**, or **bottom**. Defaults to **top**.

If no **in**, **after** or **before** option is specified then slave will be inserted at the end of the packing list for its owner unless it is already managed by the packer (in which case it will be left where it is). If one of these options is specified then slave will be inserted at the specified point. If the slave are already managed by the geometry manager then any unspecified options for them retain their previous values rather than receiving default values.

**packForget**

Removes *slave* from the packing order for its master and unmaps its window. The slave will no longer be managed by the packer.

**packInfo [ %OPTIONS ]**

In get-mode, returns a list whose elements are the current configuration state of the slave given by *$slave*. The first two elements of the list are "**in**=>*$master*" where *$master* is the slave's master.

In set-mode, sets all **pack** parameters, but does not set widget geometry property to `gt::Pack`.

**packPropagate BOOLEAN**

If *boolean* has a true boolean value then propagation is enabled for *$master*, (see the *Geometry propagation* entry below). If *boolean* has a false boolean value then propagation is disabled for *$master*. If *boolean* is omitted then the method returns **0** or **1** to indicate whether propagation is currently enabled for *$master*.

Propagation is enabled by default.

**packSlaves**

Returns a list of all of the slaves in the packing order for *$master*. The order of the slaves in the list is the same as their order in the packing order. If *$master* has no slaves then an empty list/string is returned in array/scalar context, respectively

## The packer algorithm

For each master the packer maintains an ordered list of slaves called the *packing list*. The **in**, **after**, and **before** configuration options are used to specify the master for each slave and the slave's position in the packing list. If none of these options is given for a slave then the slave is added to the end of the packing list for its owner.

The packer arranges the slaves for a master by scanning the packing list in order. At the time it processes each slave, a rectangular area within the master is still unallocated. This area is called the *cavity*; for the first slave it is the entire area of the master.

For each slave the packer carries out the following steps:

- The packer allocates a rectangular *parcel* for the slave along the side of the cavity given by the slave's **side** option. If the side is top or bottom then the width of the parcel is the width of the cavity and its height is the requested height of the slave plus the **ipady** and **pady** options. For the left or right side the height of the parcel is the height of the cavity and the width is the requested width of the slave plus the **ipadx** and **padx** options. The parcel may be enlarged further because of the **expand** option (see the *Expansion* entry below)

- The packer chooses the dimensions of the slave. The width will normally be the slave's requested width plus twice its **ipadx** option and the height will normally be the slave's requested height plus twice its **ipady** option. However, if the **fill** option is **x** or **both** then the width of the slave is expanded to fill the width of the parcel, minus twice the **padx** option. If the **fill** option is **y** or **both** then the height of the slave is expanded to fill the width of the parcel, minus twice the **pady** option.

- The packer positions the slave over its parcel. If the slave is smaller than the parcel then the **-anchor** option determines where in the parcel the slave will be placed. If **padx** or **pady** is non-zero, then the given amount of external padding will always be left between the slave and the edges of the parcel.

  Once a given slave has been packed, the area of its parcel is subtracted from the cavity, leaving a smaller rectangular cavity for the next slave. If a slave doesn't use all of its parcel, the unused space in the parcel will not be used by subsequent slaves. If the cavity should become too small to meet the needs of a slave then the slave will be given whatever space is left in the cavity. If the cavity shrinks to zero size, then all remaining slaves on the packing list will be unmapped from the screen until the master window becomes large enough to hold them again.

## Expansion

If a master window is so large that there will be extra space left over after all of its slaves have been packed, then the extra space is distributed uniformly among all of the slaves for which the **expand** option is set. Extra horizontal space is distributed among the expandable slaves whose **side** is **left** or **right**, and extra vertical space is distributed among the expandable slaves whose **side** is **top** or **bottom**.

## Geometry propagation

The packer normally computes how large a master must be to just exactly meet the needs of its slaves, and it sets the requested width and height of the master to these dimensions. This causes

geometry information to propagate up through a window hierarchy to a top-level window so that the entire sub-tree sizes itself to fit the needs of the leaf windows. However, the **packPropagate** method may be used to turn off propagation for one or more masters. If propagation is disabled then the packer will not set the requested width and height. This may be useful if, for example, you wish for a master window to have a fixed size that you specify.

## Restrictions on master windows

The master for each slave must not be a child of the slave, and must not be present in any other list of slaves that directly or indirectly refers to the slave.

## Packing order

If the master for a slave is not its owner then you must make sure that the slave is higher in the stacking order than the master. Otherwise the master will obscure the slave and it will appear as if the slave hasn't been packed correctly. The easiest way to make sure the slave is higher than the master is to create the master window first: the most recently created window will be highest in the stacking order. Or, you can use the **bring_to_front** and **send_to_back** methods to change the stacking order of either the master or the slave.

## 3.10 Prima::Widget::place

Geometry manager for fixed or rubber-sheet placement

### Synopsis

```
$widget->place(option=>value?, option=>value, ...)

$widget->placeForget;

$widget->placeInfo(option=>value?, option=>value, ...);
$widget->geometry( gt::Place);

$master->placeSlaves
```

### Description

The placer is a geometry manager from Tk. It provides simple fixed placement of windows, where you specify the exact size and location of one window, called the *slave*, within another window, called the *$master*. The placer also provides rubber-sheet placement, where you specify the size and location of the slave in terms of the dimensions of the master, so that the slave changes size and location in response to changes in the size of the master. Lastly, the placer allows you to mix these styles of placement so that, for example, the slave has a fixed width and height but is centered inside the master.

#### place %OPTIONS

The **place** method arranges for the placer to manage the geometry of *$slave*. The remaining arguments consist of one or more *option=>value* pairs that specify the way in which *$slave*'s geometry is managed. If the placer is already managing *$slave*, then the *option=>value* pairs modify the configuration for *$slave*. The **place** method returns an empty string as result. The following *option=>value* pairs are supported:

**in => *$master***

*$master* is the reference to the window relative to which *$slave* is to be placed. *$master* must neither be *$slave*'s child nor be present in a slaves list that directly or indirectly refers to the *$slave*.

If this option isn't specified then the master defaults to *$slave*'s owner.

**x => *location***

*Location* specifies the x-coordinate within the master window of the anchor point for *$slave* widget.

**relx => *location***

*Location* specifies the x-coordinate within the master window of the anchor point for *$slave* widget. In this case the location is specified in a relative fashion as a floating-point number: 0.0 corresponds to the left edge of the master and 1.0 corresponds to the right edge of the master. *Location* need not be in the range 0.0-1.0. If both **x** and **relx** are specified for a slave then their values are summed. For example, "**relx**=>0.5, **x**=-2" positions the left edge of the slave 2 pixels to the left of the center of its master.

**y => *location***

*Location* specifies the y-coordinate within the master window of the anchor point for *$slave* widget.

**rely => *location***

*Location* specifies the y-coordinate within the master window of the anchor point for *$slave* widget. In this case the value is specified in a relative fashion as a floating-point number: 0.0 corresponds to the top edge of the master and 1.0 corresponds to the bottom edge of the master. *Location* need not be in the range 0.0-1.0. If both **y** and **rely** are specified for a slave then their values are summed. For example, **rely**=>0.5, **x**=>3 positions the top edge of the slave 3 pixels below the center of its master.

**anchor => *where***

*Where* specifies which point of *$slave* is to be positioned at the (x,y) location selected by the **x**, **y**, **relx**, and **rely** options. Thus if *where* is **se** then the lower-right corner of *$slave*'s border will appear at the given (x,y) location in the master. The anchor position defaults to **nw**.

**width => *size***

*Size* specifies the width for *$slave*. If *size* is an empty string, or if no **width** or **relwidth** option is specified, then the width requested internally by the window will be used.

**relwidth => *size***

*Size* specifies the width for *$slave*. In this case the width is specified as a floating-point number relative to the width of the master: 0.5 means *$slave* will be half as wide as the master, 1.0 means *$slave* will have the same width as the master, and so on. If both **width** and **relwidth** are specified for a slave, their values are summed. For example, **relwidth**=>1.0, **width**=>5 makes the slave 5 pixels wider than the master.

**height => *size***

*Size* specifies the height for *$slave*. If *size* is an empty string, or if no **height** or **relheight** option is specified, then the height requested internally by the window will be used.

**relheight => *size***

*Size* specifies the height for *$slave*. In this case the height is specified as a floating-point number relative to the height of the master: 0.5 means *$slave* will be half as high as the master, 1.0 means *$slave* will have the same height as the master, and so on. If both **height** and **relheight** are specified for a slave, their values are summed. For example, **relheight**=>1.0, **height**=>-2 makes the slave 2 pixels shorter than the master.

**placeSlaves**

The **placeSlaves** method returns a list of all the slave windows for which *$master* is the master. If there are no slaves for *$master* then an empty list is returned.

**placeForget**

The **placeForget** method causes the placer to stop managing the geometry of *$slave*. If *$slave* isn't currently managed by the placer then the method call has no effect.

**placeInfo %OPTIONS**

In get-mode the **placeInfo** method returns a list giving the current configuration of *$slave*. The list consists of *option*=>*value* pairs in exactly the same form as might be specified to the **place** method. If the configuration of a window has been retrieved with **placeInfo**, that configuration can be restored later by first using **placeInfo** in set-mode and setting **geometry** to `gt::Place`, which is equivalent to a direct call to **place**.

## Fine points

It is not necessary for the master window to be the owner of the slave window. This feature is useful in at least two situations. First, for complex window layouts it means you can create a

hierarchy of subwindows whose only purpose is to assist in the layout of the owner. The "*real children*" of the owner (i.e. the windows that are significant for the application's user interface) can be children of the owner yet be placed inside the windows of the geometry-management hierarchy. This means that the path names of the "*real children*" don't reflect the geometry-management hierarchy and users can specify options for the real children without being aware of the structure of the geometry-management hierarchy.

A second reason for having a master different than the slave's owner is to tie two siblings together. For example, the placer can be used to force a window always to be positioned centered just below one of its siblings by specifying the configuration

**in**=>*$sibling*, **relx**=>0.5, **rely**=>1.0, **anchor**=>'n'

Whenever the *$sibling* widget is repositioned in the future, the slave will be repositioned as well.

Unlike the other geometry managers (such as the packer) the placer does not make any attempt to manipulate the geometry of the master windows or the owners of slave windows (i.e. it doesn't set their requested sizes).

## 3.11 Prima::Window

Top-level window management

### Synopsis

```
use Prima;
use Prima::Application;

# this window, when closed, terminated the application
my $main = Prima::MainWindow-> new( text => 'Hello world' );

# this is a modal window
my $dialog = Prima::Dialog->create( size => [ 100, 100 ]);
my $result = $dialog-> execute;
$dialog-> destroy;

run Prima;
```

### Description

The Prima::Window class is a descendant of the Prima::Widget class. It represents the top-level windows that are treated specially by the system. The class's major difference from Prima::Widget is that instances of Prima::Window cannot reside inside of other windows and that the system or the window manager adds decorations to these - title bar, menus, and buttons. Prima::Window provides methods that communicate with the system and access these decorations.

### Usage

A typical program communicates with the user with the help of various widgets collected under one or more top-level windows. The creation of a Prima::Window object is straightforward:

```
my $w = Prima::Window-> new(
    size => [300,300],
    text => 'Startup window',
);
```

#### System window management

The top-level windows are special not only in their 'look', but also in 'feel': the system adds specific functions to the windows, aiding the user with the navigation through the desktop. The system often dictates the size and position of the newly created windows, and sometimes these rules are hard or even impossible to circumvent. This document would be quite long if it would venture off to describe the specificities of various window management systems, and it would never be complete - new window managers emerge every year, and the old ones unpredictable change their behavior. Therefore a word of advice: do not rely on the behavior of one window manager, test programs on at least two.

The Prima toolkit provides simple access to the buttons, title bar, and borders of the window. The buttons and title bar are managed by the `::borderIcons` property, and borders by the `::borderStyle` property. These properties operate with a set of the predefined constants `bi::XXX` and `bs::XXX`, correspondingly. The button constants can be combined bitwise, but not all combinations may be realized by the system. The same is valid also for the border constants, except that they cannot be combined; the value of the `::borderStyle` property contains a single `bs::XXX` constant.

There are other requests that the toolkit can ask from the window manager. The system can be supplied with the icon that the window is shown with together. The system icon dimensions can differ from system to system, and although they can be requested via the `sv::XIcon` and `sv::YIcon` system values, the `::icon` property scales the image automatically to the closest size the system can recognize. The window icon is not shown by the toolkit itself, it usually resides in the window decorations and sometimes on the taskbar, along with the window's name. The system can be requested to not add the window to the taskbar, by setting the `::taskListed` property to 0.

Another issue is the window positioning. Usually, if no explicit position is given, the window is positioned automatically by the system. The same is valid for the size. But some window managers bend that to the extreme - for example, the default CDE setup forces the user to set positions of the newly created windows explicitly. There is at least one point of certainty, however. Typically, when the initial size and/or position of the top-level window are expected to be set by the system, the `::originDontCare` and `::sizeDontCare` properties can be set to 1 during the window creation. If these are set, the system is requested to set the size and/or the position of the window according to its policy. The reverse is not always true, unfortunately. When either of these properties is set to 0, or the explicit size or position is given, the system is requested to use these values instead, but this does not always succeed from the program's point of view. Such behavior however is expected from the user's perspective and often does not even get noticed as something special. Therefore it is a good practice to test top-level windowing code on several window managers.

Different policies define that define window positioning and sizing. Some window managers behave best when the position is given to the window including the system-dependent decorations. This hardly can be called a good policy, since it is not possible to calculate the derived window coordinates with certainty. This leads to the fact that it can be impossible to know the exact size and position of the windows size before these are set explicitly. The only, not specially efficient help the toolkit can provide here, is the properties `::frameOrigin` and `::frameSize`, which along with the `::origin` and `::size` properties reflect the position and size of the window, but also taking into account the system-dependent decorations.

### Dialog execution

The `execute` method switches the window into the modal state. That means that the window is requested to reside on top of the other windows from the same program. The method returns after the window is dismissed in one or another way. It is special because it runs its own event loop, similar to the

```
run Prima;
```

code. The event flow is not disrupted, but the windows and widgets that do not belong to the currently executed, 'modal' window group can not be activated. There can be many modal windows on top of each other, but only one will be accessible for the user. A typical message box window, that prevents other message boxes from being operated, is an example of this scheme. This is also called the *exclusive* modality.

The toolkit also provides the *shared* modality scheme, where there can be several stacks of modal windows not interfering with each other. Each window stack contains its own windows. An analogy, consider the situation when several independent applications run with their own modal message boxes being executed; the windows under the message boxes still are not accessible to the user, but the user can switch between the applications. This scheme, however, can not be programmed with a single execute()-like call without creating interlocking conditions. The shared model call, the `execute_shared()` method, inserts the window into the shared modal stack, activates the window, and returns immediately.

Both kinds of modal windows can coexist in the same program, but the exclusive windows prevent the shared windows from being accessed by the user. While there are exclusive windows, the shared ones have the same rights as the normal windows.

The stacking order for these two models is also slightly different. The window after a call to the execute() method is sent to the top of the last exclusive modal window, or, in other words, is added to the exclusive window stack. There can be only one exclusive window stack, but many shared window stacks. The window after a call to the execute_shared() method is added to the shared window stack, to the one that the window's owner belongs to. The shared window stacks are located on so-called *modal horizons*, the windows with the boolean property `::modalHorizon` set to 1. The default modal horizon is `::application`.

The window in any modal state can return to the normal non-modal state by calling the `end_modal()` method. The window is then hidden and disabled, and the windows below it become accessible to the user. When the window's exclusive modal state is finished, its execute() method is finished as well; it returns the exit code, the same as the value of the `::modalResult` property. Two shortcut methods end the modal state and set the `::modalResult` property to the basic 'ok' or 'not ok' code, correspondingly by the `ok()` and `cancel()` methods. The behavior of the `cancel()` method is identical to when the user closes the modal window by clicking the system close button, pressing the Escape key, or otherwise canceling the dialog execution. The `ok()` method sets `::modalResult` to `mb::OK`, `cancel()` to `mb::Cancel`, correspondingly. There are more `mb::XXX` constants but these have no special meaning, any integer value can be passed. For example, the `Prima::MsgBox::message` method uses these constants so the message window can return up to four different `mb` codes.

### Menu

A top-level window can be equipped with a menu bar. Its outlook is system-dependent but can be controlled by the toolkit up to a certain level. The `::menuItems` property, which manages the menu items of a `::menu` object of the the *Prima::Menu* section class, arranges the layout of the menu. The syntax of the items-derived properties is described in the *Prima::Menu* section, but it must be reiterated that menu items contain only hints, not requests for their exact representation. The same is valid for the color and font properties, `::menuColorIndex` and `::menuFont`.

Only one menu at a time can be displayed in a top-level window, although a window can be an owner for many menu objects. The key property is `Prima::Menu::selected` - if a menu object is selected on a widget or a window object, it refers to the default menu actions, which, in the case of Prima::Window is being displayed as a menu bar.

Note: A window can be an owner for several menu objects and still not have a menu bar displayed, if no menu objects are marked as selected.

### Prima::Dialog

The Prima::Dialog class, a descendant from Prima::Window, introduces no new functionality. It only has its default values adjusted so that the colors it uses are matching the appropriate system dialog colors. It also requests the system that the look of the dialog window is to be different, to resemble the system dialogs on systems where such are provided.

### Prima::MainWindow

The class is a simple descendant of the Prima::Window class that overloads the `Destroy` notification and calls the `$application->close` inside it. The purpose of the declaration of a separate class for such a trifle difference is that many programs are designed under the paradigm where there exists the main window that is most important to the user. Since such a construct is used more often than any other, it is considered to be an optimization to write

```
Prima::MainWindow->new( ... )
```

rather than

```
Prima::Window->new( ...,
   mainWindow => 1,
   onDestroy  => sub { $::application-> close }
)
```

Additionally, the `$::main_window` scalar points to the newly created main window.
See also `mainWindow`.

## API

### Properties

#### borderIcons INTEGER

Requests the system to provide decorations for the window, by selecting a combination of the `bi::XXX` constants. These constants are:

```
bi::SystemMenu  - the system menu button and/or close button
                  ( usually with the icon )
bi::Minimize    - minimize button
bi::Maximize    - maximize/restore button
bi::TitleBar    - the window title
bi::All         - all of the above
```

Not all systems respect these requests, and some systems provide more decoration controls, but these are not addressable by the toolkit.

#### borderStyle STYLE

Requests the system to set the window border style, by selecting one of the `bs::XXX` constants. These constants are:

```
bs::None      - no border
bs::Single    - thin border
bs::Dialog    - thick border
bs::Sizeable  - border that can be resized
```

`bs::Sizeable` is a unique window mode. If selected, the user can resize the window, not only by dragging the window borders with the mouse but by other system-dependent means. The other border styles do not allow interactive resizing.

Not all systems recognize all of the requests, although all recognize the interactive resizing request.

#### effects HASH or undef

This generic property implements system-specific window effects, not necessarily portable. The format of the hash is also system-specific. The only portable behavior here is that setting the value to `undef` cancels all the effects.

Example:

```
$window->effects({
    effect1 => {
       key1 => $value1,
       ...
    },
});
```

Previously this mechanism was used for setting the DWM blur on Windows 7 and 8, but as Windows 10 removed it, this capability was also removed, so for now this is an empty call reserved for future use.

**frameHeight HEIGHT**

Maintains the height of the window, including the window decorations.

**frameOrigin X_OFFSET, Y_OFFSET**

Maintains the left X and bottom Y boundaries of the window's decorations relative to the screen.

**frameSize WIDTH, HEIGHT**

Maintains the width and height of the window, including the window decorations.

**frameWidth WIDTH**

Maintains the width of the window, including the window decorations.

**icon OBJECT**

Requests the system to associate the icon with the window. If OBJECT is set to `undef`, removes the association.

See also: `ownerIcon`

**mainWindow BOOLEAN**

Tells the system that the window is the main window for the application. The X11 implementation uses this field to associate dialogs with the main application window.

**menu OBJECT**

Manages the Prima::Menu object associated with the window. Prima::Window can host many Prima::Menu objects, but only the one that is registered in the `::menu` property is visualized as the menu bar.

See also: `Prima::Menu`, `menuItems`

**menuColorIndex INDEX, COLOR**

Manages eight color properties of a menu associated with the window. INDEX must be one of the `ci::XXX` constants ( see the *Prima::Widget* section, *colorIndex* section ).

See also: `menuItems`, `menuFont`, `menu`

**menuColor COLOR**

Basic foreground menu color.

See also: `menuItems`, `menuColorIndex`, `menuFont`, `menu`

**menuBackColor COLOR**

Basic background menu color.

See also: `menuItems`, `menuColorIndex`, `menuFont`, `menu`

**menuDark3DColor COLOR**

The color for drawing dark shades in menus.

See also: `menuItems`, `menuColorIndex`, `menuFont`, `menu`

**menuDisabledColor COLOR**

Foreground color for the disabled items in menus.

See also: `menuItems`, `menuColorIndex`, `menuFont`, `menu`

**menuDisabledBackColor COLOR**

Background color for the disabled items in menus.

See also: `menuItems`, `menuColorIndex`, `menuFont`, `menu`

**menuFont %FONT**

Manages the font of the menu

See also: `menuItems`, `menuColorIndex`, `menu`

**menuHiliteColor COLOR**

Foreground color for the selected items in menus.

See also: `menuItems`, `menuColorIndex`, `menuFont`, `menu`

**menuHiliteBackColor COLOR**

Background color for the selected items in menus.

See also: `menuItems`, `menuColorIndex`, `menuFont`, `menu`

**menuItems [ ITEM_LIST ]**

Manages items of the Prima::Menu object that is associated with the window. The ITEM_LIST format is the same as in the `Prima::AbstractMenu::items` property and is described in the *Prima::Menu* section.

See also: `menu`, `menuColorIndex`, `menuFont`

**menuLight3DColor COLOR**

Color for drawing light shades in menus.

See also: `menuItems`, `menuColorIndex`, `menuFont`, `menu`

**modalHorizon BOOLEAN**

Sets a flag that tells if the window serves as root to the shared modal window stack. A window with `::modalHorizon` set to 1 groups its children windows in a window stack, separate from other shared modal stacks. The `::modalHorizon` is therefore useful only when several shared modal window stacks are needed.

The property also serves as an additional grouping factor for widgets and windows. For example, default keyboard navigation by tab and arrow keys is limited to the windows and widgets of the same window stack.

**modalResult INTEGER**

Manages a custom integer value returned by the `execute()` method. Historically it is one of the `mb::XXX` constants, but any integer value can be used. The most useful `mb::` constants are:

```
mb::OK, mb::Ok
mb::Cancel
mb::Yes
mb::No
mb::Abort
mb::Retry
mb::Ignore
mb::Help
```

Note: These constants are defined so they can be or'ed bitwise, and the *Prima::MsgBox* package uses this feature in one of its parameters that can be a combination of the `mb::` constants.

**onTop BOOLEAN**

If set, the window is requested to stay on top of all other windows in the system.

Default value: 0

**ownerIcon BOOLEAN**

If 1, the icon is synchronized with the owner's. Automatically set to 0 if the `::icon` property is explicitly set. The default value is 1, so assigning an icon to $::application automatically assigns it to all windows.

**taskListed BOOLEAN**

If set to 0, requests that the system should not show the window in the system taskbar or the top-level window menu, if there is any.

If 1, does not request anything.

Default value: 1

**windowState STATE**

The property that manages the state of the window. STATE can be one of the four `ws::XXX` constants:

```
ws::Normal
ws::Minimized
ws::Maximized
ws::Fullscreen
```

There can be other window states provided by the system, but these four were chosen as a 'least common denominator'. The property can be changed either by an explicit set-mode call or by the user. In either case, a `WindowState` notification is triggered.

The property has the corresponding convenience wrappers: `maximize()`, `minimize()`, `restore()`, and `fullscreen()`.

See also: `WindowState`

### Methods

**cancel**

A standard method to dismiss the modal window with the `mb::Cancel` result. The effect of calling this method is equal to the action when the user closes the window with the system-provided menu, button, or some other command.

See also: `ok`, `modalResult`, `execute`, `execute_shared`

**end_modal**

Turns off the window modal state, sends the `EndModal` notification, and hides and disables the window. If the window is on top in the exclusive modal state, the last called `execute()` method finishes. If the window was not on top in the exclusive modal state, the corresponding `execute()` function finishes after all subsequent execute() calls are finished.

**execute INSERT_BEFORE = undef**

Switches the window to the exclusive modal state and puts it on top of all non-modal and shared-modal windows. By default, if INSERT_BEFORE object is undef, the window is also put on top of other exclusive-modal windows; if INSERT_BEFORE is one of the exclusive-modal windows the window is placed in the queue before the INSERT_BEFORE window. The window is made visible and enabled, if necessary, and the `Execute` notification is triggered.

The function is returned after the window is dismissed, or if the system-dependent 'exit'-event is triggered by the user ( the latter case makes the execution fall through all of the running execute() calls and terminates the `run Prima;` call, exiting gracefully).

**execute_shared INSERT_BEFORE = undef**

Switches the window to the shared modal state and put it on top of all non-modal windows that belong to the same modal horizon. If the window has the `::modalHorizon` property value set to 1, starts its own stack, independent of all other window stacks.

By default, if the INSERT_BEFORE object is undef, the window is also put on top of other shared-modal windows in the same stack. If INSERT_BEFORE is one of the shared-modal windows in the stack, the window is placed in the queue before the INSERT_BEFORE window.

The window is made visible and enabled, if necessary, and the `Execute` notification is triggered.

The function returns immediately.

**fullscreen**

Sets the window in the fullscreen mode. A shortcut for the `windowState(ws::Fullscreen)` call.

**get_client_handle**

Returns the system handle for the special client window that is inserted in the top-level window and covers all of its areas. It is different from the `get_handle` method in that the latter returns the system handle of the top-level window itself. In other terms, the handle returned by this function is a child of the window returned by `get_handle`.

See also: `get_handle`

**get_default_menu_font**

Returns the default font for the Prima::Menu class.

**get_modal**

Returns one of the three constants that reflect the modal state of the window:

```
mt::None
mt::Shared
mt::Exclusive
```

The value of `mt::None` is 0, so the result of get_modal() can be also treated as a boolean value if one needs to check if the window is modal or not.

**get_modal_window MODALITY_TYPE = mt::Exclusive, NEXT = 1**

Returns the modal window that is next to the given window in the modality chain. MODALITY_TYPE selects the chain, and can be either `mt::Exclusive` or `mt::Shared`. NEXT is the boolean flag selecting the lookup direction; if it is 1, the 'upper' window is returned, otherwise the 'lower' one ( in a simple case when the window A is made modal (executed) after the modal window B, the A window is the 'upper' one ).

If the window has no immediate modal siblings, `undef` is returned.

**maximize**

Maximizes the window. A shortcut for `windowState(ws::Maximized)`.

**minimize**

Minimizes the window. A shortcut for `windowState(ws::Minimized)`.

**ok**

The standard method to dismiss the modal window with the `mb::OK` result. Typically the effect of calling this method is equal to when the user presses the enter key of on the modal window, signalling that the default action is to be taken.

See also: `cancel`, `modalResult`, `execute`, `execute_shared`

**restore**

Restores the window to a normal state from the minimized or maximized state. A shortcut for `windowState(ws::Normal)`.

### Events

**Activate**

Triggered when the window is activated by the user. The active window is the one that has the keyboard focus; its decorations are usually highlighted by the system.

The toolkit does not provide a standalone activation function, the `select()` method is used for this instead.

**Deactivate**

Triggered when the window is deactivated by the user. The window is marked inactive when it has no keyboard focus.

The toolkit does not provide a standalone deactivation function, the `deselect()` method is used for this instead.

**EndModal**

Called before the window leaves the modal state.

**Execute**

Called as soon as the window enters the modal state.

**SysHandle**

Same as in the `Prima::Widget` class, but in addition to the Widget properties that may trigger the event, the following `Window` properties can trigger it as well: the *taskListed* entry, the *borderIcons* entry, the *borderStyle* entry, the *onTop* entry

**WindowState STATE**

Triggered when the window state is changed, either by an explicit `windowState()` call or by the user. STATE is the new window state, one of the four `ws::XXX` constants.

## 3.12 Prima::Clipboard

GUI interprocess data exchange

### Description

Prima::Clipboard is an interface to system clipboards. Depending on the OS, there can be only one clipboard (Win32), or three (X11). The class is also used for data exchange in drag-and-drop interactions.

### Synopsis

```
my $c = $::application-> Clipboard;

# paste data
my $string = $c-> text;
my $image  = $c-> image;
my $other  = $c-> fetch('Other type');

# copy datum
$c-> text( $string);

# copy data
$c-> open;
$c-> text( $string);
$c-> image( $image);
$c-> store( $image);
$c-> close;

# clear
$c-> clear;
```

### Usage

Prima::Clipboard provides access to the system clipboard data storage. For easier communication, the system clipboard has one 'format' field, which is stored along with the data. This field is used to distinguish between data formats. Moreover, a clipboard can hold simultaneously several data instances, of different data formats. Since the primary usage of a clipboard is 'copying' and 'pasting', an application can store copied information in several formats, increasing the possibility that the receiving application can recognize the data.

Different systems provide a spectrum of predefined data types, but the toolkit uses only three of these out of the box - ascii text, utf8 text, and image. It does not limit, however, the data format being one of these three types - an application is free to register its own formats. Both predefined and newly defined data formats are described by a string, while the three predefined formats are represented by the `'Text'`, `'UTF8'`, and `'Image'` string constants.

The most frequent usage of Prima::Clipboard is to perform two tasks - copying and pasting. Both can be exemplified by the following:

```
my $c = $::application-> Clipboard;

# paste
my $string = $c-> text;

# copy
$c-> text( $string);
```

Here is what happens under the hood:

First, the default clipboard is accessible by an implicit name call, as an object named 'Clipboard'. This scheme makes it easily overridable. A more important point is, that the default clipboard object might be accompanied by other clipboard objects. This is the case with the X11 environment, which defines also 'Primary' and 'Secondary' system clipboards. Their functionality is identical to the default clipboard, however. `get_standard_clipboards()` method returns strings for the clipboards, provided by the system.

Second, the code for fetching and/or storing multi-format data is somewhat different. Clipboard is viewed as a shared system resource and has to be 'opened' before a process can grab it, so other processes can access the clipboard data only after the clipboard is 'closed' ( note: It is not so under X11, where there is no such thing as clipboard locking, -- but the toolkit imposes this model for the consistency sake).

`fetch()` and `store()` implicitly call `open()` and `close()`, but these functions must be called explicitly for the multi-format data handling. The code below illustrates the following:

```
# copy text and image
if ( $c-> open) {
   $c-> clear;
   $c-> store('Text', $string);
   $c-> store('Image', $image);
   $c-> close;
}

# check present formats and paste
if ( $c-> open) {
  if ( $c-> format_exists('Text')) {
     $string = $c-> fetch('Text');
  }
  # or, check the desired format alternatively
  my %formats = map { $_ => 1 } $c-> get_formats;
  if ( $formats{'Image'}) {
     $image = $c-> fetch('Image');
  }

  $c-> close;
}
```

The clear() call in the copying code is necessary so the newly written data will not mix with the old.

At last, the newly registered formats can be accessed by the following example code:

```
my $myformat = 'Very Special Old Pale Data Format';
if ( $c-> register_format($myformat)) {
   $c-> open;
   $c-> clear;
   $c-> store('Text', 'sample text');
   $c-> store($myformat', 'sample ## text');
   $c-> close;
}
```

**On-demand storage**

Under X11 it is possible to skip the generation of data in all possible clipboard formats when copying the data. The native X11 mechanism allows to ask the source application for the exact data format needed by the target application, and the toolkit uses the special notification `onClipboard` triggered on the application whenever necessary.

By default, this event handler responds to querying images in file-encoded formats (gif,jpg) under X11 on the fly. It can be extended to generate other formats as well. See the **Events** entry in the *Prima::Application* section Clipboard for the details.

### Custom formats

Once registered, all processes in the GUI space can access the data in this format. The registration must take place also if a Prima-driven program needs to read data in a format, defined by another program. In either case, the duplicate registration is a valid case. When no longer needed, the format can be de-registered. It is not a mandatory action, however - the toolkit de-registers these formats before exiting. Moreover, the system maintains a reference counter on the custom-registered formats; de-registering thus does not mean deletion. If two processes use a custom format, and one exits and re-starts, the other still can access the data in the same format, registered by its previous incarnation.

### Unicode

Applications can interchange text in both ascii and utf8, leaving the selection choice to reader programs. While it is possible to access both at the same time, by `fetch`'ing content of `Text` and `UTF8` clipboard slots, the widget proposes its own pasting scheme, where the mechanics are hidden under the `text` property call. The property is advised to be used instead of individual `'Text'` and `'UTF8'` formats. This method is used in all the standard widgets and is implemented so the programmer can reprogram its default action by overloading the `PasteText` notification of `Prima::Application` ( see the **PasteText** entry in the *Prima::Application* section ).

The default action of `PasteText` is to query first if the `'Text'` format is available, and if so, return the ascii text scalar. If `Prima::Application::wantUnicodeInput` is set (default), the `'UTF8'` format is checked before resorting to `'Text'`. This scheme is not the only possibly needed, for example, an application may want to ignore ASCII text, or, recognize UTF8 text but have the `Prima::Application::wantUnicodeInput` cleared, etc.

The symmetric action is `CopyText`, which allows for a custom text conversion code to be installed.

### Images

Image data can be transferred in different formats in different OSes. The lowest level is raw pixel data in display-based format, whereas GTK-based applications can also exchange images in file-based formats, such as bmp, png, etc. To avoid further complications in the implementations, the `PasteImage` action was introduced to handle these cases, together with a symmetrical `CopyImage`.

The default action of `PasteImage` is to check whether lossless encoded image data is present, and if so, load a new image from this data, before falling back to the OS-dependent image storage.

When storing the image on the clipboard, only the default format, raw pixel data is used. Under X11 the toolkit can also serve images encoded as file formats.

Note: Under X11 you'll need to keep the image alive during the whole time it might get copied from the application - Prima doesn't keep a copy of the image, only the reference. Changing the image after it was stored in the clipboard **will** affect the clipboard content.

### Exact and meta formats

Prima registers two special *meta formats*, `Image` and `Text`, that interoperate with the system clipboard, storing data in the format that matches best with system conventions when copying and pasting images and text, correspondingly. It is recommended to use the meta-format calls (has_format, text, image, copy, paste) rather than exact format calls (format_exists, store, fetch) whenever possible.

Where the exact format method operates on a single format data storage, meta format calls may operate on several exact formats. F.ex. `text` can check whether there exists a UTF-8 text

storage before resorting to 8-bit text. `image` on X11 is even more complicated and may use image codecs to transfer encoded PNG streams, for example.

### Special system formats

Warning: this section is experimental.

Under win32, it is possible to access files dropped from Explorer. The application must register a special `Win32.CF_HDROP` format, and read raw binary data in the `onDragEnd` handler:

```
use constant CF_HDROP => "Win32.CF_HDROP";
$::application-> Clipboard-> register_format( CF_HDROP );

onDragEnd => sub {
        my ( $self, $clipboard, $action, $modmap, $x, $y, $counterpart, $ref) = @_;
        if ($clipboard->has_format(CF_HDROP)) {
                my $raw = $clipboard->fetch(CF_HDROP);
                my ($offset, $x, $y, $nonclient, $wide) = unpack("Llllll", $raw);
                my $files = substr($raw, $offset);
                if ($wide) {
                        use Encode;
                        $files = Encode::decode("utf-16le", $files);
                }
                my @files = split "\x0", $files;
                print "dropped files: [@files] at [$x,$y]\n";
        }
}
```

## API

### Properties

**image OBJECT, [KEEP]**

Provides access to an image, stored in the system clipboard. In the get-mode call returns `undef` if no image is stored. In the set-mode clears the clipboard unless the KEEP flag is set.

**text STRING, [KEEP]**

Provides access to the text stored in the system clipboard. In the get-mode returns `undef` if no text information is present. In the set-mode clears the clipboard unless the KEEP flag is set.

### Methods

**clear**

Deletes all data from the clipboard.

**close**

Closes the open/close brackets. open() and close() can be called recursively. Only the last close() removes the actual clipboard locking, so that other processes can use it as well.

**copy Format, DATA, KEEP**

Sets DATA in Format. Clears the clipboard before unless the KEEP flag is set.

**deregister_format FORMAT_STRING**

De-registers a previously registered data format. Called implicitly for all custom formats before the program exits.

**fetch FORMAT_STRING**

Returns the data of exact FORMAT_STRING data format, if present in the clipboard. Depending on the FORMAT_STRING, data is either a text string for the `'Text'` format, a Prima::Image object for the `'Image'` format, or a binary scalar value for all custom formats.

**format_exists FORMAT_STRING**

Returns a boolean flag, reflecting whether the FORMAT_STRING exact format data is present in the clipboard or not.

**has_format FORMAT_STRING**

Returns a boolean flag, reflecting whether the FORMAT_STRING meta format data is present in the clipboard or not.

**get_handle**

Returns the system handle for the clipboard object.

**get_formats INCLUDE_UNREGISTERED = 0**

Returns an array of strings, where each is a format ID, reflecting the formats present in the clipboard.

Only the predefined formats, and the formats registered via `register_format()` are returned if `INCLUDE_UNREGISTERED` is unset. If the flag is set, then all existing formats are returned, however, their names are not necessarily the same as those registered with Prima.

**get_registered_formats**

Returns an array of strings, each representing a registered format. `Text` and `Image` are returned also.

**get_standard_clipboards**

Returns an array of strings, each representing a system clipboard. The default `Clipboard` is always present. Other clipboards are optional. As an example, this function returns only `Clipboard` under win32, but also `Primary` and `Secondary` under X11. The code, specific to these clipboards must refer to this function first.

The drag-and-drop clipboard name is also returned here; it is system-specific.

**is_dnd**

Returns 1 if the clipboard is the special clipboard used as a proxy for drag-and-drop interactions.

See also: `Widget/Drag and drop`, `Application/get_dnd_clipboard`.

**open**

Opens a system clipboard and locks it for the process single use; returns a success flag. Subsequent `open` calls are possible and always return 1. Each `open()` must correspond to `close()`, otherwise the clipboard will stay locked until the blocking process is finished.

**paste FORMAT_STRING**

Returns data of meta format FORMAT_STRING if found in the clipboard, or undef otherwise.

**register_format FORMAT_STRING**

Registers a data format under FORMAT_STRING string ID, and returns a success flag. If a format is already registered, 1 is returned. All formats, registered via `register_format()` are de-registered with `deregister_format()` when a program is finished.

**store FORMAT_STRING, SCALAR**

Stores SCALAR value into the clipboard in FORMAT_STRING exact data format. Depending on FORMAT_STRING, the SCALAR value is treated as follows:

```
FORMAT_STRING     SCALAR
-----------------------------------
Text              text string in ASCII
UTF8              text string in UTF8
Image             Prima::Image object
other formats     binary scalar value
```

Note: All custom formats are treated as binary data. In case when the data are transferred between hosts with different byte orders no implicit conversions are made. It is up to the programmer whether to convert the data into a portable format or leave it as is. The former option is of course preferable. As far as the author knows, the *Storable* module from the *CPAN* collection provides the system-independent conversion routines.

## 3.13 Prima::Menu

Pull-down and pop-up menu objects

### Synopsis

```
use Prima;
use Prima::Application;

my $window = Prima::Window-> new(
     menuItems => [
        [ '~File' => [
           [ '~Open', 'Ctrl+O', '^O', \&open_file ],
           [ '-save_file', '~Save', km::Ctrl | ord('s'), sub { save_file() } ],
           [],
           [ '~Exit', 'Alt+X', '@X', sub { exit } ],
        ]],
        [ '~Options' => [
           [ '*option1'  => 'Checkable option' => sub { $_[0]-> menu-> toggle( $_[1]) }],
           [ '*@option2' => 'Checkable option' => sub {}], # same
        ]],
        [],
        [ '~Help' => [
           [ 'Show help' => sub { $::application-> open_help("file://$0"); }],
        ]],
     ],
);

sub open_file
{
    # enable 'save' menu item
    $window-> menu-> save_file-> enable;
}

$window-> popupItems( $window-> menuItems);
```

### Description

The document describes the interfaces of Prima::AbstractMenu class, and its three descendants - Prima::Menu, Prima::Popup, and Prima::AccelTable. Prima::AbstractMenu is a descendant of the Prima::Component class, and its specialization is the handling of menu items, held in a tree-like structure. Descendants of Prima::AbstractMenu are designed to be attached to widgets and windows, to serve as hints for the system-dependent pop-up and pull-down menus.

### Usage

#### Menu items

The central point of functionality in Prima::AbstractMenu-derived classes and their object instances ( further referred to as 'menu classes' and 'menu objects'), is the handling of a complex structure, contained in the `::items` property. This property is special in that its structure is a tree-like array of scalars, each of which is either a description of a menu item or a reference to an array.

Parameters of an array must follow a special syntax, so the property input can be parsed and assigned correctly. In general, the syntax is

```
$menu-> items( [
   [ menu item description ],
   [ menu item description ],
   ...
]);
```

where the 'menu item description' is an array of scalars, that can hold from 0 up to 6 elements. Each menu item has six fields, that qualify a full description of a menu item. The shorter arrays are the shortcuts that imply some default or special cases. These base six fields are:

**Menu item name**

A string identifier. There are defined several shortcut properties in the Prima::MenuItem namespace that access the menu items and their data by the name. If the menu item name is not given or is empty, the name is assigned a string in the form '#ID' where the ID is a unique integer value within the menu object.

The IDs are set for each menu item, disregarding whether they have names or not. Any menu item can be uniquely identified by its ID value, by supplying the '#ID' string, in the same fashion as the named menu items. When creating or copying menu items, names in the format '#ID' are ignored and treated as if an empty string is passed. When copying menu items to another menu object, all menu items to be copied change their IDs, but the explicitly set names are preserved. Since the anonymous menu items do not have names their auto-generated names change also.

If the name is prepended by the special characters ( see below ), these characters are not treated as a part of the name but as an item modifier. This syntax is valid only for `::items` and `insert()` functions, not for `set_variable()` method.

**`-` - the item is disabled**

**`*` - the item is checked**

**`@` - the item is using auto-toggling**

**`?` - the item is custom drawn**

Expects the `onMeasure` and `onPaint` callbacks in `options`

**`(` and `)` - radio group**

The items marked with parentheses are treated as a part of a group, where only a single item can be checked at any time. Checking and unchecking happen automatically.

A group is only valid on the same level where it was defined (i.e. submenus are not a part of the group). A group is automatically closed on the separator item. If that is not desired, mark it as `(` too (consequent `(`'s are allowed):

```
[ '(one' ... ]
[ 'two' ... ]
[ '(' ],
[ ')last' ... ]
```

If the user hits an already checked item then nothing happens. However, when combined with auto-toggling (i.e. marked with `(@`), a checked item becomes unchecked, thus the group can present a state where no items are checked as well.

See also: `group`

**Menu text / menu image**

A non-separator menu item can be visualized either as a text string or an image. These options exclude each other and therefore occupy the same field. The menu text is an arbitrary string, with the `~` ( tilde ) character escaping a shortcut character, so that the system uses

it as a hotkey during the menu navigation. The menu image is a the *Prima::Image* section object.

Note: the tilde-marked character is also recognized when navigating the custom drawn menu items, even though they not necessarily might draw the highlighted character.

The menu text in the menu item is accessible via the `::text` property, and the menu image via the `::image` property. Only one of these could be used, depending on whether the menu item contains text or image.

**Accelerator text**

An alternative text string that appears next to the menu item or the menu image, usually serving as a hotkey description. For example, if the hotkey is a combination of the 'enter' and the 'control' keys, then usually the accelerator text is the 'Ctrl+Enter' string.

The accelerator text in the menu item is accessible via the `::accel` property.

Note: there is the `Prima::KeySelector::describe` function which converts an integer key value to a string in the human-readable format, perfectly usable as accelerator text.

**Hotkey**

An integer value, is a combination of either a `kb::XXX` constant or a character index with the modifier key values ( `km::XXX` constant ). This format is less informative than the three-integer key event format (CODE,KEY,MOD), described in the *Prima::Widget* section. However, these formats are easily converted to each other: CODE,KEY,MOD are translated to the INTEGER format by the `translate_key()` method. The reverse operation is not needed for the `Prima::AbstractMenu` functionality and is performed by the `Prima::KeySelector::translate_codes` method.

The integer value can be given in a more readable format when calling the `::items` method. Character and F-keys (from F1 to F16) can be used as string literals, without the `kb::` constant, and the modifier keys can be hinted as prefix characters: km::Shift as '#', km::Ctrl as '^', and km::Alt as '@'. This way the combination of the 'control' and 'G' keys can be expressed as the `'^G'` literal, and 'control'+'shift'+'F10' - as `'^#F10'`.

The hotkey in menu items is accessible via the `::key` property. This property accepts the literal key format described above.

A literal key string can be converted to an integer value by the `translate_shortcut` method.

When the user presses the key combination that matches the hotkey entry in a menu item, its action is triggered.

**Action**

Every non-separator and non-submenu item performs an action that needs to be defined explicitly. The action can be set either as an anonymous sub or as a string with the name of the method on the owner of the menu object. Both ways have their niches, and both use three parameters when called - the owner of the menu object, the name of the menu item, that triggered the action, and the new checked status of the menu item

```
Prima::MainWindow-> new(
     menuItems => [
             ['@item', 'Test',
             sub {
                 my (
                    $window,  # MainWindow
                    $item,    # 'item'
                    $checked  # MainWindow->men('item')->checked
                 ) = @_;
```

```
                }],
        ]
    );
```

The action scalar in the menu item is accessible via the `::action` property.

A special built-in action can automatically toggle a menu item without the need to program that explicitly. The manual toggle of the menu item can be done by a code like this:

```
$window->menu->toggle($item)
```

However, Prima can toggle the item automatically too, if the `@` character is added to the menu item name (see the *Menu item name* entry).

**Options**

At last, the non-separator menu items can hold an extra hash in the `options` property. The toolkit reserves the following keys for internal use:

**group INTEGER**

Same as the `group` property.

**icon HANDLE**

Is used to replace the default checkmark bitmap on a menu item

**onMeasure MENUITEM, REF**

Required when the custom painting is requested. It is called when the system needs to query the menu item dimensions. `REF` is a 2-item arrayref that needs to be set with the pixel dimensions of the item.

**onPaint MENUITEM, CANVAS, SELECTED, X1, Y1, X2, Y2**

Required when custom painting is requested. It is called whenever the system needs to draw the menu item. The X1 - Y2 are the coordinates of the rectangle where the drawing is allowed.

The syntax of the `::items` method does not provide the 'disabled' and the 'checked' states for a menu item as separate fields. These states can be only set by using the `-` and the `*` prefix characters, as described above, in the *Menu item name* entry. They can though be assigned later on a per-item basis via the `::enabled` and the `::checked` properties when the menu object is created.

All these fields comprise the most common type of a menu item, that has a text, a shortcut key, and an action - a 'text item'. However, there are also two other types of menu items - a sub-menu and a separator. The type of the menu item cannot be changed on the fly except by changing the full menu tree by the functions `::items`, `remove()`, and `insert()`.

A sub-menu item can hold the same references as a text menu item does, except for the action field. Instead, the action field is used for a sub-menu reference scalar pointing to another set of menu item description arrays. From that point of view, the syntax of `::items` can be more elaborated and shown in the following example:

```
$menu-> items( [
   [ text menu item description ],
   [ sub-menu item description [

      [ text menu item description ],
      [ sub-menu item description [
          [ text menu item description ],
          ...
```

```
        ]
        [ text menu item description ],
        ...
    ] ],
    ...
]);
```

The separator items don't have any fields, except the name. Their purpose is to hint a logical division of the menu items, usually as non-selectable horizontal lines.

In the menu bars, the first separator item met by the menu parser is treated differently. It serves as a hint that the following items must be shown in the right corner of the menu bar, contrary to the left-adjacent default layout. Subsequent separator items in a menu bar declaration can be either shown as a vertical division bar, or ignored.

All of these menu item types can be constructed by specifying menu description arrays. An item description array can hold between 0 to 6 scalars, and each combination is treated differently:

**six - [ NAME, TEXT/IMAGE, ACCEL, KEY, ACTION/SUBMENU, DATA ]**

A six-scalar array is a fully qualified text-item description. All fields correspond to the described above scalars.

**five [ NAME, TEXT/IMAGE, ACCEL, KEY, ACTION/SUBMENU ]**

Same as the six-scalar syntax, but without the DATA field. If DATA is skipped then it is set to `undef`.

**four [ TEXT/IMAGE, ACCEL, KEY, ACTION/SUBMENU ] or [ NAME, TEXT/IMAGE, ACTION/SUBMENU, DATA ]**

One of the two definitions, depending on whether the last item is a hashref or not.

If the last item is not a hashref, then treated the same as the five-scalar syntax, but without the NAME field. When NAME is skipped it is assigned to a unique string within the menu object.

Otherwise same as the three-scalar syntax plus the DATA hashref.

**three [ NAME, TEXT/IMAGE, ACTION/SUBMENU ] or [ TEXT/IMAGE, ACTION/SUBMENU, DATA ]**

One of the two definitions, depending on whether the last item is a hashref or not.

If the last item is not a hashref, then treated the same as the five-scalar syntax, but without the ACCEL and the KEY fields. KEY is `kb::NoKey` by default, so no keyboard combination is bound to the item. The default ACCEL value is an empty string.

Otherwise the same as the two-scalar syntax plus DATA hashref.

**two [ TEXT/IMAGE, ACTION/SUBMENU ] or [ NAME, DATA ]**

One of the two definitions, depending on whether the last item is a hashref or not.

If the last item is not a hashref, then treated the same as the three-scalar syntax, but without the NAME field.

Otherwise treated as the menu items with the data reference. Useful for custom menu items that need at least the `'?'` flag in the NAME.

**one and zero [ NAME ]**

Both empty and 1-scalar arrays define a separator menu item. In the case of the 1-scalar syntax, the scalar value is the name of the separator item.

As an example of all the above, here's an example of a menu tree:

```
$img = Prima::Image-> create( ... );
...
$menu-> items( [
   [ "~File" => [
       [ "Anonymous" => "Ctrl+D" => '^d' => sub { print "sub\n";}],   # anonymous sub
       [ $img => sub {
          my $img = $_[0]-> menu-> image( $_[1]);
          my @r = @{$img-> palette};
          $img-> palette( [reverse @r]);
          $_[0]->menu->image( $_[1], $img);
       }],                          # image
       [],                          # division line
       [ "E~xit" => "Exit"    ]    # calling named function of menu owner
   ]],
   [ ef => "~Edit" => [                  # example of system commands usage
      ...
      [ "Pa~ste" => sub { $_[0]->foc_action('paste')} ],
      ...
      ["~Duplicate menu"=>sub{ TestWindow->create( menu=>$_[0]->menu)}],
   ]],
   ...
   [],                              # divisor in the main menu opens
   [ "~Clusters" => [               # right-adjacent part
     [ "*".checker =>  "Checking Item"   => "Check"     ],
     [],
     [ "-".slave   =>  "Disabled state"   => "PrintText"],
     ...
   ]]
] );
```

The code is from the *examples/menu.pl* in the toolkit installation. The reader is advised to run the example and learn the menu mechanics.

### Prima::MenuItem

As briefly mentioned above, all menu items can be accessed using the following properties: `::accel`, `::text`, `::image`, `::checked`, `::enabled`, `::action`, `::data`. These, plus some other methods can be also called in an alternative way, resembling name-based component calls of the *Prima::Object* section. For example, the call

```
$menu-> checked('CheckerMenuItem', 1);
```

can be also written as

```
$menu-> CheckerMenuItem-> checked(1);
```

Such name-based calls create temporary Prima::MenuItem objects that are only used to mimic the accessor functions from the Prima::AbstractMenu class and not much else.

### Prima::Menu

The Prima::Menu objects complement the Prima::Window objects so that their menu items are shown as the menu bar on top of the window.

Prima::Menu's top-level items are laid out horizontally, and the top-level separator items behave differently ( see above, the *Menu items* entry ).

If the `::selected` property is set to 1, then a menu object is visualized in a window, otherwise it is not. This behavior allows a window to host multiple menu objects without interfering with each other. When a Prima::Menu object gets 'selected', it displaces the previous 'selected' menu, and its items are installed in the window menu bar. The Prima::Window property `::menu` then points to that new menu object. Another Prima::Window property `::menuItems` is an alias for the `::items` property of the currently selected menu object. Prima::Window's properties `::menuFont` and `::menuColorIndex` are used as visualization hints, if/when the system supports that.

Prima::Menu provides no new methods or properties.

### Prima::Popup

Objects derived from the Prima::Popup class are used together with the Prima::Widget objects in the same way as the menu objects with the window objects. Popup items are shown when the user presses the system-defined pop-up key or mouse button, as a response to the Prima::Widget's `Popup` notification.

If the `::selected` property is set to 1, and the `autoPopup` property is also set to 1, then a popup object can appear fully automatically, without the need to program the popup-menu appearance and handling. This behavior allows a widget to host multiple popup objects without interfering with each other. When a Prima::Popup object gets 'selected', it displaces the previous 'selected' popup object. The Prima::Widget property `::popup` then points to that object. Another widget property `::popupItems` is an alias for the `::items` property of the currently selected popup object. Prima::Widget's properties `::popupFont` and Prima::Widgets's properties `::popupFont` and `::popupColorIndex` are used as visualization hints, if/when the system supports that.

A Prima::Popup object can be also visualized explicitly, by calling the `popup` method.

### Prima::AccelTable

This class has a more limited functionality than Prima::Menu or Prima::Popup and is primarily used for mapping keystrokes to actions. Prima::AccelTable objects are never visualized, and consume no system resources, although the full menu item management syntax is supported.

If the `::selected` property is set to 1, then an acceltable object displaces the previous 'selected' acceltable object. The Prima::Widget property `::accelTable` then points to that object. Another widget property `::accelItems` is an alias for the `::items` property of the currently selected acceltable object.

Prima::AccelTable provides no new methods or properties.

## API

### Properties

**accel NAME, STRING / Prima::MenuItem::accel STRING**

Manages accelerator text for the menu item. NAME is the name of the menu item.

**action NAME, SCALAR / Prima::MenuItem::action SCALAR.**

Manages the action for the menu item. NAME is the name of the menu item. SCALAR can be either an anonymous sub or a method name, defined in the menu object owner's namespace. Both are called with three parameters - the owner of the menu object, the menu object itself, and the name of the menu item.

**autoPopup BOOLEAN**

Only in Prima::Popup

If set to 1 in the selected state, calls the `popup()` method in response to the `Popup` notification, when the user presses the system-defined hotkey or mouse button combination.

If 0, the pop-up menu can only be shown by a call to the `popup` method programmatically.

Default value: 1

**autoToggle NAME, SCALAR / Prima::MenuItem::autoToggle SCALAR.**

Manages the autoToggle flag for the menu item. When set, the `checked` option is flipped when the user selects the item. Also, in the unchecked state, the system displays an empty check box icon where normally a check icon would appear, to hint to the user that the menu item is toggle-able, despite it being unchecked.

**checked NAME, BOOLEAN / Prima::MenuItem::checked BOOLEAN**

Manages the 'checked' state of a menu item. If 'checked', a menu item is visualized with a distinct checkmark near the menu item text or image. Its usage with the sub-menu items is possible, although discouraged.

NAME is the name of the menu item.

**data NAME, HASH / Prima::MenuItem::data HASH**

Manages the user data hash. NAME is the name of the menu item.

**enabled NAME, BOOLEAN / Prima::MenuItem::enabled BOOLEAN**

Manages the 'enabled' state of the menu item. If 'enabled' is set, a menu item is visualized with a grayed or otherwise dimmed color palette. If a sub-menu item is disabled, the whole sub-menu is inaccessible.

Default: true

NAME is the name of the menu item.

**group NAME, GROUP_ID / Prima::MenuItem::group GROUP_ID**

If not 0, the menu item is treated as a member of a radio group with the GROUP_ID number. That means if one of the menu items that belong to the same group is checked, the other items are automatically unchecked.

**image NAME, OBJECT / Prima::MenuItem::image OBJECT**

Manages the image that is bound to the menu item. The OBJECT is a non-null Prima::Image object reference, with no particular color space or dimensions ( because of dimensions, its usage in top-level Prima::Menu items is discouraged ).

The `::image` and the `::text` properties are mutually exclusive, and can not be set together, but a menu item can change its representation between an image and a text during the runtime if these properties are called.

NAME is the name of the menu item.

**items SCALAR**

Manages the whole menu items tree. SCALAR is a multi-level anonymous array structure, with the syntax described in the *Menu items* entry.

The `::items` property is an ultimate tool for reading and writing the menu items tree, but often it is too powerful, so there exist several easier-to-use properties `::accel`, `::text`, `::image`, `::checked`, `::enabled`, `::action`, `::data`, that can access menu items individually.

**key NAME, KEY / Prima::MenuItem::key KEY**

Manages the hotkey combination, bound with the menu item. Internally the KEY is kept as an integer value, and a get-mode call always returns integers. The set-mode calls, however, accept the literal key format - strings such as '^C' or 'F5'.

NAME is the name of the menu item; KEY is an integer value.

**selected BOOLEAN**

If set to 1, the menu object is granted extra functionality from a window or widget owner object. Different Prima::AbstractMenu descendants are equipped with different extra functionalities. In the *Usage* section, see the *Prima::Menu* section, the *Prima::Popup* section, and the *Prima::AccelTable* section.

Within each menu-owner object hierarchy , only one menu object can be selected for its owner.

If set to 0, the only actions performed are implicit hotkey lookup when on the `KeyDown` event.

Default value: 1

**submenu NAME, ARRAY / Prima::MenuItem::submenu ARRAY**

Manages a submenu, if it is present. A get-call of the `submenu` property is equivalent to the `get_items(NAME, 1)` call. On a set-call removes all of the items under the NAME and inserts new ones.

See also: the *is_submenu* entry.

**text NAME, STRING / Prima::MenuItem::text STRING**

Manages the text bound to the menu item. The STRING is an arbitrary string, with the '~' ( tilde ) escape character of a hotkey character. The hotkey character is only used when the keyboard navigation of a pop-up or the pull-down user action is performed; does not influence outside the menu sessions.

The `::image` and the `::text` properties are mutually exclusive, and can not be set together, but a menu item can change its representation between an image and a text during the runtime if these properties are called.

### Methods

**check NAME / Prima::MenuItem::check**

Alias for `checked(1)`. Sets the menu item in the checked state.

**disable NAME / Prima::MenuItem::disable**

Alias for `enabled(0)`. Sets the menu item in the disabled state.

**enabled NAME / Prima::MenuItem::enabled**

Alias for `enabled(1)`. Sets the menu item in the enabled state.

**execute NAME**

Calls the action associated with the menu item

**find_item_by_key KEY**

Finds items by the associated hotkey combination

**get_handle**

Returns a system-dependent menu handle.

NB: Prima::AccelTable uses no system resources, and this method returns its object handle instead.

**get_children NAME**

Returns the list of children of the menu item with the name NAME

**get_item NAME, FULL_TREE = 0**

Returns the item entry corresponding to NAME, with or without the eventual full tree of children items, depending on the FULL_TREE flag.

**get_items NAME, FULL_TREE = 1**

Returns immediate children items entries that have NAME as a parent, with or without the eventual full tree of children items, depending on the FULL_TREE flag.

**has_item NAME**

Returns a boolean value, that is true if the menu object has a menu item with the name NAME.

**insert ITEMS, ROOT_NAME, INDEX**

Inserts menu items inside the existing item tree. ITEMS has the same syntax as the `::items` property. ROOT_NAME is the name of the menu item, where the insertion must take place; if ROOT_NAME is an empty string, the insertion is performed to the top-level items. INDEX is an offset, that the newly inserted items would possess after the insertion. INDEX 0 indicates the very start of the menu.

Returns no value.

**is_separator NAME**

Returns true if the item is a separator, false otherwise

**is_submenu NAME**

Returns true if the item has a submenu, false otherwise

**popup X_OFFSET, Y_OFFSET, [ LEFT = 0, BOTTOM = 0, RIGHT = 0, TOP = 0 ]**

Only in Prima::Popup

Executes the system-driven pop-up menu, in the location near (X_OFFSET,Y_OFFSET) pixel on the screen, with the items from the `::items` tree. The pop-up menu is hinted to be positioned so that the rectangle, defined by (LEFT,BOTTOM) - (RIGHT,TOP) coordinates is not covered by the first-level menu. This is useful when a pop-up menu is triggered by a button widget, for example.

If during the execution the user selects a menu item, then its associated action is executed ( see `action` ).

The method returns immediately and returns no value.

There is no functionality to cancel the running popup session.

**remove NAME / Prima::MenuItem::remove**

Deletes the menu item named NAME from the items tree, and its eventual sub-menus

**select**

Alias for `selected(1)`. Sets the menu object in the selected state, and deselects all menu siblings of the same type (ie Menu->select(1) won't affect the selected status for a popup, for example).

**set_variable NAME, NEW_NAME**

Changes the name of the menu item from NAME to NEW_NAME. NEW_NAME must not be an empty string and must not be in the '#integer' form.

**toggle NAME / Prima::MenuItem::toggle**

Toggles the checked state of the menu item and returns the new state.

**translate_accel TEXT**

Locates a '~' ( tilde ) - escaped character in the TEXT string and returns its index ( as ord(lc())), or 0 if no escaped characters were found.

The method can be called with no object.

**translate_key CODE, KEY, MOD**

Translates the three-integer key representation into the one-integer format and returns the integer value. The three-integer format is used in the `KeyDown` and the `KeyUp` notifications for Prima::Widget.

See the *Prima::Widget* section

The method can be called with no object.

**translate_shortcut KEY**

Converts string literal KEY string into the integer format and returns the integer value.

The method can be called with no object.

**uncheck NAME / Prima::MenuItem::uncheck**

Alias for `checked(0)`. Sets the menu item in the unchecked state.

**Events**

**Change ACTION [, NAME [, VALUE ]]**

Triggered when the structure of the menu tree is changed. ACTION is the method call that triggered that action, and NAME is the menu item name, when applicable. If NAME is an empty string, that means the affected menu item is the root of the item tree. VALUE is the new value, if applicable.

**ItemMeasure ITEMID, REF**

Called when the system needs to query the dimensions of a menu item that has the custom painting bit set. `REF` is a 2-item arrayref that needs to be set pixel-wise dimensions.

See also: the *Options* entry

**ItemPaint CANVAS, ITEMID, SELECTED, X1, Y1, X2, Y2**

Called whenever the system needs to draw a menu item that has the custom painting bit set. X1 - Y2 are the coordinates of the rectangle where the drawing is allowed.

See also: the *Options* entry

## Bugs

Menu colors and fonts don't work on Windows and probably never will.

## 3.14 Prima::Timer

Programmable periodical events

### Synopsis

```
my $timer = Prima::Timer-> create(
    timeout => 1000, # milliseconds
    onTick  => sub {
       print "tick!\n";
    },
);

$timer-> start;
```

### Description

The Prima::Timer class arranges for the periodical notifications to be delivered in certain time intervals. The notifications are triggered by the system and are seen as the `Tick` events. There can be many active Timer objects at one time, spawning events simultaneously.

### Usage

The Prima::Timer class is a descendant of the Prima::Component class. Objects of the Prima::Timer class are created in the standard fashion:

```
my $t = Prima::Timer-> create(
   timeout => 1000,
   onTick  => sub { print "tick\n"; },
);
$t-> start;
```

If no 'owner' is given, $::application is assumed.

Timer objects are created in the inactive state; no events are spawned by default. To start spawning events, the `start()` method must be explicitly called. The time interval value is assigned by calling the <::timeout> property.

When the system generates a timer event, no callback is called immediately, - an event is pushed into the internal event stack instead, to be delivered during the next event loop. Therefore it cannot be guaranteed that the onTick noitifications will be called precisely after a timeout. A more accurate timing scheme, as well as timing with a precision of less than a millisecond, is not supported by the toolkit.

### API

#### Properties

**timeout MILLISECONDS**

Manages time intervals between the `Tick` events. In the set-mode call, if the timer is in the active state already ( see `get_active()`, the new timeout value is applied immediately.

#### Methods

**get_active**

Returns the boolean flag that reflects whether the object is in the active state or not. In the active state `Tick` events are spawned after `::timeout` time intervals.

**get_handle**

Returns the system-dependent handle of the printer object

**start**

Sets the object in the active state. If succeeds or if the object is already in the active state, returns 1. If the system is unable to create a system timer instance, the value of 0 is returned.

**stop**

Sets object in the inactive state.

**toggle**

Toggles the timer state

### Events

**Tick**

The system-generated event spawned every `::timeout` milliseconds if the object is in the active state.

## 3.15 Prima::Application

The root of the widget hierarchy

### Description

The Prima::Application class serves as the hierarchy root for the majority of Prima objects. All toolkit widgets are ultimately owned by the application object. There can be only one instance of the Prima::Application class at a time.

### Synopsis

```
use Prima qw(Application);
Prima::MainWindow->new();
run Prima;
```

### Usage

Prima::Application class and its only instance are treated is a special way in the toolkit's paradigm. Its only object instance is stored in the

```
$::application
```

scalar, defined in *Prima.pm* module. The application instance must be created whenever a widget, window, or event loop functionality is needed. Usually the

```
use Prima::Application;
```

or

```
use Prima qw(Application);
```

code is enough, but *$::application* can also be created and assigned explicitly. The 'use' syntax has an advantage as more resistant to eventual changes in the toolkit design. It can also be used in conjunction with custom parameters hash like the new() syntax:

```
use Prima::Application name => 'Test application', icon => $icon;
```

In addition to this functionality, Prima::Application is also a wrapper to a set of system functions, not directly related to the object classes. This functionality is generally explained in the *API* entry.

#### Inherited functionality

Prima::Application is a descendant of Prima::Widget but does not conform strictly ( in the OO sense ) to any of the built-in classes. It has methods from both Prima::Widget and Prima::Window, also, the methods inherited from the Prima::Widget class may work quite differently. For example, the `::origin` property from Prima::Widget is also implemented in Prima::Application, but always returns (0,0), an expected but not much usable result. The `::size` property, on the contrary, returns the extent of the screen in pixels. There are a few properties inherited from Prima::Widget, which return actual but uninformative results, - `::origin` is one of those, but there are several others. The methods and properties, that are like `::size` providing different functionality, are described separately in the *API* entry.

### Global functionality

Prima::Application is a wrapper to a set of unrelated functions that do not belong to other classes. A notable example, the painting functionality that is inherited from the Prima::Drawable class, allows drawing on the screen, possibly overwriting the graphic information created by the other programs. Although it is still a subject to the begin_paint()/end_paint() brackets, this functionality does not belong to a single object and is considered global.

### Painting

As stated above, the Prima::Application class provides an interface to the on-screen painting. This mode is triggered by the begin_paint()/end_paint() methods, while the other pair, begin_paint_info()/end_paint_info() triggers the information mode. This three-state paint functionality is more thoroughly described in the *Prima::Drawable* section.

The painting on the screen surfaces under certain environments (XQuartz, XWayland) is either silently ignored or results in an error. There, `begin_paint` may return a false value (`begin_paint_info` though always true).

### Hints

`$::application` hosts a special `Prima::HintWidget` class object, accessible via `get_hint_widget()`, but with its color and font functions aliased ( see `::hintColor`, `::hintBackColor`, `::hintFont` ).

This widget serves as a hint label, floating over other widgets if the mouse pointer hovers longer than `::hintPause` milliseconds.

Prima::Application internally manages all of the hint functionality. The hint widget itself, however, can be replaced before the application object is created, using the `::hintClass` create-only property.

### Printer

The result of the the *get_printer* entry method points to an automatically created printer object, responsible for the system printing. Depending on the operating system, it is either Prima::Printer, if the system provides GUI printing capabilities, or generic Prima::PS::Printer, the PostScript/PDF document interface.

See the *Prima::Printer* section for details.

### Clipboard

`$::application` hosts a set of Prima::Clipboard objects created automatically to reflect the system-provided clipboard IPC functionality. Their number depends on the system, - under the X11 environment, there are three clipboard objects, and one under Win32.

There are no specific methods to access these clipboard objects, except bring() ( or the indirect name call ); the clipboard objects are named after the system clipboard names, which are returned by the Prima::Clipboard::get_standard_clipboards method.

The default clipboard is named *Clipboard*, and is accessible via the

```
my $clipboard = $::application-> Clipboard;
```

call.

See the *Prima::Clipboard* section for details.

### Help subsystem

The toolkit has a built-in help viewer, that understands perl's native POD ( plain old documentation ) format. Whereas the viewer functionality itself is a part of the toolkit that resides in the `Prima::HelpViewer` module, any custom help viewing module can be assigned.

The create-only `Prima::Application` properties `::helpClass` and `::helpModule` can be used to set these options.

`Prima::Application` provides two methods for communicating with the help viewer window: `open_help()` opens a selected topic in the help window, and `close_help()` closes the window.

### System-dependent information

A complex program will need eventually more information than the toolkit provides. Knowing the toolkit boundaries in some platforms, the program may change its behavior accordingly. Both these topics are facilitated by extra system information returned by Prima::Application methods. The `get_system_value` method returns a system-defined value for each of the `sv::XXX` constants, so the program can read the system-specific information. Another method `get_system_info` returns the short description of the system that augments perl's `$^O` variable.

The `sys_action` method is a wrapper to system-dependent functionality that is called in a non-portable way. This method is rarely used in the toolkit, its usage is discouraged, primarily because its options do not serve the toolkit design, its syntax is subject to changes, and cannot be relied upon.

### Exceptions and signals

By default Prima doesn't track exceptions caused by `die`, `warn`, and signals. Currently, it is possible to enable a GUI dialog tracking the `die` exceptions, by either operating the boolean `guiException` property or using the

```
use Prima qw(sys::GUIException)
```

syntax.

If you need to track signals or warnings you may do so by using standard perl practices. It is though not advisable to call Prima interactive methods directly inside signal handlers but use a minimal code instead. F.ex. code that would ask whether the user wants to quit would look like this:

```
use Prima qw(Utils MsgBox);
$SIG{INT} = sub {
   Prima::Utils::post( sub {
       exit if message_box("Got Ctrl+C", "Do you really want to quit?", mb::YesNo) == mb:
   });
};
```

and if you want to treat all warnings as potentially fatal, like this:

```
use Prima qw(Utils MsgBox);
$SIG{__WARN__} = sub {
   my ($warn, $stack) = ($_[0], Carp::longmess);
   Prima::Utils::post( sub {
       exit if $::application && Prima::MsgBox::signal_dialog("Warning", $warn, $stack) =
   });
};
```

See also: the *Die* entry, the **signal_dialog** entry in the *Prima::MsgBox* section

## API

### Properties

**autoClose BOOLEAN**

If set to 1, issues `close()` after the last top-level window is destroyed. Does not influence anything if set to 0.

This feature is designed to help with generic 'one main window' application layouts.

Default value: 0

**guiException BOOLEAN**

If set to 1, when a `die` exception is thrown, displays a system message dialog. allowing the user to choose the course of action -- to stop, to continue, etc.

Is 0 by default.

Note that the exception is only handled inside the `Prima::run` and `Prima::Dialog::execute` calls; if there is a call to f ex `Prima::Window::execute` or a manual event loop run with `yield`, the signal dialog will not be shown. One needs to explicitly call `$::application->notify(Die => $@)` and check the notification result to decide whether to propagate the exception or not.

The alternative syntax for setting `guiException` to 1 is the

```
use Prima::sys::GUIException;
```

or

```
use Prima qw(sys::GUIException);
```

statement.

If for some reason an exception is thrown during dialog execution, it will not be handled by Prima but by the current `$SIG{__DIE__}` handler.

See also the **signal_dialog** entry in the *Prima::MsgBox* section .

**icon OBJECT**

Holds the icon object associated with the application. If `undef`, the system-provided default icon is assumed. Prima::Window objects inherit this application icon by default.

**insertMode BOOLEAN**

The system boolean flag signaling whether text widgets through the system should insert ( 1 ) or overwrite ( 0 ) text on user input. Not all systems provide the global state of the flag.

**helpClass STRING**

Specifies the class of the object used as the help viewing package. The default value is Prima::HelpViewer. Run-time changes to the property do not affect the help subsystem until a call to `close_help` is made.

**helpModule STRING**

Specifies the perl module loaded indirectly when a help viewing call is made via the `open_help` method. Used when the `::helpClass` property is overridden and the new class is contained in a third-party module. Run-time changes to the property do not affect the help subsystem until a call to `close_help` is made.

**hintClass STRING**

Create-only property.

Specifies the class of the widget used as the hint label.

Default value: Prima::HintWidget

**hintColor COLOR**

The alias to the foreground color property of the hint label widget.

**hintBackColor COLOR**

The alias to the background color property of the hint label widget.

**hintFont %FONT**

The alias to the font property of the hint label widget.

**hintPause TIMEOUT**

Sets the timeout in milliseconds before the hint label is shown when the mouse pointer hovers over a widget.

**language STRING**

By default contains the user interface language deduced either from the `$ENV{LANG}` environment variable (unix) or a system default setting (win32). When changed, updates the `textDirection` property.

See also: `get_system_info`.

**modalHorizon BOOLEAN**

A read-only property. Used as the lowest-level modal horizon. Always returns 1.

**palette [ @PALETTE ]**

Used only within the paint and information modes. Selects solid colors in the system palette, as many as possible. PALETTE is an array of 8-bit integer triplets, where each is a red, green, and blue component.

**printerClass STRING**

Create-only property.

Specifies the class of the object used as the printer. The default value is system-dependent, but is either `Prima::Printer` or `Prima::PS::Printer`.

**printerModule STRING**

Create-only property.

Specifies the perl module loaded indirectly before the printer object of the `::printerClass` class is created. Used when the `::printerClass` property is overridden and the new class is contained in a third-party module.

**pointerVisible BOOLEAN**

Manages the system pointer visibility. If 0, hides the pointer so it is not visible in all system windows. Therefore this property usage must be considered with care.

**size WIDTH, HEIGHT**

A read-only property.

Returns two integers, the width and height of the screen.

**showHint BOOLEAN**

If 1, the toolkit is allowed to show the hint label over a widget. If 0, the display of the hint is forbidden. In addition to the functionality of the `::showHint` property in Prima::Widget, Prima::Application::showHint is another layer of hint visibility control - if it is 0, all hint actions are disabled, disregarding `::showHint` value in the widgets.

**skin SCALAR**

The same as the **skin** entry in the *Prima::Widget* section, but is mentioned here because it is possible to change the whole application skin by changing this property, f ex like this:

```
use Prima::Application skin => 'flat';
```

**textDirection BOOLEAN**

Contains the preferred text direction initially deduced from the preferred interface language. If 0 ( default ), the preferred text direction is left-to-right (LTR), otherwise right-to-left (RTL), f.ex. for Arabic and Hebrew languages.

The value is used as a default when shaping text and setting widget input direction.

**uiScaling FLOAT**

The property contains an advisory multiplier factor, useful for UI elements that have a fixed pixel value, but that would like to be represented in a useful manner when the display resolution is too high (on modern High-DPI displays) or too low (on ancient monitors).

By default, it acquires the system display resolution and sets the scaling factor so that when the DPI is 96 it is 1.0, 192 it is 2.0, etc. The increase step is 0.25, so that bitmaps may look not that distorted when scaled. However, when the value is manually set the step is not enforced and any value can be accepted.

See also: the **Stress** entry in the *Prima* section.

**wantUnicodeInput BOOLEAN**

Selects if the system is allowed to generate key codes in unicode. Returns the effective state of the unicode input flag, which cannot be changed if perl or the operating system does not support UTF8.

If 1, the `Prima::Clipboard::text` property may return UTF8 text from system clipboards is available.

Default value: 1

**Events**

**Clipboard $CLIPBOARD, $ACTION, $TARGET**

With (the only implemented) `$ACTION` *copy*, is called whenever another application requests clipboard data in the format `$TARGET`. This notification is handled internally to optimize image pasting through the clipboard. Since the clipboard pasting semantics in Prima is such that data must be supplied to the clipboard in advance, before another application can request it, there is a problem with which format to use. To avoid encoding an image or other complex data in all possible formats but do that on demand and in the format the other application wants, this notification can be used.

Only implemented for X11.

**CopyImage $CLIPBOARD, $IMAGE**

The notification stores `$IMAGE` in the clipboard.

**CopyText $CLIPBOARD, $TEXT**

The notification stores `$TEXT` in the clipboard.

**Die $@, $STACK**

Called when an exception occurs inside the event loop `Prima::run`. By default, consults the `guiException` property, and if it is set, displays the system message dialog allowing the user to decide what to do next.

**Idle**

Called when the event loop handled all pending events, and is about to sleep waiting for more.

**PasteImage $CLIPBOARD, $$IMAGE_REF**

The notification queries `$CLIPBOARD` for image content and stores in `$$IMAGE_REF`. The default action is that the `'Image'` format is queried. On unix, encoded formats `'image/bmp'`, `'image/png'` etc are queried if the default `'Image'` is not found.

The `PasteImage` mechanism can read images from the clipboard in the GTK environment.

**PasteText $CLIPBOARD, $$TEXT_REF**

The notification queries `$CLIPBOARD` for text content and stores it in the `$$TEXT_REF` scalar. Its default action is that only the `'Text'` format is queried if `wantUnicodeInput` is unset. Otherwise, the `'UTF8'` format is queried first.

The `PasteText` mechanism is devised to ease defining text unicode/ascii conversion between clipboard and standard widgets, in a unified way.

### Methods

**add_startup_notification @CALLBACK**

CALLBACK is an array of anonymous subs, which are all executed when the Prima::Application object is created. If the application object is already created during the call, CALLBACKs are called immediately.

Useful for initialization of add-on packages.

**begin_paint**

Enters the enabled ( active paint ) state, and returns the success flag. Once the object is in the enabled state, painting and drawing methods can perform drawing operations on the whole screen.

**begin_paint_info**

Enters the information state, and returns the success flag. The object information state is the same as the enabled state ( see `begin_paint()`), except that painting and drawing methods are not permitted to change the screen.

**close**

Issues a system termination call, resulting in calling the `close` method for all top-level windows. The call can be interrupted by the latter, and effectively canceled. If not canceled stops the application event loop.

**close_help**

Closes the help viewer window.

**end_paint**

Quits the enabled state and returns the application object to the normal state.

**end_paint_info**

Quits the information state and returns the application object to the normal state.

**font_encodings**

Returns an array of encodings represented by strings, that are recognized by the system and available for at least one font. Each system provides different sets of encoding strings; the font encodings are not portable.

**fonts NAME = '', ENCODING = ''**

Returns a hash of font hashes ( see the **Fonts** entry in the *Prima::Drawable* section ) describing fonts of NAME font family and of ENCODING text encoding. If NAME is '' or `undef`, returns one font hash for each of the font families that match the ENCODING string. If ENCODING is '' or `undef`, no encoding match is performed. If ENCODING is not valid ( not present in the `font_encodings` result), it is treated as if it was '' or `undef`.

In the special case when both NAME and ENCODING are '' or `undef`, each font metric hash contains the element `encodings`, which points to an array of the font encodings, available for the fonts of the NAME font family.

**get_active_window**

Returns the object reference to the currently active window, if any, that belongs to the program. If no such window exists, `undef` is returned.

The exact definition of 'active window' is system-dependent, but it is generally believed that an active window is the one that has a keyboard focus on one of its children widgets.

**get_caption_font**

Returns the title font that the system uses to draw top-level window captions. The method can be called with a class string instead of an object instance.

**get_default_cursor_width**

Returns the width of the system cursor in pixels. The method can be called with a class string instead of an object instance.

**get_default_font**

Returns the default system font. The method can be called with a class string instead of an object instance.

**get_default_scrollbar_metrics**

Returns dimensions of the system scrollbars - width of the standard vertical scrollbar and height of the standard horizon scrollbar. The method can be called with a class string instead of an object instance.

**get_dnd_clipboard**

Returns the predefined special clipboard used as a proxy for drag-and-drop interactions.

See also: `Widget/Drag and drop`, `Clipboard/is_dnd`.

**get_default_window_borders BORDER_STYLE = bs::Sizeable**

Returns width and height of standard system window border decorations for one of the `bs::XXX` constants. The method can be called with a class string instead of an object instance.

**get_focused_widget**

Returns object reference to the currently focused widget, if any, that belongs to the program. If no such widget exists, `undef` is returned.

**get_fullscreen_image**

Syntax sugar for grabbing the whole screen as in

```
$::application->get_image( 0, 0, $::application->size)
```

(MacOSX/XQuartz note: get_image() does not grab all screen bits, but `get_fullscreen_image` does if Prima is compiled with the Cocoa library).

**get_hint_widget**

Returns the hint label widget, attached automatically to the Prima::Application object during startup. The widget is of the `::hintClass` class, Prima::HintWidget by default.

**get_image X_OFFSET, Y_OFFSET, WIDTH, HEIGHT**

Returns Prima::Image object with WIDTH and HEIGHT dimensions filled with graphic content of the screen, copied from X_OFFSET and Y_OFFSET coordinates. If WIDTH and HEIGHT extend beyond the screen dimensions, they are adjusted. If the offsets are outside the screen boundaries, or WIDTH and HEIGHT are zero or negative, `undef` is returned.

Note: When running on MacOSX under XQuartz, the latter does not give access to the whole screen, so the function will not be able to grab the top-level menu bar. This problem is addressed in the `get_fullscreen_image` method.

**get_indents**

Returns 4 integers that correspond to extensions of eventual desktop decorations that the windowing system may present on the left, bottom, right, and top edges of the screen. For example, for win32 this reports the size of the part of the screen that the windows taskbar may occupy, if any.

**get_printer**

Returns the printer object attached automatically to the Prima::Application object. The object is an instance of the `::printerClass` class.

**get_message_font**

Returns the font the system uses to draw the message text. The method can be called with a class string instead of an object instance.

**get_modal_window MODALITY_TYPE = mt::Exclusive, TOPMOST = 1**

Returns the modal window that resides on an end of the modality chain. MODALITY_TYPE selects the chain, and can be either `mt::Exclusive` or `mt::Shared`. TOPMOST is a boolean flag selecting the lookup direction: if it is 1, the 'topmost' window is returned, if 0, the 'lowermost' one ( in a simple case when window A is made modal (executed) after modal window B, the A window is the 'topmost' one ).

If the chain is empty `undef` is returned. In case the chain consists of just one window, the TOPMOST value is irrelevant.

**get_monitor_rects**

Returns set of rectangles in the format of (X,Y,WIDTH,HEIGHT) identifying monitor layouts.

**get_scroll_rate**

Returns two integer values of two system-specific scrolling timeouts. The first is the initial timeout that is applied when the user drags the mouse from a scrollable widget ( a text field, for example ), and the widget is about to scroll, but the actual scroll is performed after the timeout has expired. The second value is the repetitive timeout, - if the dragging condition did not change, the scrolling performs automatically after this timeout. The timeout values are in milliseconds.

**get_system_info**

Returns a hash with the following keys containing information about the system:

**apc**

One of the `apc::XXX` constants reporting the platform the program is running on. Currently, the list of the supported platforms is one of these two:

```
apc::Win32
apc::Unix
```

**gui**

One of the `gui::XXX` constants reporting the graphic user interface used in the system:

```
gui::Default
gui::Windows
gui::XLib
gui::GTK
```

**guiDescription**

Description of the graphic user interface returned as an arbitrary string.

**guiLanguage**

The preferred language of the interface returned as an ISO 639 code.

**system**

An arbitrary string representing the operating system software.

**release**

An arbitrary string, contains the OS version information.

**vendor**

The OS vendor string

**architecture**

The machine architecture string

The method can be called with a class string instead of an object instance.

**get_system_value**

Returns the system integer value, associated with one of the `sv::XXX` constants. The constants are:

```
sv::YMenu            - height of menu bar in top-level windows
sv::YTitleBar        - height of title bar in top-level windows
sv::XIcon            - width and height of main icon dimensions,
sv::YIcon              acceptable by the system
sv::XSmallIcon       - width and height of alternate icon dimensions,
sv::YSmallIcon         acceptable by the system
sv::XPointer         - width and height of mouse pointer icon
sv::YPointer           acceptable by the system
sv::XScrollbar       - width of the default vertical scrollbar
sv::YScrollbar       - height of the default horizontal scrollbar
                                                    ( see get_default_scrollbar_
sv::XCursor          - width of the system cursor
                                                    ( see get_default_cursor_wid
sv::AutoScrollFirst  - the initial and the repetitive
sv::AutoScrollNext     scroll timeouts
```

```
                                                    ( see get_scroll_rate() )
sv::InsertMode       - the system insert mode
                                                    ( see insertMode )
sv::XbsNone          - widths and heights of the top-level window
sv::YbsNone            decorations, correspondingly, with borderStyle
sv::XbsSizeable        bs::None, bs::Sizeable, bs::Single, and
sv::YbsSizeable        bs::Dialog.
sv::XbsSingle          ( see get_default_window_borders() )
sv::YbsSingle
sv::XbsDialog
sv::YbsDialog
sv::MousePresent     - 1 if the mouse is present, 0 otherwise
sv::MouseButtons     - number of the mouse buttons
sv::WheelPresent     - 1 if the mouse wheel is present, 0 otherwise
sv::SubmenuDelay     - timeout ( in ms ) before a sub-menu shows on
                                                    an implicit selection
sv::FullDrag         - 1 if the top-level windows are dragged dynamically,
                                                    0 - with marquee mode
sv::DblClickDelay    - mouse double-click timeout in milliseconds
sv::ShapeExtension   - 1 if Prima::Widget::shape functionality is supported,
                                                    0 otherwise
sv::ColorPointer     - 1 if the system accepts color pointer icons.
sv::CanUTF8_Input    - 1 if the system can generate key codes in unicode
sv::CanUTF8_Output   - 1 if the system can output utf8 text
sv::CompositeDisplay - 1 if the system uses double-buffering and alpha composition f
                       0 if it doesn't, -1 if unknown
sv::LayeredWidgets   - 1 if the system supports layering
sv::FixedPointerSize - 0 if the system doesn't support arbitrarily sized pointers an
sv::MenuCheckSize    - width and height of default menu check icon
sv::FriBidi          - 1 if Prima is compiled with libfribidi and full bidi unicode
sv::Antialias        - 1 if the system supports antialiasing and alpha layer for pri
sv::LibThai          - 1 if Prima is compiled with libthai
```

The method can be called with a class string instead of an object instance.

**get_widget_from_handle HANDLE**

HANDLE is an integer value of a toolkit widget handle as used in the underlying GUI level, for example, it is a HWND value on win32. It is usually passed to the program by other IPC means, so that the method can return the associated widget. If no widget is associated with HANDLE, `undef` is returned.

**get_widget_from_point X_OFFSET, Y_OFFSET**

Returns the widget that occupies the screen area under (X_OFFSET,Y_OFFSET) coordinates. If no toolkit widgets are found, `undef` is returned.

**go**

The main event loop. Called by the

```
run Prima;
```

standard code. Returns when the program is about to terminate, if `stop` was called, or if the exception was signaled. In the latter two cases, the loop can be safely restarted.

**lock**

Effectively blocks the graphic output for all widgets. The output can be restored with `unlock()`.

**load_font FONTNAME**

Registers a font resource in the system-specific format. The resource is freed after the program ends.

Notes for win32: To add a font whose information comes from several resource files, point FONTNAME to a string with the file names separated by a | - for example, `abcxxxxx.pfm | abcxxxxx.pfb` .

Notes for unix: available only when Prima is compiled with fontconfig and Xft .

Returns the number of the font resources added.

**open_help TOPIC**

Opens the help viewer window with TOPIC string in the link POD format ( see *perlpod* ) - the string is treated as "manpage/section", where 'manpage' is the file with POD content and 'section' is the topic inside the manpage.

Alternatively can handle the syntax in the form of `file://path|section` where `path` is the file with the pod content and `section` is an optional pod section within the file.

**stop**

Stops the event loop. The loop can be started again.

**sync**

Synchronizes all pending requests where there are any. Is an effective `XSync(false)` on X11, and is a no-op otherwise.

**sys_action CALL**

CALL is an arbitrary string of the system service name and the parameters to it. This functionality is non-portable, and its usage should be avoided. The system services provided are not documented and are subject to change. The actual services can be looked at in the toolkit source code under the *apc_system_action* tag.

**unlock**

Unlocks the graphic output for all widgets, previously locked with `lock()`.

**yield $wait_for_event=0**

An event dispatcher, called from within the event loop. If the event loop can be schematized, then in this code

```
while ( application not closed ) {
        yield
}
```

yield() is the only function called repeatedly inside the loop. The yield(0) call shouldn't be used to organize event loops, but it can be employed to process stacked system events explicitly, to increase the responsiveness of a program, for example, inside a long calculation cycle.

yield(1) though is adapted exactly for external implementation of event loops; it does the same as yield(0), but if there are no events it sleeps until there comes at least one, processes it, and then returns. The return value is 0 if the application doesn't need more event processing, because of shutting down. The corresponding code will be

```
while ( yield(1)) {
    ...
}
```

but in turn, this call cannot be used for increasing UI responsiveness inside tight calculation loops.

The method can be called with a class string instead of an object instance; however, the $::application object must be initialized.

## 3.16 Prima::Printer

Printing services

### Synopsis

```
my $printer = $::application-> get_printer;
print "printing to ", $printer->printer, "...\n";
$p-> options( Orientation => 'Landscape', PaperSize => 'A4');
if ( $p-> begin_doc) {
   $p-> bar( 0, 0, 100, 100);
   print "another page...\n";
   $p-> new_page or die "new_page:$@";
   $p-> ellipse( 100, 100, 200, 200);
   (time % 1) ? # depending on the moon phase, print it or cancel out
         $p-> end_doc :
         $p-> abort_doc;
} else {
   print "failed:$@\n";
}
```

### Description

The *Prima::Printer* class is a descendant of the *Prima::Drawable* class. It provides access to the system printing services, where available. If the system provides no graphics printing, the default PostScript (tm) interface module *Prima::PS::Printer* is used instead.

### Usage

*Prima::Printer* objects are never created directly. During the life of a program, there exists only one instance of a printer object, created automatically by *Prima::Application*. A *Prima::Printer* object is created only when the system provides the graphic printing capabilities, ie drawing and painting procedures on a printer device. If there are no such API, *Prima::Application* creates an instance of the *Prima::PS::Printer* class instead, which emulates a graphic device, and can produce PostScript and PDF output. The difference between the *Prima::Printer* and the *Prima::PS::Printer* class is almost nonexistent for both the user and the programmer unless printer device-specific adjustments are needed.

A printing session is started by calling the `begin_doc()` method which switches the printer object into the painting state. If the session is finished by the `end_doc()` call then the document is duly delivered to the selected printer device. The alternative finishing method, `abort_doc()`, terminates the printing session with no information printed, unless the document is multi-paged and pages are already sent to the printer via the `new_page()` method call.

A printer object ( that means, derived from either *Prima::Printer* or *Prima::PS::Printer* ) provides a mechanism that allows the selection of the printer. The `printers()` method returns an array of hashes, each describing a printer device. The `get_default_printer()` method returns the default printer string identifier. The printer device can be selected by calling the `::printer` property.

The capabilities of the selected printer can be adjusted via the `setup_dialog()` method which invokes the system-provided ( or, in the case of *Prima::PS::Printer*, toolkit-provided ) printer setup dialog so the user can adjust the settings of the printer device. It depends on the system, whether the setup changes only the instance settings, or the default behavior of the printer driver affecting every program.

Some printer capabilities that can be queried include the `::size()` property that reports the dimension of the page, the `::resolution()` property that reports the DPI resolution selected by the printer driver, and the list of available fonts ( by the `fonts()` method ).

A typical code that prints the document looks like this:

```
my $p = $::application-> get_printer;
if ( $p-> begin_doc) {
   ... draw ...
   $p-> end_doc;
} else {
   print "failed:$@\n";
}
```

In addition, the standard package *Prima::Dialog::PrintDialog* can be recommended so the user can select a printer device and adjust its setup interactively.

## API

### Properties

**printer STRING**

Selects the printer device specified by its STRING identifier. Cannot select a device if a printing session is started.

**resolution X, Y**

A read-only property; returns the horizontal and vertical resolutions in DPI currently selected for the printer device. The user can change this setting, if the printer device supports several resolutions, inside the call of the `setup_dialog()` method.

**size WIDTH, HEIGHT**

A read-only property; returns the dimensions of the printer device page. The user can change this setting, if the printer device supports several resolutions or page formats, inside the call of the `setup_dialog()` method.

### Methods

**abort_doc**

Stops the printing session returns the object to the disabled painting state. Since the document can be passed to the system spooler, parts of it could have been sent to a printing device when `abort_doc()` is called, so some information could still have been printed.

**begin_doc DOCUMENT_NAME = ""**

Initiates the printing session and triggers the object into the enabled painting state. The document is assigned the DOCUMENT_NAME string identifier.

Returns the success flag; if failed, `$@` contains the error.

**begin_paint**

Identical to the `begin_doc("")` call.

**begin_paint_info**

Triggers the object into the information painting state. In this state, all graphic functions can be accessed, but no data is printed. Neither the `new_page()` and `abort_doc()` methods work. The information mode is exited via the `end_paint_info()` method.

**end_doc**

Ends the printing session and delivers the document to the printer device. Does not report eventual errors that occurred during the spooling process - the system is expected to take care of such situations.

**end_paint**

Identical to `abort_doc()`.

**end_paint_info**

Quits the information painting mode initiated by `begin_paint_info()` and returns the object to the disabled painting state.

**font_encodings**

Returns an array of the encodings, represented by strings, that are recognized by the system and available in at least one font. Each system provides different sets of encoding strings; the font encodings are not portable.

**fonts NAME = '', ENCODING = ''**

Returns a hash of font hashes ( see the *Prima::Drawable* section, Fonts section ) describing fonts from the NAME font family with the ENCODING encoding. If the NAME is '' or `undef`, returns one font hash for each of the font families that match the ENCODING string. If ENCODING is '' or `undef`, no encoding match is performed. If the ENCODING is not valid ( not present in the `font_encodings` result), it is treated as if it was '' or `undef`.

In the special case, when both NAME and ENCODING are '' or `undef`, each font metric hash contains the element `encodings`, which points to an array of the font encodings, available for the fonts of the NAME font family.

**new_page**

Finalizes the current page and starts a new blank page.

Returns the success flag; if failed, `$@` contains the error.

**options [ OPTION, [ VALUE, [ ... ]]]**

Queries and sets printer-specific setup options, such as orientation, paper size, etc. If called without parameters, returns the list of options the printer supports. If called with one parameter, treats it as the option name and return the corresponding value. Otherwise, treats parameters as a list of key-value pairs, and changes the printer options. Returns the number of the options that were successfully set.

The compatibility between the options and the values used by different OSes is low. The only fully compatible options are `Orientation`[`Portrait`|`Landscape`], `Color`[`Color`|`Monochrome`], `Copies`[`integer`], and `PaperSize`[`A`*`integer`*|`B`*`integer`*|`Executive`|`Folio`|`Ledger`|`Legal`|`Letter`|`Tabloid`]. The other options are OS-dependent. For win32, consult Microsoft manual on the DEVMODE structure the *https:learn.microsoft.comen-uswindowswin32apiwingdins-wingdi-devmodew* entry for Prima's own PostScript printer, consult the *Prima::PS::Printer* section.

**printers**

Returns an array of hashes where each entry describes a printer device. The hash consists of the following entries:

**name**

The printer device's name

**device**

The physical device name, that the printer is connected to

**defaultPrinter**

The boolean flag, is 1 if the printer is default, is 0 otherwise.

**setup_dialog**

Invokes the system-provided printer device setup dialog. In this setup, the user can adjust the capabilities of the printer, such as page setup, resolution, color, etc etc.

**get_default_printer**

Returns the string identifying the default printer device.

**get_handle**

Returns the system handle for the printer object.

## 3.17 Prima::File

Asynchronous stream I/O

### Synopsis

```
use strict;
use Prima qw(Application);

# create pipe and autoflush the writer end
pipe(READ, WRITE) or die "pipe():$!\n";
select WRITE;
$|=1;
select STDOUT;

# create a Prima listener on the reader end
my $read = Prima::File-> new(
    file => \*READ,
    mask => fe::Read,
    onRead => sub {
       $_ = <READ>;
       print "read:$_\n";
    },
);

print WRITE "line\n";
run Prima;
```

### Description

The Prima::File class provides access to the I/O stream notifications that are called when a file handle becomes readable, writable, or if an exception/out-of-band message occurs. Registering file handles to Prima::File objects makes it possible for the stream operations to coexist with the event loop.

### Usage

Prima::File is a descendant of Prima::Component. Objects of the Prima::File class must be bounded to a valid file handle object before the associated events can occur:

```
my $f = Prima::File-> create();
$f-> file( *STDIN);
```

When a file handle bound via the `::file` property becomes readable, writable, or when an exception/out-of-band message is signaled, one of three corresponding events is sent - `Read`, `Write`, or `Exception`. When the file handle is always readable, or always writable, or, on the contrary, some of these events are desired to be blocked, the file event mask can be set via the `::mask` property:

```
$f-> mask( fe::Read | fe::Exception);
```

When the file handle is not needed anymore it is expected to be detached from the object explicitly:

```
$f-> file( undef);
```

However, if the system detects that the file handle is no longer valid, it is automatically detached. It is possible to check if a file handle is still valid by calling the `is_active()` method.

Prima::File events based on the events provided by the `select()` function on unix or on the `WSAEnumNetworkEvents` function on Win32.

## API

### Properties

#### file HANDLE

Selects the file handle to be monitored for the I/O events. If the HANDLE is `undef`, the object is returned to the passive state, and the previously bonded file handle is de-selected.

If the OS reports an error when attaching the file, f ex because there are too many objects to monitor, the file handle is reverted to `undef`. Use that to check for an error.

#### fd INTEGER

Same as file(), but to be used for the file descriptors instead. When this property is used, consequent get-calls to file() will return undef.

If the OS reports an error when attaching the file, f ex because there are too many objects to monitor, the file handle is reverted to `undef`. Use that to check for an error.

#### mask EVENT_MASK

Selects the event mask that is a combination of the `fe::XXX` integer constants, each representing an event:

```
fe::Read
fe::Write
fe::Exception
```

The masked events are effectively excluded from the system file event multiplexing mechanism.

### Methods

#### get_handle

Returns `sprintf("0x%08x", fileno( file ))` string. If `::file` is `undef`, -1 is used instead of the fileno() result.

#### is_active AUTODETACH = 0

Returns the boolean flag indicating if the file handle is valid. If AUTODETACH is 1 and the file handle is not valid `file(undef)` is called.

### Events

#### Read

Called when the file handle becomes readable. The callback procedure is expected to call a non-blocking read() on the file handle.

#### Write

Called when the file handle becomes writable. The callback procedure is expected to call a non-blocking write() on the file handle.

#### Exception

Called when an exception is signaled on the file handle. The exceptions are specific to the handle type and the operating system. For example, a unix socket may signal the `Exception` event when control status data for a pseudoterminal or an out-of-band message arrives.

## OS considerations

### Unix

Prima can monitor max FD_SETSIZE file handles (not Prima::File objects, these can refer to the same file handles just fine). See also `man 2 select`.

### Win32

#### Files

If Prima detects that the handle is neither a socket nor a console, it assumes that it is a regular file. Prima doesn't use any win32 api for checking on regular file handle availability for reading and writing and therefore sends synthetic events without actual correlation on whether the file handle is readable or writable.

#### Pipes

Pipe handles are not implemented and won't work.

#### Sockets

Sockets work natively, however, there's a single catch: according to the MSDN, WSAEventSelect() sets sockets in a non-blocking mode, however, I couldn't confirm that when I was experimenting. If you want to be 100% covered, remember to save and restore the blocking flag in your event handlers.

There can be normally a maximum of 63 sockets (not Prima::File objects, these can refer to the same sockets just fine). Or a maximum of 62 sockets if STDIN is monitored too. See also the *https:learn.microsoft.comen-uswindowswin32apiwinusernf-winusermsgwaitformultipleobjectsex* entry .

#### STDIN

STDIN works fine when it is a console. Use `Prima::sys::win32::ReadConsoleInput` for detailed input parsing. See also the *https:learn.microsoft.comen-uswindowsconsolereadconsoleinputex* entry.

# 4 Widget library

## 4.1 Prima::Buttons

Buttons, checkboxes, radios

### Synopsis

```
use Prima qw(Application Buttons StdBitmap);

my $window = Prima::MainWindow-> create;
Prima::Button-> new(
        owner => $window,
        text  => 'Simple button',
        pack  => {},
);
$window-> insert( 'Prima::SpeedButton' ,
        pack => {},
        image => Prima::StdBitmap::icon(0),
);

run Prima;
```


### Description

Prima::Buttons provides button widgets that include push buttons, check-boxes, and radio buttons. It also provides the `Prima::GroupBox` class used as a container for the checkboxes and radio buttons.

The module provides the following classes:

```
*Prima::AbstractButton
        Prima::Button
                Prima::SpeedButton
        *Prima::Cluster
                Prima::CheckBox
                Prima::Radio
Prima::GroupBox
```

Note: `*` - marked classes are abstract.

## Usage

```
use Prima::Buttons;

my $button = $widget-> insert( 'Prima::Button',
        text => 'Push button',
        onClick => sub { print "hey!\n" },
);
$button-> flat(1);

my $group = $widget-> insert( 'Prima::GroupBox',
        onRadioClick => sub { print $_[1]-> text, "\n"; }
);
$group-> insert( 'Prima::Radio', text => 'Selection 1');
$group-> insert( 'Prima::Radio', text => 'Selection 2', pressed => 1);
$group-> index(0);
```

## Prima::AbstractButton

Prima::AbstractButton realizes the common functionality of buttons. It provides a reaction to mouse and keyboard events and calls the the *Click* entry notification when the user activates the button. The activation can be done by:

- Mouse click
- The spacebar key is pressed
- {`default`} ( see the *default* entry property ) boolean variable is set and the Enter key is pressed. This works even if the button is out of focus.
- {`accel`} character variable is assigned and the corresponding character key is pressed. The {`accel`} variable is extracted automatically from the text string passed to the the *text* entry property. This works even if the button is out of focus.

### Events

#### Check

Abstract callback event.

#### Click

Called whenever the user activates the button.

### Properties

#### hotKey CHAR

A key (defined by CHAR) that the button will react to if pressed if the button has the focus. The combination ALT + Key works always whether the button has the focus or not

#### pressed BOOLEAN

Manages the pressed state of the button

Default value: 0

#### text STRING

The text that is drawn in the button. If STRING contains the ~ ( tilde ) character, the following character is treated as a hotkey, and the character is underlined. If the user presses the corresponding character key then the the *Click* entry event is called. This works even if the button is out of focus.

### Methods

**draw_veil CANVAS, X1, Y1, X2, Y2**

Draws a rectangular veil shape over the CANVAS in given boundaries. This is the default method of drawing the button in the disabled state.

**draw_caption CANVAS, X, Y**

Draws a single line of text stored in the the *text* entry property on the CANVAS at the X, Y coordinates. Underlines an eventual tilde-escaped character and draws the text with dimmed colors if the button is disabled. If the button is focused, draws a dotted rectangle around the text.

**caption_box [ CANVAS = self ]**

Calculates geometrical extensions of the string stored in the the *text* entry property, in pixels. Returns two integers, the width and the height of the string for the font currently selected on the CANVAS.

If CANVAS is undefined, the widget's font is used for the calculations instead.

## Prima::Button

Push button widget, extends the Prima::AbstractButton functionality by allowing an image to be drawn together with text.

### Properties

**autoHeight BOOLEAN**

If 1, the button height is automatically changed as text extensions change.

Default value: 1

**autoRepeat BOOLEAN**

If set, the widget behaves like a keyboard button - after the first the *Click* entry event, a timeout is set, after which if the button is still pressed, the the *Click* entry event is repeatedly fired. Can be useful f ex for emulating scroll-bar arrow buttons.

Default value: 0

**autoShaping BOOLEAN**

If 1, the button `shape` is automatically updated when the button size and/or font are updated, if the current skin can make use of non-rectangular shapes. Generally is unneeded unless the owner of the button has a different back color or features some custom painting.

Default value: 0

See also: *examples/triangle.pl*, *examples/dragdrop.pl*

**autoWidth BOOLEAN**

If 1, the button width is automatically changed as text extensions change.

Default value: 1

**borderWidth INTEGER**

Width of the border around the button.

Default value: depends on the skin

**checkable BOOLEAN**

Selects if the button toggles the the *checked* entry state when the user presses it.

Default value: 0

**checked BOOLEAN**

Selects whether the button is checked or not. Only actual when the the *checkable* entry property is set. See also the *holdGlyph* entry.

Default value: 0

**default BOOLEAN**

Defines if the button should react when the user presses the enter button. If set, the button is drawn with a black border, indicating that it executes the 'default' action. Useful for OK-buttons in dialogs.

Default value: 0

**defaultGlyph INTEGER | IMAGE | METAFILE**

Selects the index of the default sub-image.

Default value: 0

**disabledGlyph INTEGER | IMAGE | METAFILE**

Selects the index of the sub-image for the disabled button state. If `image` does not contain such a sub-image, the `defaultGlyph` sub-image is drawn and is dimmed over using the the *draw_veil* entry method.

Default value: 1

**flat BOOLEAN**

Selects special 'flat' mode, when a button is painted without a border when the mouse pointer is outside the button boundaries. This mode is useful for the toolbar buttons. See also the *hiliteGlyph* entry.

Default value: 0

**glyphs INTEGER**

If a button is to be drawn with an image, it can be passed in the the *image* entry property. If, however, the button must be drawn with several different images, there are no several image-holding properties. Instead, the the *image* entry object can be logically split vertically into several equal sub-images. This allows the image resource to contain all button states in a single image file. The `glyphs` property assigns how many such sub-images the image object contains.

The sub-image indices can be assigned to reflect the different button states. These indices are selected by the following integer properties: the *defaultGlyph* entry, the *hiliteGlyph* entry, the *disabledGlyph* entry, the *pressedGlyph* entry, and the *holdGlyph* entry.

Default value: 1

**hilite**

Read-only property, return 1 if the button is highlighted, 0 otherwise.

**hiliteGlyph INTEGER | IMAGE | METAFILER**

Selects the index of the sub-image for the state when the mouse pointer is hovering over the button. This image is used only when the the *flat* entry property is set. If `image` does not contain such a sub-image, the `defaultGlyph` sub-image is drawn.

Default value: 0

**holdGlyph INTEGE | IMAGE | METAFILER**

Selects the index of the sub-image for the state when the button is the *checked* entry. This image is used only when the the *checkable* entry property is set. If `image` does not contain such a sub-image, the `defaultGlyph` sub-image is drawn.

Default value: 3

**image OBJECT**

If set, the image object is drawn next with the button text, on the top or on the left ( see the the *vertical* entry property ). If the OBJECT contains several sub-images, then the corresponding sub-image is drawn for each button state. See the the *glyphs* entry property.

Can also be a `Prima::Drawable::Metafile` object, however, the `imageScale` factor wouldn't work on it.

Default value: undef

**imageFile FILENAME**

An alternative to image selection that loads an image from the file. During the creation state, if set together with the the *image* entry property, is superseded by the latter.

To allow easy multiframe image access, the FILENAME string is checked if it contains a number after the colon in the string end. Such as, `imageFile('image.gif:3')` loads the fourth frame from `image.gif` .

**imageScale SCALE**

Manages the zoom factor for the the *image* entry.

Default value: 1

**modalResult INTEGER**

Contains a custom integer value, preferably one of `mb::XXX` constants. If a button with non-zero `modalResult` is owned by a currently executing modal window, and is pressed, its `modalResult` value is copied to the `modalResult` property of the owner window, and the latter is closed. This scheme is helpful for the following dialog design:

```
$dialog-> insert( 'Prima::Button', modalResult => mb::OK,
        text => '~Ok', default => 1);
$dialog-> insert( 'Prima::Button', modalResult => mb::Cancel,
        text => 'Cancel);
return if $dialog-> execute != mb::OK.
```

The toolkit defines the following default constants for `modalResult` use:

```
mb::OK or mb::Ok
mb::Cancel
mb::Yes
mb::No
mb::Abort
mb::Retry
mb::Ignore
mb::Help
```

However, any other integer value can be safely used.

Default value: 0

**smoothScaling BOOL**

Tries to paint the image as smoothly as possible. When the system doesn't support ARGB layering, smooth scaling of icons will be restricted to integer-scaling only (i.e. 2x, 3x, etc) because the smoothed color plane will not match pixelated mask plane, and because box-scaling with non-integer zooms looks ugly.

Default value: true

See also: the **ui_scale** entry in the *Prima::Image* section .

**pressedGlyph INTEGER | IMAGE | METAFILE**

Selects the index of the sub-image for the pressed state of the button. If `image` does not contain such a sub-image, the `defaultGlyph` sub-image is drawn.

**transparent BOOLEAN**

See the **transparent** entry in the *Prima::Widget* section. If set, the background is not painted.

**vertical BOOLEAN**

Determines the position of the image next to the text string. If 1, the image is drawn above the text; left to the text if 0. In a special case when the *text* entry is an empty string, the image is centered.

## Prima::SpeedButton

A convenience class, same as the *Prima::Button* section but with default squared shape and text property set to an empty string.

## Prima::Cluster

An abstract class with common functionality of the *Prima::CheckBox* section and the *Prima::RadioButton* section. Reassigns default actions on tab and back-tab keys, so the sibling cluster widgets are not selected. Has `ownerBackColor` property set to 1, to prevent usage of background color from `wc::Button` palette.

### Properties

**auto BOOLEAN**

If set, the button is automatically checked when the button is in focus. This functionality allows the use of arrow keys for navigating the radio buttons without pressing the spacebar key. It also has a drawback, if a radio button gets focused without user intervention, or indirectly, it also gets checked, so that behavior might confuse. The said can be exemplified when an unchecked radio button in a notebook widget becomes active by turning the notebook page.

Although this property is present in the the *Prima::CheckBox* section class, it is not used in there.

### Methods

**check**

Alias to `checked(1)`

**uncheck**

Alias to `checked(0)`

**toggle**

Reverts the `checked` state of the button and returns the new state.

## Prima::Radio

Represents the standard radio button that can be checked or unchecked. When checked, delivers the the *RadioClick* entry event to the owner if the latter provides one.

The button uses the standard toolkit images with `sbmp::RadioXXX` indices when using the classic skin. If the images can not be loaded, the button is drawn with the graphic primitives.

### Events

**Check**

Called when the button was checked.

## Prima::CheckBox

Represents the standard check box button, that can be checked or unchecked.

The button uses the standard toolkit images with `sbmp::CheckBoxXXX` indices when using the classic skin. If the images can not be loaded, the button is drawn with graphic primitives.

## Prima::GroupBox

A container for radio and checkbox buttons (but can contain any widgets).

The widget draws a rectangular box and a title string. Uses the `transparent` property to determine if it needs to paint its background.

The class does not provide a method to calculate the extension of the inner rectangle. However, it can be safely assumed that all offsets except the upper are 5 pixels. The upper offset is dependent on a font and constitutes half of the font height.

### Events

**RadioClick BUTTON**

Called whenever one of the children radio buttons is checked. BUTTON parameter contains the newly checked button.

The default action of the class is that all checked buttons, except BUTTON, are unchecked. Since the flow type of the `RadioClick` event is `nt::PrivateFirst`, the `on_radioclick` method must be directly overloaded to disable this functionality.

### Properties

**border BOOLEAN**

If set (default), draws a border along the widget boundaries

**index INTEGER**

Checks the child radio button with `index`. The indexing is based on the index in the widget list, returned by the `Prima::Widget::widgets` method.

**value BITFIELD**

BITFIELD is an unsigned integer, where each bit corresponds to the `checked` state of a child check-box button. The indexing is based on the index in the widget list, returned by the `Prima::Widget::widgets` method.

## Bugs

Tilde escaping in `text` is not realized, but is planned to. There currently is no way to avoid tilde underscoring.

Radio buttons can get unexpectedly checked when used in notebooks. See the *auto* entry.

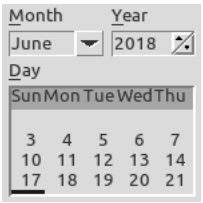
The `Prima::GroupBox::value` parameter is an integer, which size is architecture-dependent. Shifting towards a vector is considered a good idea.

## 4.2 Prima::Calendar

Standard calendar widget

### Synopsis

```
use Prima qw(Calendar Application);
my $cal = Prima::Calendar-> create(
        useLocale => 1,
        size      => [ 150, 150 ],
        onChange  => sub {
                print $_[0]-> date_as_string, "\n";
        },
);
$cal-> date_from_time( localtime );
$cal-> month( 5);
run Prima;
```




### Description

Provides interactive selection for dates between the years 1900 and 2099. The main property, the *date* entry, is a three-integer array of day, month, and year, in the format of perl localtime ( see localtime in *perlfunc* ) - day's range is between 1 and 31, month's 0 to 11, year's from 0 to 199.

### API

#### Events

#### Change

Called when the the *date* entry property is changed.

#### Properties

#### date DAY, MONTH, YEAR

Accepts three integers in the format of `localtime`. DAY can be from 1 to 31, MONTH from 0 to 11, YEAR from 0 to 199.

Default value: today's date.

#### day INTEGER

Selects the day in the month.

#### firstDayOfWeek INTEGER

Selects the first day of the week, an integer between 0 and 6, where 0 is Sunday as the first day, 1 is Monday, etc.

Default value: 0

**month**

Selects the month

**useLocale BOOLEAN**

If 1, the locale-specific names of months and days of week are used. These are read by calling `POSIX::strftime`. If an invocation of the POSIX module fails, the property is automatically assigned to 0.

If 0, the English names of months and days of week are used.

Default value: 1

See also: the *date_as_string* entry

**year**

Selects the year.

### Methods

**can_use_locale**

Returns a boolean value, whether the locale information can be retrieved by calling `strftime` or not.

**month2str MONTH**

Returns the MONTH name according to the the *useLocale* entry value.

**make_months**

Returns an array of the 12 month names according to the the *useLocale* entry value.

**day_of_week DAY, MONTH, YEAR, [ USE_FIRST_DAY_OF_WEEK = 1 ]**

Returns an integer value between 0 and 6, the day of week on DAY, MONTH, YEAR date. If boolean USE_FIRST_DAY_OF_WEEK is set, the value of the `firstDayOfWeek` property is taken into account, so f ex the result of 0 means that this is a Sunday shifted forward by `firstDayOfWeek` days.

The switch from the Julian to the Gregorian calendar is ignored.

**date_as_string [ DAY, MONTH, YEAR ]**

Returns string representation of date on DAY, MONTH, YEAR according to the the *useLocale* entry property value.

**date_from_time SEC, MIN, HOUR, M_DAY, MONTH, YEAR, ...**

Copies the *date* entry from `localtime` or `gmtime` results. This helper method allows the following syntax:

```
$calendar-> date_from_time( localtime( time));
```

## 4.3 Prima::ComboBox

Standard combo box widget

### Synopsis

```
use Prima qw(Application ComboBox);

my $combo = Prima::ComboBox-> new( style => cs::DropDown, items => [ 1 .. 10 ]);
$combo-> style( cs::DropDownList );
print $combo-> text;

run Prima;
```


### Description

Provides a combo box widget that consists of an input line, a list box of possible selections, and an eventual drop-down button. The combo box can be either in the form of a drop-down list that can be shown and hidden or in a form where the selection list is always visible.

The combo box is a grouping widget and contains neither painting nor user input code by itself. All such functionality is delegated to the children widgets: input line, list box, and drop button. `Prima::ComboBox` exports a fixed list of methods and properties from the namespaces of the *Prima::InputLine* section and the *Prima::ListBox* section. It is possible to tweak the `Prima::ComboBox` ( using its the *editClass* entry and the *listClass* entry create-only properties ) so the input line and list box widgets can be instantiated from other classes. The list of exported names is stored in package variables %listProps, %editProps, and %listDynas. These are also described in the the *Exported names* entry section.

The module defines the `cs::` package for the constants used by the the *style* entry property.

### API

#### Properties

**autoHeight BOOLEAN**

If 1, adjusts the height of the widget automatically when its font changes. Only for styles not equal to `cs::Simple`.

Default value: 1

**buttonClass STRING**

Assigns the drop-down button class.

Create-only property.

Default value: `Prima::Widget`

**buttonDelegations ARRAY**

Assigns the list of delegated notifications to the drop-down button.

Create-only property.

**buttonProfile HASH**

Assigns a hash of properties passed to the drop-down button during the creation.

Create-only property.

**caseSensitive BOOLEAN**

Selects whether the user input is case-sensitive or not, when a value is picked from the selection list.

Default value: 0

**editClass STRING**

Assigns the input line class.

Create-only property.

Default value: `Prima::InputLine`

**editProfile HASH**

Assigns a hash of properties passed to the input line during the creation.

Create-only property.

**editDelegations ARRAY**

Assigns the list of delegated notifications to the input line.

Create-only property.

**editHeight INTEGER**

Selects the height of the input line.

**items ARRAY**

Proxy of the list widget's `items` property. See the *Prima::Lists* section for details.

**listClass STRING**

Assigns the list box class.

Create-only property.

Default value: `Prima::ListBox`

**listHeight INTEGER**

Selects the height of the list box widget.

Default value: 100

**listVisible BOOLEAN**

Sets whether the list box is visible or not. Not writable when the `style` is `cs::Simple`.

**listProfile HASH**

Assigns a hash of properties passed to the list box during the creation.

Create-only property.

**listDelegations ARRAY**

Assigns the list of delegated notifications to the list box.

Create-only property.

**literal BOOLEAN**

Selects whether the combo box user input routine should assume that the list box contains literal strings, that can be fetched via `get_item_text` ( see the *Prima::Lists* section ). An example when this property is set to 0 is `Prima::ColorComboBox` from the the *Prima::ComboBox* section package.

Default value: 1

**style INTEGER**

Selects one of three styles of the combo box:

**cs::Simple**

The list box is always visible, but the drop-down button is not.

**cs::DropDown**

The list box is not visible, but the drop-down button is. When the user presses the drop-down button, the list box is shown; when the list-box is defocused, it gets hidden.

**cs::DropDownList**

Same as `cs::DropDown` but the user is restricted in selection: the input line can only accept user input that is present in the list box. If the *literal* entry is set to 1, the auto-completion feature is provided.

**text STRING**

Alias of the input line's `text` property.

### Events

**Change**

Triggered the value is changed.

**List events**

ComboBox forwards `SelectItem` and `DrawItem` events from the list box, and these are executed in the List's context (therefore $self there is not ComboBox, but the ComboBox->List).

See more in the *Prima::Lists* section.

### Exported names

**%editProps**

```
alignment      autoScroll  text         text
charOffset     maxLen      insertMode   firstChar
selection      selStart    selEnd       writeOnly
copy           cut         delete       paste
wordDelimiters readOnly    passwordChar focus
select_all
```

**%listProps**

```
               focusedItem    hScroll
integralHeight items          itemHeight
topItem        vScroll        gridColor
multiColumn    offset
```

**%listDynas**

```
onDrawItem
onSelectItem
```

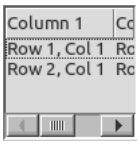
## 4.4 Prima::DetailedList

Multi-column list viewer with controlling header widget

### Synopsis

use Prima::DetailedList;

```
use Prima qw(DetailedList Application);
my $l = Prima::DetailedList->new(
        columns => 2,
        headers => [ 'Column 1', 'Column 2' ],
        items => [
                ['Row 1, Col 1', 'Row 1, Col 2'],
                ['Row 2, Col 1', 'Row 2, Col 2']
        ],
);
$l-> sort(1);
run Prima;
```




### Description

Prima::DetailedList is a descendant of Prima::ListViewer and as such also provides a certain level of abstraction. It overloads the format of the *items* entry in order to support multi-column ( 2D ) cell span. It also inserts the *Prima::Widget::Header* section widget on top of the list so that the user can interactively move, resize, and sort the content of the list. The sorting mechanism is also realized inside the package; it can be activated by the mouse click on a header tab.

Since the class inherits from Prima::ListViewer, some functionality, like 'item search by key', or `get_item_text` method can not operate on 2D lists. Therefore, the the *mainColumn* entry property is introduced, that selects the column representing the textual data.

### API

#### Events

**Sort COLUMN, DIRECTION**

Called inside the the *sort* entry method to facilitate custom sorting algorithms. If the callback procedure is willing to sort by COLUMN index, then it must call `clear_event` to signal that the event flow must stop. The DIRECTION is a boolean flag, specifying whether the sorting must be performed in ascending ( 1 ) or descending ( 0 ) order.

The callback procedure must operate on the internal storage of {`items`}, which is an array of arrays of scalars.

The default action is the literal sorting algorithm where the precedence is arbitrated by the `cmp` operator ( see Equality Operators in *perlop* ) .

### Properties

**aligns ARRAY**

An array of the `ta::` align constants where each defines the column alignment. If an item in the array is undef, it means that the value of the `align` property must be used.

**columns INTEGER**

Manages the number of columns in the *items* entry. If set-called, and the new number is different from the old number, both the *items* entry and the *headers* entry are restructured.

Default value: 0

**headerClass**

Assigns the header class.

Create-only property.

Default value: `Prima::Widget::Header`

**headerProfile HASH**

Assigns a hash of properties passed to the header widget during the creation.

Create-only property.

**headerDelegations ARRAY**

Assigns list of delegated notifications to the header widget.

Create-only property.

**headers ARRAY**

An array of strings passed to the header widget as column titles.

**items ARRAY**

An array of arrays of scalars of any kind. The default behavior, however, assumes that the scalars are strings. The data direction is from left to right and from top to bottom.

**mainColumn INTEGER**

Selects the column responsible for textual representation of all the data. When the user clicks a header tab `mainColumn` is automatically changed to the corresponding column.

Default value: 0

### Methods

**sort [ COLUMN ]**

Sorts items by the COLUMN index in ascending order. If COLUMN is not specified, sorts by the last specified column, or by #0 if it is the first `sort` invocation.

If the COLUMN was specified, and the last specified column equals to COLUMN, the sort direction is reversed.

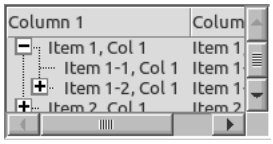
The method does not perform sorting itself, but calls the the *Sort* entry notification, so that the sorting algorithms can be customized.

## 4.5 Prima::DetailedOutline

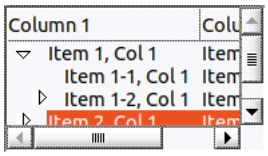
Multi-column outline viewer with controlling header widget.

### Synopsis

```
use Prima qw(DetailedOutline Application);

my $l = Prima::DetailedOutline->new(
      columns => 2,
      headers => [ 'Column 1', 'Column 2' ],
      size    => [200, 100],
      items => [
            [ ['Item 1, Col 1', 'Item 1, Col 2'], [
                  [ ['Item 1-1, Col 1', 'Item 1-1, Col 2'] ],
                  [ ['Item 1-2, Col 1', 'Item 1-2, Col 2'], [
                        [ ['Item 1-2-1, Col 1', 'Item 1-2-1, Col 2'] ],
                  ] ],
            ] ],
            [ ['Item 2, Col 1', 'Item 2, Col 2'], [
                  [ ['Item 2-1, Col 1', 'Item 2-1, Col 2'] ],
            ] ],
      ],
);
$l-> sort(1);
run Prima;
```




```
my $l = Prima::DetailedOutline->new(
   style => 'triangle',
   ...
);
```




### Description

Prima::DetailedOutline combines the functionality of Prima::OutlineViewer and Prima::DetailedList.

### API

This class inherits all the properties, methods, and events of Prima::OutlineViewer (primary ancestor) and Prima::DetailedList (secondary ancestor). One new property `autoRecalc` is introduced, and the `items` property is different enough to warrant the mention.

### Methods

#### items ARRAY

Each item is represented by an arrayref with either one or two elements. The first element is the item data, an arrayref of text strings to display. The second element, if present, is an arrayref of children.

When using the node functionality inherited from Prima::OutlineViewer, the item data (that is, the arrayref of text strings) is the first element of the node.

#### autoRecalc BOOLEAN

If this is set to a true value, the column widths will be automatically recalculated (via `autowidths`) whenever a node is expanded or collapsed.

## 4.6 Prima::DockManager

Advanced dockable widgets

### Description

`Prima::DockManager` contains classes that implement additional functionality in the dockable widgets paradigm.

The module introduces two new dockable widget classes: `Prima::DockManager::Panelbar`, a general-purpose dockable container for variable-sized widgets; and `Prima::DockManager::Toolbar`, a dockable container for fixed-size command widgets, mostly push buttons. The command widgets nested in a toolbar can also be docked.

The `Prima::DockManager` class is application-oriented in a way that a single of it is needed. It is derived from `Prima::Component` and therefore is never visualized. The class instance is stored in the `instance` property in all module classes to serve as a docking hierarchy root. Through the document, the *instance* term means the `Prima::DockManager` class instance.

The module by itself is not enough to make a docking-aware application work effectively. The reader is urged to look at the *examples/dock.pl* example code, which demonstrates the usage and capabilities of the module.

### Prima::DockManager::Toolbar

A toolbar widget class. The toolbar has a dual nature; it can both dock itself and accept dockable widgets. As a dock, toolbars can host command widgets, mostly push buttons.

The toolbar consists of two widgets. The external dockable widget is implemented in `Prima::DockManager::Toolbar`, and the internal dock in `Prima::DockManager::ToolbarDocker` classes.

#### Properties

**autoClose BOOLEAN**

Selects the behavior of the toolbar when all of its command widgets are undocked. If 1 (default), the toolbar is automatically destroyed. If 0 it calls `visible(0)`.

**childDocker WIDGET**

Pointer to the `Prima::DockManager::ToolbarDocker` instance.

See also `Prima::DockManager::ToolbarDocker::parentDocker`.

**instance INSTANCE**

`Prima::DockManager` instance, the docking hierarchy root.

### Prima::DockManager::ToolbarDocker

An internal class, implements the dock widget for command widgets, and a client in a dockable toolbar, a `Prima::LinearDockerShuttle` descendant. When its size is changed due to an eventual rearrangement of its docked widgets, also resizes the toolbar.

#### Properties

**instance INSTANCE**

The `Prima::DockManager` instance, the docking hierarchy root.

**parentDocker WIDGET**

Pointer to a `Prima::DockManager::Toolbar` instance. When in the docked state, the `parentDocker` value is always equal to `owner`.

See also `Prima::DockManager::Toolbar::childDocker`.

### Methods

**get_extent**

Calculates the minimal rectangle that encloses all docked widgets and returns its extensions.

**update_size**

Called when the size is changed to resize the owner widget. If the toolbar is docked, the change might result in a change of its position or docking state.

## Prima::DockManager::Panelbar

The class is derived from `Prima::LinearDockerShuttle`, and is different only in that the `instance` property is introduced, and the external shuttle can be resized interactively.

The class is to be used as a simple host to sizeable widgets. The user can dispose of the panel bar by clicking the close button on the external shuttle.

### Properties

**instance INSTANCE**

The `Prima::DockManager` instance, the docking hierarchy root.

## Prima::DockManager

A binder class, contains a set of functions that groups toolbars, panels, and command widgets together under the docking hierarchy.

The manager serves several purposes. First, it is a command state holder: the command widgets, mostly buttons, usually are in an enabled or disabled state in different life stages of a program. The manager maintains the enabled/disabled state by assigning each command a unique scalar value, or a *CLSID*. The toolbars can be created using a set of command widgets, using these CLSIDs. The same is valid for the panels - although they do not host command widgets, the widgets that they do host can also be created indirectly via CLSID identifier. In addition to CLSIDs, the commands can be grouped by sections. Both CLSID and group descriptor scalars are defined by the programmer.

Second, the `create_manager` method presents the standard launchpad interface that allows the rearranging of normally non-dockable command widgets, by presenting a full set of available commands to the user as icons. Dragging the icons to toolbars, dock widgets, or merely outside the configuration widget automatically creates the corresponding command widget. The notable moment here is that the command widgets are not required to know anything about dragging and docking; any `Prima::Widget` descendant can be used as a command widget.

Third, it helps maintain the toolbars' and panels' visibility when the main window is hidden or restored. The `windowState` method hides or shows the toolbars and panels effectively.

Fourth, it serves as a docking hierarchy root. All docking sessions start their protocol interactions at a `Prima::DockManager` object, which although does not provide docking capabilities itself ( it is a `Prima::Component` descendant ), redirects the docking requests to the children dock widgets.

Finally, it provides several helper methods and notifications and enforces the use of the `fingerprint` property by all dockable widgets. The module defines the following fingerprint `dmfp::XXX` constants:

```
fdmp::Tools      ( 0x0F000) - dock the command widgets
fdmp::Toolbar    ( 0x10000) - dock the toolbars
fdmp::LaunchPad  ( 0x20000) - allows widgets recycling
```

All this functionality is demonstrated in *examples/dock.pl* example.

### Properties

#### commands HASH

A hash of boolean values with keys of CLSID scalars, where if the value is 1, the command is available and is disabled otherwise. Changes to this property are reflected in the visible command widgets, which are enabled or disabled immediately. Also, the `CommandChange` notification is triggered.

#### fingerprint INTEGER

The property is read-only, and always returns `0xFFFFFFFF`, to allow landing to all dockable widgets. In case a finer granulation is needed, the default `fingerprint` values of toolbars and panels can be reset.

#### interactiveDrag BOOLEAN

If 1, the command widgets can be interactively dragged, created, and destroyed. This property is usually operated together with the `create_manager` launchpad widget. If 0, the command widgets cannot be dragged.

Default value: 0

### Methods

#### activate

Brings to front all toolbars and panels. To be used inside a callback code of a main window, that has the toolbars and panels attached to:

```
onActivate => sub { $dock_manager-> activate }
```

#### auto_toolbar_name

Returns a unique name for an automatically created toolbar, like `Toolbar1`, `Toolbar2` etc.

#### commands_enable BOOLEAN, @CLSIDs

Enables or disables commands from CLSIDs array. The changes are reflected in the visible command widgets, which are enabled or disabled immediately. Also, the `CommandChange` notification is triggered.

#### create_manager OWNER, %PROFILE

Inserts two widgets into OWNER with PROFILE parameters: a list box with command section groups, displayed as items, that usually correspond to the predefined toolbar names, and a notebook that displays the command icons. The notebook pages can be interactively selected by the list box navigation.

The icons dragged from the notebook, behave as dockable widgets: they can be landed on a toolbar, or any other dock widget, given it matches the `fingerprint` ( by default `dmfp::LaunchPad|dmfp::Toolbar|dmfp::Tools`). `dmfp::LaunchPad` constant allows the recycling; if a widget is dragged back onto the notebook, it is destroyed.

Returns the two widgets created, the list box and the notebook.

PROFILE recognizes the following keys:

**origin X, Y**

Position where the widgets are to be inserted. The default value is 0,0.

**size X, Y**

Size of the widget insertion area. By default, the widgets occupy all OWNER interiors.

**listboxProfile PROFILE**

Custom parameters passed to the list box.

**dockerProfile PROFILE**

Custom parameters passed to the notebook.

**create_panel CLSID, %PROFILE**

Spawns a dockable panel from a previously registered CLSID. PROFILE recognizes the following keys:

**profile HASH**

A hash of parameters passed to the `new()` method of the panel widget class. Before passing it is merged with the set of parameters registered by `register_panel`. The `profile` hash takes precedence.

**dockerProfile HASH**

Contains extra options passed to the `Prima::DockManager::Panelbar` widget. Before passing it is merged with the set of parameters registered by `register_panel`.

Note: The `dock` key contains a reference to the desired dock widget. If `dock` is set to `undef`, the panel is created in the non-docked state.

Example:

```
$dock_manager-> create_panel( $CLSID,
        dockerProfile => { dock => $main_window }},
        profile       => { backColor => cl::Green });
```

**create_tool OWNER, CLSID, X1, Y1, X2, Y2**

Inserts a command widget, previously registered with a CLSID by `register_tool`, into OWNER widget at X1 - Y2 coordinates. For automatic maintenance of enabled/disabled states of command widgets, OWNER is expected to be a toolbar. If it is not, the maintenance must be performed separately, for example, by reacting to the `CommandChange` event.

**create_toolbar %PROFILE**

Creates a new toolbar of the `Prima::DockManager::Toolbar` class. The following PROFILE options are recognized:

**autoClose BOOLEAN**

Manages the `autoClose` property of the toolbar.

The default value is 1 if the `name` option is set, and 0 otherwise.

**dock DOCK**

Contains a reference to the desired DOCK widget. If `undef`, the toolbar is created in the non-docked state.

**dockerProfile HASH**

Parameters passed to `Prima::DockManager::Toolbar` as creation properties.

Note: The `dock` key contains a reference to the desired dock widget. If `dock` is set to `undef`, the panel is created in the non-docked state.

**rect X1, Y1, X2, Y2**

Manages geometry of the `Prima::DockManager::ToolbarDocker` instance in the dock widget ( if docked ) or the screen ( if non-docked ) coordinates.

**toolbarProfile HASH**

Parameters passed to `Prima::DockManager::ToolbarDocker` as properties.

**vertical BOOLEAN**

Sets the `vertical` property of the toolbar.

**visible BOOLEAN**

Selects the visibility state of the toolbar.

**get_class CLSID**

Returns a class record hash, registered under a CLSID, or `undef` if the class is not registered. The hash format contains the following keys:

**class STRING**

Widget class

**profile HASH**

Creation parameters passed to `new` when the corresponding widget is instantiated.

**tool BOOLEAN**

If 1, the class belongs to a control widget. If 0, the class represents a panel client widget.

**lastUsedDock DOCK**

Saved value of the last used dock widget

**lastUsedRect X1, Y1, X2, Y2**

Saved coordinates of the widget

**panel_by_id CLSID**

Returns reference to the panel widget represented by CLSID scalar, or `undef` if none is found.

**panel_menuitems CALLBACK**

A helper function; maps all panel names into a structure, ready to feed into the `Prima::AbstractMenu::items` property ( see the *Prima::Menu* section ). The action member of the menu item record is set to the CALLBACK scalar.

**panel_visible CLSID, BOOLEAN**

Sets the visibility of a panel referred to by the CLSID scalar. If VISIBLE is 0, the panel is destroyed; if 1, a new panel instance is created.

**panels**

Returns all visible panel widgets in an array.

**predefined_panels CLSID, DOCK, [ CLSID, DOCK, ... ]**

Accepts pairs of scalars, where each first item is a panel CLSID and the second is the default dock widget. Checks for the panel visibility and creates the panels that are not visible.

The method is useful in a program startup, when some panels have to be visible from the beginning.

**predefined_toolbars @PROFILES**

Accepts an array of hashes where each array item describes a toolbar and a default set of command widgets. Checks for the toolbar visibility and creates the toolbars that are not visible.

The method recognizes the following PROFILES options:

**dock DOCK**

The default dock widget.

**list ARRAY**

An array of CLSIDs corresponding to the command widgets to be inserted into the toolbar.

**name STRING**

Selects the toolbar name.

**origin X, Y**

Selects the toolbar position relative to the dock ( if `dock` is specified ) or to the screen ( if `dock` is not specified ).

The method is useful in program startup, when some panels have to be visible from the beginning.

**register_panel CLSID, PROFILE**

Registers a panel client class and set of parameters to be associated with a CLSID scalar. PROFILE must contain the following keys:

**class STRING**

Client widget class

**text STRING**

A string of text displayed in the panel title bar

**dockerProfile HASH**

A hash of parameters passed to `Prima::DockManager::Panelbar`.

**profile**

A hash of parameters passed to the client widget.

**register_tool CLSID, PROFILE**

Registers a control widget class and set of parameters to be associated with a CLSID scalar. PROFILE must contain the following keys:

**class STRING**

Client widget class

**profile HASH**

A hash of parameters passed to the control widget.

**toolbar_by_name NAME**

Returns a reference to the toolbar of NAME, or `undef` if none is found.

**toolbar_menuitems CALLBACK**

A helper function; maps all toolbar names into a structure, ready to feed into the `Prima::AbstractMenu::items` property ( see the *Prima::Menu* section ). The action member of the menu item record is set to the CALLBACK scalar.

**toolbar_visible TOOLBAR, BOOLEAN**

Sets the visibility of a TOOLBAR. If VISIBLE is 0, the toolbar is hidden; if 1, it is shown.

**toolbars**

Returns all toolbar widgets in an array.

**windowState INTEGER**

Mimics interface of `Prima::Window::windowState`, and maintains visibility of toolbars and panels. If the parameter is `ws::Minimized`, the toolbars and panels are hidden. On any other parameter, these are shown.

To be used inside a callback code of a main window, that has the toolbars and panels attached to:

```
onWindowState => sub { $dock_manager-> windowState( $_[1] ) }
```

**Events**

**Command CLSID**

A generic event triggered by a command widget when the user activates it. It can also be called by other means.

CLSID is the widget identifier.

**CommandChange**

Called when the `commands` property changes or the `commands_enable` method is invoked.

**PanelChange**

Triggered when a panel is created or destroyed by the user.

**ToolbarChange**

Triggered when a toolbar is created, shown, gets hidden, or destroyed by the user.

## Prima::DockManager::S::SpeedButton

The package simplifies the creation of `Prima::SpeedButton` command widgets.

**Methods**

**class IMAGE, CLSID, %PROFILE**

Builds a hash with parameters, ready to feed to `Prima::DockManager::register_tool` for registering a combination of the `Prima::SpeedButton` class and the PROFILE parameters.

IMAGE is the path to an image file, loaded and stored in the registration hash. IMAGE provides an extended syntax for selecting the frame index if the image file is multiframed: the frame index is appended to the path name with the `:` character prefix.

CLSID scalar is not used but is returned so the method result can directly be passed into the `register_tool` method.

Returns two scalars: CLSID and the registration hash.

Example:

```
$dock_manager-> register_tool(
        Prima::DockManager::S::SpeedButton::class( "myicon.gif:2",
        q(CLSID::Logo), hint => 'Logo image' ));
```

## 4.7 Prima::Docks

Dockable widgets

### Description

The module contains a set of classes and an implementation of the dockable widgets interface. The interface assumes two parties, the dockable widget, and the dock widget; the generic methods for the dock widget class are contained in the `Prima::AbstractDocker::Interface` package.

### Usage

A dockable widget is required to take particular steps before it may land on a dock widget. It needs to talk to the dock and find out if it is allowed to land, or if the dock contains children dock widgets that might suit better for the docking. If there's more than one dock widget in the program, the dockable widget can select between the targets; this is especially actual when a dockable widget is dragged by the mouse and the landing arbitration is based on geometrical distance.

The interface implies that there exists at least one tree-like hierarchy of dock widgets, linked up to a root dock widget. The hierarchy is not required to follow parent-child relationships although this is the default behavior. All dockable widgets are expected to know explicitly what hierarchy tree they wish to dock to. `Prima::InternalDockerShuttle` introduces the `dockingRoot` property for this purpose.

The conversation between parties starts when a dockable widget calls the `open_session` method of the dock. The dockable widget passes a set of parameters signaling if the widget is ready to change its size in case the dock widget requires so, and how. The `open_session` method can either refuse or accept the widget. In case of the positive answer from `open_session`, the dockable widget calls the `query` method, which either returns a new rectangle or another dock widget. In the latter case, the caller can enumerate all available dock widgets by repetitive calls to the `next_docker` method. The session is closed by a `close_session` call; after that, the widget is allowed to land by setting its `owner` to the dock widget, the `rect` property to the negotiated position and size, and finally calling the `dock` method.

The `open_session`/`close_session` brackets cache all necessary calculations once, making the `query` call as light as possible. This design allows a dockable widget when dragged, to repeatedly ask all reachable docks in an optimized way. The docking sessions are kept open until the drag session is finished.

The conversation can be schematized in the following code:

```
my $dock = $self-> dockingRoot;
my $session_id = $dock-> open_session({ self => $self });
return unless $session_id;
my @result = $dock-> query( $session_id, $self-> rect );
if ( 4 == scalar @result) {       # new rectangle is returned
        if ( ..... is new rectangle acceptable ? ... ) {
                $dock-> close_session( $session_id);
                $dock-> dock( $self);
                return;
        }
} elsif ( 1 == scalar @result) {  # another dock returned
        my $next = $result[0];
        while ( $next) {
                if ( ... is new docker acceptable? ....) {
                        $dock-> close_session( $session_id);
                        $next-> dock( $self);
```

```
                        return;
                    }
                    $next = $dock-> next_docker( $session_id, $self-> origin );
                }
        }
        $dock-> close_session( $session_id);
```

Since even the simplified code is quite cumbersome, direct calls to `open_session` are rare. Instead, `Prima::InternalDockerShuttle` implements the `find_docking` method which performs the arbitration automatically and returns the appropriate dock widget.

`Prima::InternalDockerShuttle` is the class that implements the dockable widget functionality. It also provides a top-level window-like wrapper widget for the dockable widget that hosts the widget automatically if it is not docked. By default, `Prima::ExternalDockerShuttle` is used as the wrapper widget class.

It is not required, however, to use either `Prima::InternalDockerShuttle` or `Prima::AbstractDocker::Interface` to implement a dockable widget; the only requirement is to respect the `open_session`/`close_session` protocol.

The full hierarchy of widgets participating in the mechanism is as follows:

```
Prima::AbstractDocker::Interface
        Prima::SimpleWidgetDocker
                Prima::ClientWidgetDocker
        Prima::LinearWidgetDocker
                Prima::SingleLinearWidgetDocker
        Prima::FourPartDocker

Prima::InternalDockerShuttle
        Prima::LinearDockerShuttle

Prima::ExternalDockerShuttle
```

All docker widget classes are derived from `Prima::AbstractDocker::Interface`. Depending on the specialization, they employ more or less sophisticated schemes for arranging dockable widgets inside themselves. The most complicated scheme is implemented in `Prima::LinearWidgetDocker`; it does not allow children to overlap, can rearrange the children, and resize itself when a widget is docked or undocked.

The package provides only basic functionality. Module `Prima::DockManager` provides common dockable controls, - toolbars, panels, speed buttons, etc. based on the `Prima::Docks` module. See the *Prima::DockManager* section.

## Prima::AbstractDocker::Interface

Implements generic functionality of a docket widget. The class is not derived from `Prima::Widget`; is used as a secondary ascendant class for the dock widget classes.

### Properties

Since the class is not a `Prima::Object` descendant, it provides only run-time implementation of its properties. It is up to the descendant object whether the properties are recognized during the creation stage or not.

#### fingerprint INTEGER

A custom bit mask used by docking widgets to reject inappropriate dock widgets at an early stage. The `fingerprint` property is not a part of the protocol and is not required to be present in the implementation of a dockable widget.

Default value: `0x0000FFFF`

**dockup DOCK_WIDGET**

Selects the upper link in the dock widgets hierarchy tree. The upper link is required to be a dock widget but is not required to be a direct or an indirect parent. In this case, however, the maintenance of the link must be implemented separately, for example:

```
$self-> dockup( $upper_dock_not_parent );

$upper_dock_not_parent-> add_notification( 'Destroy', sub {
        return unless $_[0] == $self-> dockup;
        undef $self-> {dockup_event_id};
        $self-> dockup( undef );
}, $self);

$self-> {destroy_id} = $self-> add_notification( 'Destroy', sub {
        $self-> dockup( undef );
} unless $self-> {destroy_id};
```

**Methods**

**add_subdocker SUBDOCK**

Appends SUBDOCK to the list of children docker widgets. The items of the list are returned by the `next_docker` method.

**check_session SESSION**

A debugging procedure. Checks SESSION hash, and warns if its members are invalid or incomplete. Returns 1 if no fatal errors were encountered; 0 otherwise.

**close_session SESSION**

Closes docking SESSION and frees the associated resources.

**dock WIDGET**

Called after WIDGET successfully finished negotiations with the dock widget and changed its `owner` property. The method adapts the dock widget layout and lists the WIDGET into the list of docked widgets. The method does not change the `owner` property of the WIDGET.

The method must not be called directly.

**dock_bunch @WIDGETS**

Effectively docks set of WIDGETS by updating internal structures and calling `rearrange`.

**docklings**

Returns an array of docked widgets

**next_docker SESSION, [ X, Y ]**

Enumerates children docker widgets inside the SESSION; returns one docker widget at a time. After the last widget returns `undef`.

The enumeration pointer is reset by the `query` call.

X and Y are the coordinates of the point of interest.

**open_session PROFILE**

Opens a new docking session with parameters stored in the PROFILE hash. Returns a session ID scalar in case of success, or `undef` otherwise. The following keys must be set in PROFILE:

**position ARRAY**

Contains two integer coordinates of the desired position of a widget in (X,Y) format in the screen coordinate system.

**self WIDGET**

The widget that is about to dock.

**sizeable ARRAY**

Contains two boolean flags, representing if the widget can be resized to an arbitrary size, horizontally and vertically. The arbitrary resize option is used as a last resort if the `sizes` key does not contain the desired size.

**sizeMin ARRAY**

Two integers; the minimal size that the widget can accept.

**sizes ARRAY**

Contains an array of points in the (X,Y) format; each point represents an acceptable widget size. If both of the `sizeable` flags are set to 0 and none of the `sizes` can be accepted by the dock widget, `open_session` fails.

**query SESSION [ X1, Y1, X2, Y2 ]**

Checks if a dockable widget can be landed on the dock. If it can, returns a rectangle that the widget must be set to. If coordinates ( X1 .. Y2 ) are specified, returns the rectangle closest to these. If `sizes` or `sizeable` keys of the `open_session` profile were set, the returned size might be different from the current docking widget size.

Once the caller finds the result appropriate, it is allowed to reparent under the dock; after that, it must change its origin and size correspondingly to the result, and then call `dock`.

If the dock cannot accept the widget but contains children dock widgets, returns the first child widget. The caller can use subsequent calls to `next_docker` to enumerate all the children docks. A call to `query` resets the internal enumeration pointer.

If the widget may not be landed, an empty array is returned.

**rearrange**

Effectively re-docks all the docked widgets. The effect is as same as of

```
$self-> redock_widget($_) for $self-> docklings;
```

but usually `rearrange` is faster.

**redock_widget WIDGET**

Effectively re-docks the docked WIDGET. If WIDGET has a `redock` method in its namespace, it is called instead.

**remove_subdocker SUBDOCK**

Removes SUBDOCK from the list of children docker widgets. See also the *add_subdocker* entry.

**replace FROM, TO**

Assigns the widget TO the same owner and size as FROM. The FROM widget must be a docked widget.

**undock WIDGET**

Removes WIDGET from the list of docked widgets. The layout of the dock widget can be changed after the execution of this method. The method does not change the `owner` property of WIDGET.

The method must not be called directly.

## Prima::SimpleWidgetDocker

A simple dock widget; accepts any widget that geometrically fits into it. Allows overlapping of the docked widgets.

## Prima::ClientWidgetDocker

A simple dock widget; accepts any widget that can cover all dock's interior.

## Prima::LinearWidgetDocker

A toolbar-like docking widget class. The implementation does not allow tiling but can reshape the dock widget and rearrange the docked widgets if necessary.

`Prima::LinearWidgetDocker` is orientation-dependent; its main axis, managed by the `vertical` property, aligns the docked widgets in 'lines', which in turn are aligned by the opposite axis ( 'major' and 'minor' terms are used in the code for the axes ).

### Properties

**growable INTEGER**

A combination of the `grow::XXX` constants that describes how the dock widget can be resized. The constants are divided into two sets, direct and indirect, or, `vertical` property independent and dependent.

The first set contains explicitly named constants:

```
grow::Left       grow::ForwardLeft       grow::BackLeft
grow::Down       grow::ForwardDown       grow::BackDown
grow::Right      grow::ForwardRight      grow::BackRight
grow::Up         grow::ForwardUp         grow::BackUp
```

that select if the widget can grow in the direction shown. These do not change meaning when `vertical` changes, though they do change the dock widget behavior. The second set does not affect dock widget behavior when `vertical` changes, however, the names are not that illustrative:

```
grow::MajorLess  grow::ForwardMajorLess  grow::BackMajorLess
grow::MajorMore  grow::ForwardMajorMore  grow::BackMajorMore
grow::MinorLess  grow::ForwardMinorLess  grow::BackMinorLess
grow::MinorMore  grow::ForwardMinorMore  grow::BackMinorMore
```

The `Forward` and `Back` prefixes select if the dock widget can be respectively expanded or shrunk in the given direction. `Less` and `More` are equivalent to `Left` and `Right` when `vertical` is 0, and to `Up` and `Down` otherwise.

The use of constants from the second set is preferred.

Default value: 0

**hasPocket BOOLEAN**

A boolean flag, affects the possibility of a docked widget to reside outside the dock widget inferior. If 1, a docked widget is allowed to stay docked ( or dock into a position ) further on the major axis ( to the right when `vertical` is 0, up otherwise ) as if there's a 'pocket'. If 0, a widget is neither allowed to dock outside the inferior nor is allowed to stay docked ( and is undocked automatically ) when the dock widget shrinks so that the docked widget cannot stay in the dock boundaries.

Default value: 1

**vertical BOOLEAN**

Selects the major axis of the dock widget. If 1, it is vertical, horizontal otherwise.

Default value: 0

### Events

**Dock**

Called when the `dock` method is successfully finished.

**DockError WIDGET**

Called when the `dock` method is unsuccessfully finished. This only happens if WIDGET does not follow the docking protocol, and inserts itself into a non-approved area.

**Undock**

Called when `undock` is finished.

## Prima::SingleLinearWidgetDocker

Descendant of `Prima::LinearWidgetDocker`. In addition to the constraints introduced by the ascendant class, `Prima::SingleLinearWidgetDocker` allows only one row ( or column, depending on the `vertical` property value ) of docked widgets.

## Prima::FourPartDocker

Implementation of a docking widget that hosts four children docker widgets on its sides and one in the center. All of the children docks can grow and shrink automatically so that the whole setup has an effect as if the dock borders are dynamic.

### Properties

**indents ARRAY**

Contains four integers specifying the breadth of offset for each side. The first integer is the width of the left side, the second - the height of the bottom side, the third is the width of the right side, and the fourth - height of the top side.

**dockerClassLeft STRING**

Assigns the class of the left-side dock window.

Default value: `Prima::LinearWidgetDocker`. Create-only property.

**dockerClassRight STRING**

Assigns the class of the right-side dock window.

Default value: `Prima::LinearWidgetDocker`. Create-only property.

**dockerClassTop STRING**

Assigns the class of the top-side dock window.

Default value: `Prima::LinearWidgetDocker`. Create-only property.

**dockerClassBottom STRING**

Assigns the class of the bottom-side dock window.

Default value: `Prima::LinearWidgetDocker`. Create-only property.

**dockerClassClient STRING**

Assigns the class of the center dock window.

Default value: `Prima::ClientWidgetDocker`. Create-only property.

**dockerProfileLeft HASH**

Assigns a hash of properties, passed to the left-side dock widget during the creation.

Create-only property.

**dockerProfileRight HASH**

Assigns a hash of properties, passed to the right-side dock widget during the creation.

Create-only property.

**dockerProfileTop HASH**

Assigns a hash of properties, passed to the top-side dock widget during the creation.

Create-only property.

**dockerProfileBottom HASH**

Assigns a hash of properties, passed to the bottom-side dock widget during the creation.

Create-only property.

**dockerProfileClient HASH**

Assigns a hash of properties, passed to the center dock widget during the creation.

Create-only property.

**dockerDelegationsLeft ARRAY**

Assigns delegated notifications of the left-side dock.

Create-only property.

**dockerDelegationsRight ARRAY**

Assigns delegated notifications of the right-side dock.

Create-only property.

**dockerDelegationsTop ARRAY**

Assigns delegated notifications of the top-side dock.

Create-only property.

**dockerDelegationsBottom ARRAY**

Assigns delegated notifications of the bottom-side dock.

Create-only property.

**dockerDelegationsClient ARRAY**

Assigns delegated notifications of the bottom-side dock.

Create-only property.

**dockerCommonProfile HASH**

Assigns a hash of properties, passed to all the five dock widgets during the creation.

Create-only property.

## Prima::InternalDockerShuttle

The class provides a container, or a 'shuttle', for a client widget, while is docked to a `Prima::AbstractDocker::Interface` descendant instance. The functionality includes communicating with dock widgets, the user interface for dragging and interactive dock selection, and a client widget container for the non-docked state. The latter is implemented by reparenting the client widget to an external shuttle widget, selected by the `externalDockerClass` property. Both user interfaces for the docked and the non-docked shuttle states are minimal.

The class implements dockable widget functionality, served by `Prima::AbstractDocker::Interface`, and is derived from `Prima::Widget`.

See also: the *Prima::ExternalDockerShuttle* section.

### Properties

**client WIDGET**

Provides access to the client widget, which always resides either in the internal or the external shuttle. By default, there is no client, and any widget capable of changing its parent can be set as one. After a widget is assigned as a client, its `owner` and `clipOwner` properties must not be used.

Run-time only property.

**dock WIDGET**

Selects the dock widget that the shuttle is landed on. If `undef`, the shuttle is in the non-docked state.

Default value: `undef`

**dockingRoot WIDGET**

Selects the root of the dock widgets hierarchy. If `undef`, the shuttle can only exist in the non-docked state.

Default value: `undef`

See the *Usage* entry for reference.

**externalDockerClass STRING**

Assigns the class of the external shuttle widget.

Default value: `Prima::ExternalDockerShuttle`

**externalDockerModule STRING**

Assigns the module that contains the external shuttle widget class.

Default value: `Prima::MDI` ( `Prima::ExternalDockerShuttle` is derived from `Prima::MDI` ).

**externalDockerProfile HASH**

Assigns a hash of properties, passed to the external shuttle widget during the creation.

**fingerprint INTEGER**

A custom bit mask used to reject inappropriate dock widgets at an early stage.

Default value: `0x0000FFFF`

**indents ARRAY**

Contains four integers, specifying the breadth of offset in pixels for each widget side in the docked state.

Default value: `5,5,5,5`.

**snapDistance INTEGER**

A maximum offset, in pixels, between the actual shuttle coordinates and the coordinates proposed by the dock widget, where the shuttle is allowed to land. In other words, it is the distance between the dock and the shuttle when the latter 'snaps' to the dock during the dragging session.

Default value: 10

**x_sizeable BOOLEAN**

Selects whether the shuttle can change its width in case the dock widget suggests so.

Default value: 0

**y_sizeable BOOLEAN**

Selects whether the shuttle can change its height in case the dock widget suggests so.

Default value: 0

### Methods

**client2frame X1, Y1, X2, Y2**

Returns the rectangle that the shuttle would occupy if its client rectangle is assigned to X1, Y1, X2, Y2 .

**dock_back**

Docks to the recent dock widget, if it is still available.

**drag STATE, RECT, ANCHOR_X, ANCHOR_Y**

Initiates or aborts the dragging session, depending on the STATE boolean flag.

If it is 1, RECT is an array with the coordinates of the shuttle rectangle before the session has started; ANCHOR_X and ANCHOR_Y are coordinates of the aperture point where the mouse event occurred that has initiated the session. Depending on how the drag session ended, the shuttle can be relocated to another dock, undocked, or left intact. Also, `Dock`, `Undock`, or `FailDock` notifications can be triggered.

If the STATE is 0, RECT, ANCHOR_X ,and ANCHOR_Y parameters are not used.

**find_docking DOCK, [ POSITION ]**

Opens a session with DOCK, unless it is already opened, and negotiates about the possibility of landing ( at the POSITION if this parameter is present ).

`find_docking` caches the dock widget sessions and provides a possibility to select different parameters passed to `open_session` for different dock widgets. To achieve this, the `GetCaps` request notification is triggered, which is expected to fill the parameters. The default action sets the `sizeable` option according to the `x_sizeable` and `y_sizeable` properties.

In case an appropriate landing area is found, the `Landing` notification is triggered with the proposed dock widget and the target rectangle. The area can be rejected at this stage if `Landing` returns a negative answer.

On success, returns a dock widget found and the target rectangle; the widget is not docked though. On failure returns an empty array.

This method is used by the mouse dragging routine to provide visual feedback to the user, to indicate that a shuttle may or may not land in a particular area.

**frame2client X1, Y1, X2, Y2**

Returns the rectangle that the client would occupy if the shuttle rectangle is assigned to X1, Y1, X2, Y2 .

**redock**

Undocks from the dock widget and immediately tries to land back. If not docked, does not do anything.

### Events

**Dock**

Called when the shuttle was docked.

**EDSClose**

Triggered when the user presses the close button or otherwise activates the `close` function of the EDS ( external docker shuttle ) pseudo-window. To cancel the window closing `clear_event` must be called inside the event handler.

**FailDock X, Y**

Called after the dragging session in the non-docked stage was finished but did not result in docking. X and Y are the coordinates of the new external shuttle position.

**GetCaps DOCK, PROFILE**

Called before the shuttle opens a docking session with the DOCK widget. PROFILE is a hash reference, which is to be filled inside the event handler. After that PROFILE is passed to an `open_session` call.

The default action sets the `sizeable` option according to the `x_sizeable` and `y_sizeable` properties.

**Landing DOCK, X1, Y1, X2, Y2**

Called inside the docking session, after an appropriate dock widget is selected and the landing area is defined as X1, Y1, X2, Y2. To reject the landing on either DOCK or area, `clear_event` must be called.

**Undock**

Called when the shuttle is switched to the non-docked state.

## Prima::ExternalDockerShuttle

A shuttle class, hosts a client of the `Prima::InternalDockerShuttle` widget when it is in the non-docked state. The widget is a pseudo-window with some minimal decorations that can be moved, resized ( this feature is not on by default though ), and closed.

`Prima::ExternalDockerShuttle` is inherited from the `Prima::MDI` class, and its window-emulating functionality is a subset of its ascendant. See also the *Prima::MDI* section.

### Properties

**shuttle WIDGET**

Contains the reference to the dockable WIDGET

## Prima::LinearDockerShuttle

A simple descendant of `Prima::InternalDockerShuttle`, used for toolbars. Introduces orientation and draws a tiny header along the minor shuttle axis.

### Properties

**headerBreadth INTEGER**

The breadth of the header in pixels.

Default value: 8

**indent INTEGER**

A wrapper to the `indents` property; besides the space for the header, all indents are assigned to the `indent` property value.

**vertical BOOLEAN**

If 1, the shuttle is drawn as a vertical bar. If 0, the shuttle is drawn as a horizontal bar.

Default value: 0

## 4.8 Prima::Edit

Standard text editor

### Synopsis

```
use Prima qw(Edit Application);
my $e = Prima::Edit->new(
        text         => 'Hello $world',
        syntaxHilite => 1,
);
run Prima;
```


### Description

The class provides text editing capabilities, three types of selection, text wrapping, syntax highlighting, auto indenting, undo and redo function, and search and replace methods.

The module declares the `bt::` package that contains integer constants for the selection of block type, used by the the *blockType* entry property.

### Usage

The text is stored line-wise in the {`lines`} array; to access it use the the *get_line* entry method.

All keyboard events except the character input and tab keys are processed by the accelerator table ( see the *Prima::Menu* section ). The default `accelItems` table defines names, keyboard combinations, and the corresponding actions to the class functions. The class does not provide functionality to change these mappings. To do so, consult the **Prima::AccelTable** entry in the *Prima::Menu* section.

#### Coordinates

The class addresses the text space by (X,Y)-coordinates, where X is the visual cluster offset and Y is the line number. The addressing coordinate system can be *visual*, *physical*, or *logical*. See below.

Cluster shaping and word wrapping can play a role here. Consider f.ex. a text string "offset is zero", that for the sake of the example can wrapped by width and displayed as two lines, "offset" and "is zero". Here, the font substitutes "ff" text with a single ligature glyph. Here, for example, coord("f") will be (0,1) in all coordinates, but coord("z") is not:

#### Physical

The X coordinate is a character offset from character line number Y. These coordinates are identical with and without the `wordWrap` flag. This coordinate is used for direct text manipulation.

Example: coord("z") is (0,10);

#### Visual

The X coordinate is a glyph cluster offset from the character line number Y. These coordinates are identical with and without the `wordWrap` flag. This coordinate is used for cursor and selection API.

Example: coord("z") is (0,9);

**Logical**

The Y coordinate is the wrapped line index. The `chunkMap` internal array contains addresses in the logical coordinates. The X coordinate is a glyph cluster offset from the line start. This coordinate is used mostly internally.

To access the text chunk-wise, use the the *get_chunk* entry method.

Example: coord("z") is (1,3);

## API

### Events

**ParseSyntax TEXT, RESULT_ARRAY_REF**

Called when syntax highlighting is required - TEXT is a string to be parsed, and the parsing results to be stored in RESULT_ARRAY_REF, which is a reference to an array of integer pairs, each representing a single-colored text chunk. The first integer in the pairs is the length of a chunk, the second - color value ( `cl::XXX` constants ).

### Properties

**autoIndent BOOLEAN**

Turns the auto indenting on or off

Default value: 1

**blockType INTEGER**

Defines the type of selection block. Can be one of the following constants:

**bt::CUA**

Normal block, where the first and the last line of the selection can be partial, and the lines between occupy the whole line. CUA stands for 'common user access'.

Default keys: Shift + arrow keys

See also: the *cursor_shift_key* entry

**bt::Vertical**

Rectangular block, where all selected lines are of the same offsets and lengths.

Default key: Alt+B

See also: the *mark_vertical* entry

**bt::Horizontal**

Rectangular block, where the selection occupies the whole line.

Default key: Alt+L

See also: the *mark_horizontal* entry

**cursor X, Y**

Selects the visual position of the cursor

**cursorX X**

Selects the visual horizontal position of the cursor

**cursorY Y**

Selects the visual vertical position of the cursor

**cursorWrap BOOLEAN**

Selects cursor behavior when moved horizontally outside the line. If 0, the cursor is not moved. If 1, the cursor moved to the adjacent line.

See also: the *cursor_left* entry, the *cursor_right* entry, the *word_left* entry, the *word_right* entry.

**insertMode BOOLEAN**

Set the typing mode - if 1, the typed text is inserted, if 0, the new text overwrites the old text. When `insertMode` is 0, the cursor shape is thick and covers the whole character; when 1, it is of the default cursor width.

Default toggle key: Insert

**hiliteNumbers COLOR**

Selects the color for number highlighting

**hiliteQStrings COLOR**

Selects the color for highlighting the single-quoted strings

**hiliteQQStrings COLOR**

Selects the color for highlighting the double-quoted strings

**hiliteIDs ARRAY**

An array of scalar pairs that define words to be highlighted. The first item in the pair is an array of words, and the second item is a color value.

**hiliteChars ARRAY**

An array of scalar pairs that define characters to be highlighted. The first item in the pair is a string of characters, and the second item is a color value.

**hiliteREs ARRAY**

An array of scalar pairs that define character patterns to be highlighted. The first item in the pair is a perl regular expression, and the second item is a color value.

Note: these are tricky. Generally, these assume that whatever is captured in `()`, is highlighted, and that capturing parentheses match from the first character onwards. So for simple matches like `(\d+)` (digits) or `(#.*)` this works fine. Things become more interesting if you need to check text after, or especially before the capture. For this, you need to make sure that whatever text is matched by a regexp, it must not move the `pos` pointer as the regexes are internally concatenated with the `\G` anchor before the actual matching takes place (i.e. starting each time from the position the previous regex left off), and use `/gc` flags (advancing `pos` to the match length). Advancing the `pos` will nullify color highlighting on the text after the capture but before the end of the match - so you'll need look-ahead assertions for this type of match, `(?=pattern)` and `(?!pattern)` (see Lookaround Assertions in *perlre* ).

For example, we have a string `ABC123abc`, and we want to match *123* followed by *abc*. This won't work:

```
hiliteREs => [
        '(123)abc',cl::LightRed,
        '(abc)', cl::LightGreen
]
```

while this will:

```
hiliteREs => [
        '(123)(?=abc)',cl::LightRed,
        '(abc)', cl::LightGreen
]
```

If you need to look behind, the corresponding assertions `(?<=pattern)` and `(?<!pattern)` could be used, but these are even more restrictive in that they only support fixed-width looks-behinds (NB: `\K` won't work because of `\G` either). That way, if we want to match 123 that follows ABC, this won't work:

```
hiliteREs =>  [
        '(ABC)',cl::LightBlue,
        '(?<=[ABC]+)(123)',cl::LightRed,
]
```

while this will:

```
hiliteREs =>  [
        '(ABC)',cl::LightBlue,
        '(?<=[ABC]{3})(123)',cl::LightRed,
]
```

**mark MARK [ BLOCK_TYPE ]**

Selects block marking state. If MARK is 1, starts the block marking, if 0 - stops the block marking. When MARK is 1, BLOCK_TYPE can be used to set the selection type ( `bt::XXX` constants ). If BLOCK_TYPE is unset the value of the *blockType* entry is used.

**markers ARRAY**

An array of arrays with integer pairs, X and Y, where each represents visual coordinates in text. Used as anchor storage for fast navigation.

See also: the *add_marker* entry, the *delete_marker* entry

**modified BOOLEAN**

A boolean flag that shows if the text was modified. Can be used externally, to check if the text is to be saved to a file, for example.

**numLines INTEGER**

Returns the number of lines

**offset INTEGER**

Horizontal offset of text lines in pixels.

**persistentBlock BOOLEAN**

Selects whether the selection is canceled as soon as the cursor is moved ( 0 ) or it persists until the selection is explicitly changed ( 1 ).

Default value: 0

**readOnly BOOLEAN**

If 1, no user input is accepted. Manipulations with text are allowed though.

**selection X1, Y1, X2, Y2**

Accepts two pairs of visual coordinates, (X1,Y1) the beginning and (X2,Y2) the end of the new selection, and sets the block according to the the *blockType* entry property.

The selection is null if X1 equals to X2 and Y1 equals to Y2. the *has_selection* entry method returns 1 if the selection is non-null.

**selStart X, Y**

Manages the selection start. See the *selection* entry, X1 and Y1.

**selEnd X, Y**

Manages the selection end. See the *selection* entry, X2 and Y2.

**syntaxHilite BOOLEAN**

Manages the syntax highlighting.

**tabIndent INTEGER**

Maps the tab ( \t ) key to a `tabIndent` number of space characters.

**text TEXT**

Provides access to all the text data. The lines are separated by the new line ( \n ) character.

See also: the *textRef* entry.

**textDirection BOOLEAN**

If set, indicates RTL text input.

**textLigation BOOLEAN**

If set, text may be rendered at better quality with ligation and kerning, however, that comes with a price that some ligatures may be indivisible and form clusters (f.ex. *ff* or *ffi* ligatures). The cursor cannot be positioned inside of a cluster, and thus one can only select them, delete them as a whole, or press Del/Backspace on the cluster's edge.

**textRef TEXT_PTR**

Provides access to all the text data. The lines are separated by the new line ( \n ) character. TEXT_PTR is a pointer to a text string.

The property is more efficient than the *text* entry with large text because the copying of the text scalar to the stack is eliminated.

See also: the *text* entry.

**topLine INTEGER**

Selects the first line of the text drawn.

**undoLimit INTEGER**

Sets limit on the number of stored atomic undo operations. If 0, undo is disabled.

Default value: 1000

**wantTabs BOOLEAN**

Selects the way the tab ( \t ) character is recognized in the user input. If 1, it is recognized as the verbatim Tab key with an ascii value of 0x09; however, this disallows the toolkit widget tab-driven navigation. If 0, the tab character can be entered by pressing the Ctrl+Tab key combination.

Default value: 0

**wantReturns BOOLEAN**

Selects the way the new line ( \n ) character is recognized in the user input. If 1, it is recognized as the verbatim CR key producing newline character(s); however, this disallows the default button activation in the toolkit. If 0, the new line character can be entered by pressing the Ctrl+Enter key combination.

Default value: 1

**wordDelimiters STRING**

Contains a string of characters that are used for locating a word break. Default STRING value consists of punctuation marks, space and tab characters, and the `\xff` character.

See also: the *word_left* entry, the *word_right* entry

**wordWrap BOOLEAN**

Selects whether the long lines are wrapped, or can be positioned outside the horizontal widget borders. A line of text can be represented by more than one line of screen text ( chunk ) . To access the text chunk-wise, use the the *get_chunk* entry method.

### Methods

**add_marker X, Y**

Adds visual coordinates X,Y to the the *markers* entry property.

**back_char [ REPEAT = 1 ]**

Removes REPEAT times a character left to the cursor. If the cursor is on 0 X-position, removes the new-line character and concatenates the two lines.

Default key: Backspace

**cancel_block**

Removes the selection block

Default key: Alt+U

**change_locked**

Returns 1 if the logical locking is on, and 0 if it is off.

See also the *lock_change* entry.

**copy**

Copies the selected text, if any, to the clipboard.

Default key: Ctrl+Insert

**copy_block**

Copies the selected text and inserts it into the cursor position, according to the the *blockType* entry value.

Default key: Alt+C

**cursor_cend**

Moves cursor to the last line

Default key: Ctrl+End

**cursor_chome**

Moves cursor to the first line

Default key: Ctrl+Home

**cursor_cpgdn**

Default key: Ctrl+PageDown

Moves cursor to the end of text.

**cursor_cpgup**

Moves cursor to the beginning of text.

Default key: Ctrl+PageUp

**cursor_down [ REPEAT = 1 ]**

Moves cursor REPEAT times down

Default key: Down

**cursor_end**

Moves cursor to the end of the line

Default key: End

**cursor_home**

Moves cursor to the beginning of the line

Default key: Home

**cursor_left [ REPEAT = 1 ]**

Moves cursor REPEAT times left

Default key: Left

**cursor_right [ REPEAT = 1 ]**

Moves cursor REPEAT times right

Default key: Right

**cursor_up [ REPEAT = 1 ]**

Moves cursor REPEAT times up

Default key: Up

**cursor_pgdn [ REPEAT = 1 ]**

Moves cursor REPEAT pages down

Default key: PageDown

**cursor_pgup [ REPEAT = 1 ]**

Moves cursor REPEAT pages up

Default key: PageUp

**cursor_shift_key [ ACCEL_TABLE_ITEM ]**

Performs action of the cursor movement, bound to ACCEL_TABLE_ITEM action ( defined in `accelTable` or `accelItems` property ), and extends the selection block along the cursor movement. Not called directly.

**cut**

Cuts the selected text into the clipboard.

Default key: Shift+Delete

**delete_block**

Removes the selected text.

Default key: Alt+D

**delete_char [ REPEAT = 1 ]**

Deletes REPEAT characters from the cursor position

Default key: Delete

**delete_line LINE_ID, [ LINES = 1 ]**

Removes LINES of text at LINE_ID.

**delete_current_chunk**

Removes the chunk ( or line, if the *wordWrap* entry is 0 ) at the cursor.

Default key: Ctrl+Y

**delete_chunk CHUNK_ID, [ CHUNKS = 1 ]**

Removes CHUNKS ( or lines, if the *wordWrap* entry is 0 ) of text at CHUNK_ID

**delete_marker INDEX**

Removes marker INDEX in the the *markers* entry list.

**delete_to_end**

Removes the text to the end of the chunk.

Default key: Ctrl+E

**delete_text X, Y, TEXT_LENGTH**

Removes TEXT_LENGTH characters at X,Y physical coordinates

**draw_colorchunk CANVAS, LINE_ID, X, Y, COLOR**

Paints the syntax-highlighted chunk taken from LINE_ID line index, at X, Y. COLOR is used if the syntax highlighting information contains `cl::Fore` as a color reference.

**end_block**

Stops the block selection session.

**find SEARCH_STRING, [ X = 0, Y = 0, REPLACE_LINE = '', OPTIONS ]**

Tries to find ( and, if REPLACE_LINE is defined, to replace with ) SEARCH_STRING starting from (X,Y) physical coordinates. OPTIONS is an integer that is a combination of the `fdo::` constants; the same constants are used in the *Prima::Dialog::FindDialog* section, which provides a graphic interface to the find and replace facilities of this class.

Returns X1, Y, X2, NEW_STRING where X1.Y-X2.Y are physical coordinates of the string found, and NEW_STRING is the replaced version (if any)

**fdo::MatchCase**

If set, the search is case-sensitive.

**fdo::WordsOnly**

If set, SEARCH_STRING must constitute the whole word.

**fdo::RegularExpression**

If set, SEARCH_STRING is a regular expression.

**fdo::BackwardSearch**

If set, the search direction is backward.

**fdo::ReplacePrompt**

Not used in the class, however, used in the *Prima::Dialog::FindDialog* section.

See also: *examples/editor.pl*

**get_chunk CHUNK_ID**

Returns the chunk of text, located at CHUNK_ID. Returns an empty string if the chunk is nonexistent.

**get_chunk_cluster_length CHUNK_ID**

Return the length of a chunk in clusters

**get_chunk_dimension CHUNK_ID**

Finds the line number the CHUNK_ID belongs to, and returns the first chunk of that line and how many chunks the line consists of.

**get_chunk_width TEXT, FROM, LENGTH, [ RETURN_TEXT_PTR ]**

Returns the width in pixels of `substr( TEXT, FROM, LENGTH)`. If FROM is larger than the length of TEXT, TEXT is padded with the space characters. Tab character in TEXT replaced to the *tabIndent* entry times space character. If the RETURN_TEXT_PTR pointer is specified, the converted TEXT is stored there.

**get_line INDEX**

Returns the line of text located at INDEX. Returns an empty string if the line is nonexistent.

**get_line_cluster_length LINE_ID**

Return the length of the line in clusters

**get_line_dimension INDEX**

Returns two integers representing the line at INDEX in the the *wordWrap* entry mode: the first value is the corresponding chunk index, and the second is how many chunks are contained in the line.

See also: the *physical_to_logical* entry.

**get_selected_text**

Return the currently selected text.

**has_selection**

Returns a boolean value, indicating if the selection block is active.

**insert_empty_line LINE_ID, [ REPEAT = 1 ]**

Inserts REPEAT empty lines at LINE_ID.

**insert_line LINE_ID, @TEXT**

Inserts @TEXT strings at LINE_ID

**insert_text TEXT, [ HIGHLIGHT = 0 ]**

Inserts TEXT at the cursor position. If HIGHLIGHT is set to 1, the selection block is canceled and the newly inserted text is selected.

**lock_change BOOLEAN**

Increments ( 1 ) or decrements ( 0 ) lock count. Used to defer change notification in multi-change calls. When the internal lock count hits zero, the `Change` notification is called.

**physical_to_logical X, Y**

Maps visual X,Y coordinates to the logical coordinate system. Returns the same values when the *wordWrap* entry is 0.

**logical_to_physical X, Y**

Maps logical X,Y coordinates to the physical text offset relative to the Y line

Returns the same values when the *wordWrap* entry is 0.

**logical_to_visual X, Y**

Maps logical X,Y coordinates to the visual coordinate system.

Returns the same values when the *wordWrap* entry is 0.

**visual_to_physical X, Y**

Maps visual X,Y coordinates to the physical text offset relative to the Y line

Returns the same X when the line does not contain any right-to-left (RTL) characters or complex glyphs.

**physical_to_visual X, Y**

Maps test offset X from line Y to the visual X coordinate.

Returns the same X when the line does not contain any right-to-left (RTL) characters or complex glyphs.

**mark_horizontal**

Starts block marking session with the `bt::Horizontal` block type.

Default key: Alt+L

**mark_vertical**

Starts block marking session with the `bt::Vertical` block type.

Default key: Alt+B

**overtype_block**

Copies the selected text and overwrites the text next to the cursor position, according to the the *blockType* entry value.

Default key: Alt+O

**paste**

Copies text from the clipboard and inserts it at the cursor position.

Default key: Shift+Insert

**realize_panning**

Performs deferred widget panning, activated by setting {`delayPanning`} to 1. The deferred operations are those performed by the *offset* entry and the *topLine* entry.

**set_line LINE_ID, TEXT, [ OPERATION, FROM, LENGTH ]**

Changes line at LINE_ID to new TEXT. Hint scalars OPERATION, FROM, and LENGTH are used to maintain selection and marking data. OPERATION is an arbitrary string, the ones that are recognized are `'overtype'`, `'add'`, and `'delete'`. FROM and LENGTH define the range of the change; FROM is the cluster offset and LENGTH is the length of the changed text.

**split_line**

Splits a line in two at the cursor position.

Default key: Enter ( or Ctrl+Enter if the *wantReturns* entry is 0 )

**select_all**

Selects all text

**start_block [ BLOCK_TYPE ]**

Begins the block selection session. The block type is BLOCK_TYPE, if it is specified, or the value of the the *blockType* entry property is otherwise.

**update_block**

Adjusts the selection inside the block session, extending or shrinking it to the current cursor position.

**word_left [ REPEAT = 1 ]**

Moves the cursor REPEAT words to the left.

**word_right [ REPEAT = 1 ]**

Moves the cursor REPEAT words to the right.

## 4.9 Prima::ExtLists

Extended functionality for list boxes

### Synopsis

```
use Prima qw(ExtLists Application);

my $vec = '';
vec( $vec, 0, 8) = 0x55;
Prima::CheckList-> new(
        items  => [1..10],
        vector => $vec,
);
run Prima;
```


### Description

The module is intended to be a collection of list boxes with particular enhancements. Currently, the only package defined here is the `Prima::CheckList` class.

### Prima::CheckList

Provides a list box class where each item is equipped with a check box. The check box state can interactively be toggled by the enter key; also the list box reacts differently to click and double click.

#### Properties

**button INDEX, STATE**

Runtime only. Sets INDEXth button STATE to 0 or 1. If the STATE is -1 the button state is toggled.

Returns the new state of the button.

**vector VEC**

VEC is a vector scalar where each bit corresponds to the checked state of each list box item.

See also: `vec` in *perlfunc*.

#### Methods

**clear_all_buttons**

Sets all buttons to state 0

**set_all_buttons**

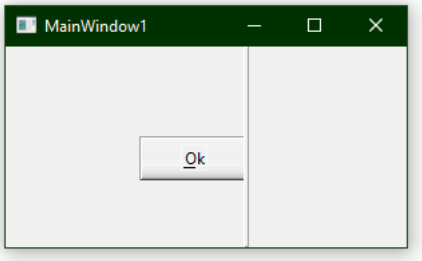
Sets all buttons to state 1

## 4.10 Prima::FrameSet

Frameset widget

### Synopsis

```
use Prima qw(Application Buttons FrameSet);

my $w = Prima::MainWindow->create( size => [300, 150] );

my $frame = $w-> insert( 'FrameSet' =>
        pack          => { fill => 'both', expand => 1 },
        frameSizes    => [qw(60% *)],
        frameProfiles => [ 0,0, { minFrameWidth => 123 }],
);

$frame->insert_to_frame( 0, Button =>
        bottom        => 50,
        text          => '~Ok',
);

run Prima;
```




### Description

Provides the standard frameset widget. The frameset divides its surface among groups of children and allows interactive change of the surface by dragging the frame bars with the mouse.

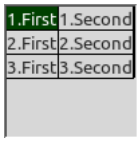
This module defines the `fra::` and `frr::` constants used by the the *arrangement* entry and the *resizeMethod* entry properties, respectively.

## 4.11 Prima::Grids

Grid widgets

### Synopsis

```
use Prima qw(Grids Application);

my $grid = Prima::Grid-> new(
        cells       => [
                [qw(1.First 1.Second 1.Third)],
                [qw(2.First 2.Second 2.Third)],
                [qw(3.First 3.Second 3.Third)],
        ],
        onClick     => sub {
                print $_[0]-> get_cell_text( $_[0]-> focusedCell), " is selected\n";
        }
);
run Prima;
```




## Description

The module provides classes for several abstraction layers for the representation of grid widgets. The class hierarchy is as follows:

```
AbstractGridViewer
        AbstractGrid
        GridViewer
                Grid
```

The root class, `Prima::AbstractGridViewer`, provides a common interface, while by itself it is not directly usable. The main difference between classes is in the way the cell data are stored. The simplest organization of a text-only cell, provided by `Prima::Grid`, stores data as a two-dimensional array of text scalars. More elaborated storage and representation types are not realized, and the programmer is urged to use the more abstract classes to derive their own mechanisms. To organize an item storage different from `Prima::Grid` it is usually enough to overload either the `Stringify`, `Measure`, and `DrawCell` events, or their method counterparts: `get_cell_text`, `columnWidth`, `rowHeight`, and `draw_items`.

The grid widget is designed to contain cells of variable extents, of two types, normal and indent. The indent rows and columns are displayed near to the widget borders, and their cells are drawn with distinguished colors. An example usage for a bottom indent row is a sum row in a spreadsheet application; the top indent row can be used for displaying columns' headers. The normal cells can be selected by the user, scrolled, and selected. The cell selection can only contain rectangular areas and therefore is operated with two integer pairs at the beginning and the end of the selection.

The widget operates in two visual scrolling modes; when the space allows, the scrollbars affect the leftmost and the topmost cell. When the widget is not large enough to accommodate at least one cell and all indent cells, the layout is scrolled pixel-wise. These modes are named 'cell' and 'pixel', after the scrolling units.

The widget allows the interactive changing of cell widths and heights by dragging the grid lines between the cells.

## Prima::AbstractGridViewer

`Prima::AbstractGridViewer`, the base for all grid widgets in the module, provides the interface to generic grid browsing functionality, plus some functionality for text-oriented grids. The class is not usable directly.

`Prima::AbstractGridViewer` is a descendant of `Prima::Widget::GroupScroller`, and some of its properties are not described here.

### Properties

**allowChangeCellHeight BOOLEAN**

If 1, the user is allowed to change the vertical extents of cells by dragging the horizontal grid lines. The prerequisites are: the lines must be set visible via the `drawHGrid` property, the `constantCellHeight` property set to 0, and the changes to the vertical extents can be recorded via the `SetExtent` notification.

Default value: 0

**allowChangeCellWidth BOOLEAN**

If 1, the user is allowed to change the horizontal extents of cells by dragging the horizontal grid lines. The prerequisites are: the lines must be set visible via the `drawVGrid` property, the `constantCellWidth` property set to 0, and the changes to the horizontal extents can be recorded via the `SetExtent` notification.

Default value: 0

**cellIndents X1, Y1, X2, Y2**

Marks the marginal rows and columns as 'indent' cells. The indent cells are drawn with another color pair ( see the *indentCellColor* entry, the *indentCellBackColor* entry ), and cannot be selected and scrolled. X1 and X2 correspond to the number of the indent columns, and Y1 and Y2 to the number of the indent rows.

`leftCell` and `topCell` do not count the indent cells as the leftmost or topmost visible cells; in other words, X1 and Y1 are minimal values for `leftCell` and `topCell` properties.

Default value: 0,0,0,0

**clipCells INTEGER**

A three-state integer property that manages the way clipping is applied when cells are drawn. Depending on the kind of graphic in cells, the clipping may be necessary, or unnecessary.

If the value is 1, the clipping is applied for every column drawn, as the default drawing routines proceeds column-wise. If the value is 2, the clipping is applied for every cell. This setting reduces the drawing speed significantly. If the value is 0, no clipping is applied.

This property is destined for custom-drawn grid widgets when it is the developer's task to decide what kind of clipping suits better. Text grid widgets, `Prima::AbstractGrid` and `Prima::Grid`, are safe with `clipCells` set to 1.

Default value: 1

**columns INTEGER**

Sets the number of columns, including the indent columns. The number of columns must be larger than the number of indent columns.

Default value: 0.

**columnWidth COLUMN [ WIDTH ]**

A run-time property, selects the width of a column. To acquire or set the width, `Measure` and `SetExtent` notifications can be invoked. The result of `Measure` may be cached internally using the `cache_geometry_requests` method.

The width does not include the widths of eventual vertical grid lines.

If `constantCellWidth` is defined, the property is used as its alias.

**constantCellHeight HEIGHT**

If defined, all rows have equal height, HEIGHT pixels. If `undef`, rows have different heights.

Default value: undef

**constantCellWidth WIDTH**

If defined, all rows have equal width, WIDTH pixels. If `undef`, columns have different widths.

Default value: undef

**drawHGrid BOOLEAN**

If 1, horizontal grid lines between cells are drawn with `gridColor`.

Default value: 1

**drawVGrid**

If 1, vertical grid lines between cells are drawn with `gridColor`.

Default value: 1

**dx INTEGER**

A run-time property. Selects horizontal offset in pixels of grid layout in pixel mode.

**dy INTEGER**

A run-time property. Selects vertical offset in pixels of grid layout in pixel mode.

**focusedCell X, Y**

Selects coordinates or the focused cell.

**gridColor COLOR**

Selects the color of the grid lines.

Default value: `cl::Black` .

**gridGravity INTEGER**

The property selects the breadth of the area around the grid lines that react to the grid-dragging mouse events. The minimal value of 0, marks only grid lines themselves as the dragging areas but makes the operation inconvenient for the user. Larger values make the dragging more convenient, but increase the chance that the user will not be able to select too narrow cells with the mouse.

Default value: 3

**indentCellBackColor COLOR**

Selects the background color of the indent cells.

Default value: `cl::Gray` .

**indentCellColor**

Selects the foreground color of the indent cells.

Default value: `cl::Gray` .

**leftCell INTEGER**

Selects the index of the leftmost visible normal cell.

**multiSelect BOOLEAN**

If 1, the normal cells in an arbitrary rectangular area can be marked as selected ( see the *selection* entry ). If 0, only one cell at a time can be selected.

Default value: 0

**rows INTEGER**

Sets the number of rows, including the indent rows. The number of rows must be larger than the number of the indent rows.

Default value: 0.

**topCell**

Selects the index of the topmost visible normal cell.

**rowHeight INTEGER**

A run-time property, selects the height of a row. To acquire or set the height, `Measure` and `SetExtent` notifications can be invoked. The result of `Measure` may be cached internally using the `cache_geometry_requests` method.

The height does not include the heights of eventual horizontal grid lines.

If `constantCellHeight` is defined, the property is used as its alias.

**selection X1, Y1, X2, Y2**

If `multiSelect` is 1, manages the extent of a rectangular area that contains selected cells. If no such area is present, the selection is (-1,-1,-1,-1), and `has_selection` returns 0 .

If `multiSelect` is 0, in get-mode returns the focused cell, and ignores the parameters in the set-mode.

### Methods

**cache_geometry_requests CACHE**

If CACHE is 1, starts caching results of the `Measure` notification, thus making the subsequent `columnWidth` and `rowHeight` calls lighter. If CACHE is 0, flushes the cache.

If a significant geometry change happens during the caching, the cache is not updated automatically, so it is the caller's responsibility to flush the cache.

**deselect_all**

Removes the selection if `multiSelect` is 1.

**draw_cells CANVAS, COLUMNS, ROWS, AREA**

A bulk draw routine, called from `onPaint` to draw individual cells. AREA is an array of four integers in inclusive-inclusive coordinates of the widget inferior without borders and scrollbars ( the result of `get_active_area(2)` call; see the **get_active_area** entry in the *Prima::Widget::IntIndents* section ).

COLUMNS and ROWS are structures that reflect the columns and rows of the cells to be drawn. Each item in these corresponds to a column or row, and is an array with the following layout:

```
0: column or row index
1: type; 0 - normal cell, 1 - indent cell
2: visible cell breadth
3: visible cell start
4: visible cell end
5: real cell start
6: real cell end
```

The coordinates are in the inclusive-inclusive coordinate system and do not include eventual grid space, nor gaps between the indent and normal cells. By default, internal arrays {colsDraw} and {rowsDraw} are passed as COLUMNS and ROWS parameters.

In the Prima::AbstractGrid and Prima::Grid classes <draw_cells> is overloaded to transfer the call to std_draw_text_cells, the text-oriented optimized routine.

**draw_text_cells SCREEN_RECTANGLES, CELL_RECTANGLES, CELL_INDECES, FONT_HEIGHT**

A bulk routine for drawing text cells, is called from std_draw_text_cells .

SCREEN_RECTANGLES and CELL_RECTANGLES are arrays, where each item is a rectangle with an exterior of a cell. SCREEN_RECTANGLES contains rectangles that cover the visible area of the cell; CELL_RECTANGLES contains rectangles that span the cell extent disregarding its eventual partial visibility. For example, a 100-pixel cell with only its left half visible would contain corresponding arrays [150,150,200,250] in SCREEN_RECTANGLES, and [150,150,250,250] in CELL_RECTANGLES.

CELL_INDECES contains arrays of the cell coordinates; each array item is an array of integer pairs where item 0 is the column, and item 1 is the row of the cell.

FONT_HEIGHT is the current font height value, as draw_text_cells is mostly used for text operations and may require vertical text justification.

**get_cell_area [ WIDTH, HEIGHT ]**

Returns screen area in the inclusive-inclusive pixel coordinates. The area is used to display normal cells. The extensions are related to the current size of a widget, however, can be overridden by specifying WIDTH and HEIGHT.

**get_cell_alignment COLUMN, ROW**

Returns two ta:: constants for horizontal and vertical cell text alignment. Since the class does not assume the item storage organization, the values are queried via the GetAlignment notification.

**get_cell_text COLUMN, ROW**

Returns the text string assigned to the cell in COLUMN and ROW. Since the class does not assume the item storage organization, the text is queried via the Stringify notification.

**get_range VERTICAL, INDEX**

Returns a pair of integers, the minimal and maximal breadth of INDEXth column or row in pixels. If VERTICAL is 1, the rows are queried; if 0, the columns.

The method calls the GetRange notification.

**get_screen_cell_info COLUMN, ROW**

Returns information about the cell in COLUMN and ROW, if it is currently visible. The returned parameters are indexed by the gsci::XXX constants:

```
gsci::COL_INDEX - visual column number where the cell is displayed
gsci::ROW_INDEX - visual row number where the cell is displayed
gsci::V_FULL    - cell is fully visible

gsci::V_LEFT    - an inclusive-inclusive rectangle of the visible
gsci::V_BOTTOM      part of the cell. These four indices are grouped
gsci::V_RIGHT       under list constant, gsci::V_RECT.
gsci::V_TOP

gsci::LEFT      - an inclusive-inclusive rectangle of the cell, as if
gsci::BOTTOM        it is fully visible. These four indices are grouped
gsci::RIGHT         under list constant, gsci::RECT. If gsci::V_FULL
gsci::TOP           is 1, these values are identical to those in gsci::V_RECT.
```

If the cell is not visible, returns an empty array.

**has_selection**

Returns a boolean value, indicating whether the grid contains a selection (1) or not (0).

**point2cell X, Y, [ OMIT_GRID = 0 ]**

Returns information about the point X, Y in widget coordinates. The method returns two integers CX and CY with cell coordinates and an eventual HINTS hash that contains more information about the pixel location. If OMIT_GRID is set to 1 but the pixel belongs to the grid, the pixels are treated as part of an adjacent cell. The call syntax is:

```
($CX, $CY, %HINTS) = $self->point2cell( $X, $Y);
```

If the pixel lies within the cell boundaries by either coordinate, CX and/or CY are correspondingly set to that cell column and/or row. When the pixel is outside the cell space, CX and/or CY are set to -1.

HINTS may contain the following values:

`x` **and** `y`

If 0, the coordinate lies within the boundaries of a cell.

If -1, the coordinate is on the left/top of the cell body.

If +1, the coordinate is on the right/bottom of the cell body, but within the widget.

If +2, the coordinate is on the right/bottom of the cell body, but outside the widget.

`x_type` **and** `y_type`

Present when `x` or `y` values are 0.

If 0, the cell is a normal cell.

If -1, the cell is a left or a top indent cell.

If +1, the cell is a right or a bottom indent cell.

`x_grid` **and** `y_grid`

If 1, the point is at a grid line. This case can only happen when OMIT_GRID is 0. If `allowChangeCellHeight` and/or `allowChangeCellWidth` are set, treats also `gridGravity`-broad pixels strips on both sides of the line as the grid.

Also, values of `x_left`/`x_right` or `y_bottom`/`y_top` might be set.

`x_left`/`x_right` **and** `y_bottom`/`y_top`

Present together with `x_grid` or `y_grid`. Contain the indices of the cells adjacent to the grid line.

`x_gap` **and** `y_gap`

If 1, the point is inside a gap between the last normal cell and the first right/bottom indent cell.

`normal`

If 1, the point lies within the boundaries of a normal cell.

`indent`

If 1, the point lies within the boundaries of an indent cell.

`grid`

If 1, the point is at a grid line.

`exterior`

If 1, the point is in an inoperable area or outside the widget boundaries.

**redraw_cell X, Y**

Repaints the cell with coordinates X and Y.

**reset**

Recalculates internal geometry variables.

**select_all**

Marks all cells as selected if `multiSelect` is 1.

**std_draw_text_cells CANVAS, COLUMNS, ROWS, AREA**

An optimized bulk routine for text-oriented grid widgets. The optimization is achieved under the assumption that each cell is drawn with two colors only so that the color switching can be reduced.

The routine itself paints the cells' background and then calls `draw_text_cells` to draw the cells' content.

For the explanation of COLUMNS, ROWS, and AREA parameters see the *draw_cells* entry .

**Events**

**DrawCell CANVAS, COLUMN, ROW, INDENT, @SCREEN_RECT, @CELL_RECT, SELECTED, FOCUSED, PRELIGHT**

Called when a cell with COLUMN and ROW coordinates is to be drawn on CANVAS. SCREEN_RECT is a cell rectangle in widget coordinates, where the item is to be drawn. CELL_RECT is the same as SCREEN_RECT but calculated as if the cell is fully visible.

SELECTED, FOCUSED, and PRELIGHT are boolean flags, if the cell must be drawn correspondingly in selected, focused, and pre-lighted states.

**GetAlignment COLUMN, ROW, HORIZONTAL_ALIGN_REF, VERTICAL_ALIGN_REF**

Stores two text alignment `ta::` constants, assigned to the cell with COLUMN and ROW coordinates, into HORIZONTAL_ALIGN_REF and VERTICAL_ALIGN_REF scalar references.

**GetRange VERTICAL, INDEX, MIN, MAX**

Stores minimal and maximal breadth of INDEXth column ( VERTICAL = 0 ) or row ( VERTICAL = 1) in the corresponding MIN and MAX scalar references.

**Measure VERTICAL, INDEX, BREADTH**

Stores breadth in pixels of the INDEXth column ( VERTICAL = 0 ) or row ( VERTICAL = 1) into BREADTH scalar reference.

This notification by default may be called from within the `begin_paint_info/end_paint_info` brackets. To disable this feature set the internal flag {`NoBulkPaintInfo`} to 1.

**SelectCell COLUMN, ROW**

Called when a cell with COLUMN and ROW coordinates is focused.

**SetExtent VERTICAL, INDEX, BREADTH**

Reports breadth in pixels of the INDEXth column ( VERTICAL = 0 ) or row ( VERTICAL = 1), as a response to the `columnWidth` and `rowHeight` calls.

**Stringify COLUMN, ROW, TEXT_REF**

Stores the text string, assigned to the cell with COLUMN and ROW coordinates, into the TEXT_REF scalar reference.

## Prima::AbstractGrid

The same as its ascendant, `Prima::AbstractGridViewer`, except that it does not propagate the `DrawItem` message, assuming that the items must be drawn as text.

## Prima::GridViewer

The class implements cell data and geometry storage mechanisms but leaves the data format to the programmer. The cells are accessible via the `cells` property and several other helper routines.

The cell data are stored in an array, where each item corresponds to a row, and contains an array of scalars, where each corresponds to a column. All data managing routines, that accept two-dimensional arrays, assume that the column arrays are of the same width.

For example, `[[1,2,3]]` is a valid one-row, three-column structure, and `[[1,2],[2,3],[3,4]]` is a valid three-row, two-column structure. The structure `[[1],[2,3],[3,4]]` is invalid, since its first row has one column, while the others have two.

`Prima::GridViewer` is derived from `Prima::AbstractGridViewer`.

### Properties

**allowChangeCellHeight**

Default value: 1

**allowChangeCellWidth**

Default value: 1

**cell COLUMN, ROW, [ DATA ]**

Run-time property. Selects the data in the cell with COLUMN and ROW coordinates.

**cells [ ARRAY ]**

The property accepts or returns all cells as a two-dimensional rectangular array or scalars.

**columns INDEX**

A read-only property; returns the number of columns.

**rows INDEX**

A read-only property; returns the number of rows.

### Methods

**add_column CELLS**

Inserts a one-dimensional array of scalars to the end of columns.

**add_columns CELLS**

Inserts a two-dimensional array of scalars to the end of columns.

**add_row CELLS**

Inserts a one-dimensional array of scalars to the end of rows.

**add_rows CELLS**

Inserts a two-dimensional array of scalars to the end of rows.

**delete_columns OFFSET, LENGTH**

Removes LENGTH columns starting from OFFSET. Negative values are accepted.

**delete_rows OFFSET, LENGTH**

Removes LENGTH rows starting from OFFSET. Negative values are accepted.

**insert_column OFFSET, CELLS**

Inserts a one-dimensional array of scalars as column OFFSET. Negative values are accepted.

**insert_columns OFFSET, CELLS**

Inserts a two-dimensional array of scalars in column OFFSET. Negative values are accepted.

**insert_row**

Inserts a one-dimensional array of scalars as row OFFSET. Negative values are accepted.

**insert_rows**

Inserts a two-dimensional array of scalars in row OFFSET. Negative values are accepted.

## Prima::Grid

Descendant of `Prima::GridViewer`, declares format of cell data as a single text string. Provides the standard text grid widget that has all the functionality of its ascendants.

### Methods

**get_cell_alignment COLUMN, ROW**

Returns two `ta::` constants for horizontal and vertical cell text alignment. Since the item storage organization is implemented, does so without calling the `GetAlignment` notification.

**get_cell_text COLUMN, ROW**

Returns text string assigned to the cell in COLUMN and ROW. Since the item storage organization is implemented, does so without calling the `Stringify` notification.

## 4.12 Prima::HelpViewer

The built-in POD browser

### Usage

The module presents two packages, `Prima::HelpViewer` and `Prima::PodViewWindow`. Their purpose is to serve as a mediator between the `Prima::PodView` package, the toolkit help interface, and the user. `Prima::PodViewWindow` includes all the user functionality needed, including text search, color and font setup, printing, etc. `Prima::HelpViewer` provides two methods - `open` and `close`, used by `Prima::Application` for invocation of the help viewer .

### Help

The browser can be used to view and print POD ( plain old documentation ) files. See the command overview below for a more detailed description

**File**

**Open**

Presents a file selection dialog, when the user can select a file to browse. The file must contain POD content, otherwise, a warning is displayed.

**Goto**

Asks for a manpage, that is searched in PATH and the perl installation directories.

**New window**

Opens the new viewer window with the same context.

**Run**

Commands in this group call external processes

**prima-class**

prima-class is the utility for displaying the widget class hierarchies. The command asks for the Prima class to display the hierarchy information.

**Print**

Provides a dialog where the user can select the appropriate printer device and its options.

Prints the current topic to the selected printer.

If the the *Full text view* entry menu item is checked, prints the whole manpage.

**Close window**

Closes the window.

**Close all windows**

Closes all help viewer windows.

**View**

**Increase font**

Increases the currently selected font by 2 points.

**Decrease font**

Decreases the currently selected font by 2 points.

**Full text view**

If checked, the whole manpage is displayed. Otherwise, its content is presented as a set of topics, and only a single topic is displayed.

**Find**

Presents a text find dialog where the user can type the text to search, and select the search options - the search direction, scope, etc.

**Find again**

Starts search for the text, entered in the last text find dialog, with the same search options.

**Fast find**

The following commands provide a simple vi-style text search functionality - character keys ?,/,n,N bound to the commands below:

**Forward**

Presents an input line where a text can be entered; the text search is performed parallel to the input.

**Backward**

Same as the the *Forward* entry option, except that the search direction is backward.

**Repeat forward**

Repeats the search in the same direction as the initial search

**Repeat backward**

Repeat the search in the reverse direction from the initial search

**Setup**

Presents a setup dialog where the user can select appropriate fonts and colors.

**Go**

**Back**

Displays the previously visited manpage or topic

**Forward**

Displays the previously visited manpage or topic that was left via the the *Back* entry command.

**Up**

Displays the upper-level topic within the manpage

**Previous**

Moves to the previous topic within the manpage

**Next**

Moves to the next topic within the manpage

**Help**

**About**

Displays the information about the help viewer.

**Help**

Displays the information about the usage of the help viewer

## 4.13 Prima::ImageViewer

Image, icon, and bitmap viewer

### Synopsis

```
use Prima qw(ImageViewer StdBitmap Application);
Prima::ImageViewer-> new(
        image => Prima::StdBitmap::image(0),
        zoom  => 2.718,
);
run Prima;
```

## Description

The module contains the `Prima::ImageViewer` class which provides image-displaying functionality. The widget can display images, icons, and bitmaps, and allows zooming.

`Prima::ImageViewer` is a descendant of `Prima::Widget::ScrollWidget` and inherits its document scrolling behavior and programming interface. See the *Prima::Widget::ScrollWidget* section for details.

## API

### Properties

**alignment INTEGER**

One of the following `ta::XXX` constants:

```
ta::Left
ta::Center
ta::Right
```

Selects the horizontal image alignment.

Default value: `ta::Left`

**autoZoom BOOLEAN**

When set, the image is automatically stretched while keeping aspects to the best available fit, given the `zoomPrecision`. Scrollbars are turned off if `autoZoom` is set to 1.

**image OBJECT**

Selects the image object to be displayed. OBJECT can be an instance of the `Prima::Image`, `Prima::Icon`, or `Prima::DeviceBitmap` classes.

**imageFile FILE**

Sets the image FILE to be loaded and displayed. Is rarely used since does not return a success flag.

**scaling ist::XX**

Applies scaling when drawing an image.

Default: `ist::Box`, default cheap scaling.

Warning: scaling types above the `ist::Box` might be somewhat expensive

**stretch BOOLEAN**

If set, the image is simply stretched over the visual area, without keeping the aspect. Scroll bars, zooming and keyboard navigation become disabled.

**quality BOOLEAN**

A boolean flag, selects if the palette of `image` is to be copied into the widget palette, providing higher visual quality on paletted displays. See also the **palette** entry in the *Prima::Widget* section.

Default value: 1

**valignment INTEGER**

One of the following `ta::XXX` constants:

```
ta::Top
ta::Middle or ta::Center
ta::Bottom
```

Selects the vertical image alignment.

Note: The `ta::Middle` value is not equal to `ta::Center`'s, however, both constants produce an equal effect here.

Default value: `ta::Bottom`

**zoom FLOAT**

Selects the image zoom level. The acceptable value range is between 0.01 and 100. The zoom value is rounded to the closest value divisible by 1/`zoomPrecision`. For example, if `zoomPrecision` is 100, the zoom values will be rounded to the precision of hundredth - to fiftieth and twentieth fractional values - .02, .04, .05, .06, .08, and 0.1 . When `zoomPrecision` is 1000, the precision is one thousandth, and so on.

Default value: 1

**zoomPrecision INTEGER**

Zoom precision of the `zoom` property. The minimal acceptable value is 10, where the zoom factor will be rounded to 0.2, 0.4, 0.5, 0.6, 0.8, and 1.0 .

The reason behind this arithmetics is that when an image of an arbitrary zoom factor is requested to be displayed, the image sometimes must be drawn from a fractional image pixel. In an example that only involves integer pixels, a 10x zoomed image shifted 3 pixels left must be displayed so that the first image pixel from the left occupies 7 screen pixels, and the next ones - 10 screen pixels. That means that the correct image display routine must ask the system to draw the image at the offset of -3 screen pixels, where the first image pixel column would correspond to that offset.

When the zoom factor is fractional, the picture is getting more complex. For example, with the zoom factor of 12.345 and zero screen offset, the first image pixel begins at the 12th screen pixel, the next one - at the 25th ( because of the roundoff ), then the 37th, etc etc. If the image is 2000x2000 pixels wide and is asked to be drawn so that it appears shifted 499 screen image pixels left, it needs to be drawn from the 499/12.345=40.42122th image pixel. It might seem that indeed it would be enough to ask the system to begin

drawing from image pixel 40, and offset int(0.42122*12.345)=5 screen pixels to the left, however, that procedure will not account for the correct fixed point roundoff that accumulates as the system scales the image. For the zoom factor of 12.345 this roundoff sequence is, as we have seen before, (12,25,37,49,62,74,86,99,111,123) for the first 10 pixels displayed, that occupy (12,13,12,12,13,12,12,13,12,12) screen pixels correspondingly. For the pixels starting at 499, the sequence is (506,519,531,543,556,568,580,593,605,617) offsets or (13,12,12,13,13,12,12,13,12,12) widths -- note the two subsequent 13s there. This sequence begins to repeat itself after 200 iterations (12.345*200=2469.000), which means that to achieve correct display results, the image must be asked to be displayed from as far as image pixel 0 if image's first pixel on the screen is between 0 and 199 ( or for screen pixels 0-2468), then from image pixel 200 for offsets 200-399, ( screen pixels 2469-4937), and so on.

Since the system internally allocates memory for image scaling, that means that up to 2*200*min(window_width,image_width)*bytes_per_pixel unnecessary bytes will be allocated for each image drawing call (2 because the calculations are valid for both the vertical and horizontal strips), and this can lead to a slowdown or even request failure when image or window dimensions are large. The proposed solution is to round off the accepted zoom factors so that these offsets are kept small. For example, the N.25 zoom factors require only max 1/.25=4 extra pixels. When the `zoomPrecision` value is set to 100, the zoom factors are rounded to 0.X2, 0.X4, 0.X5, 0.X6, 0.X8, and 0.X0, thus requiring max 50 extra pixels.

NB. If, despite the efforts, the property gets in the way, increase it to 1000 or even 10000, but note that this may lead to problems.

Default value: 100

### Methods

#### on_paint SELF, CANVAS

The `Paint` notification handler is mentioned here for the specific case of its return value, that is the return value of the internal `put_image` call. For those who might be interested in `put_image` failures, which mostly occur when trying to draw an image that is too big, the following code might be useful:

```
sub on_paint
{
    my ( $self, $canvas) = @_;
    warn "put_image() error:$@" unless $self-> SUPER::on_paint($canvas);
}
```

#### screen2point X, Y, [ X, Y, ... ]

Performs translation of integer pairs as (X,Y)-points from the widget coordinates to pixel offsets in the image coordinate system. Takes into account zoom level, image alignments, and offsets. Returns an array of the same length as the input.

Useful for determining correspondence, for example, of a mouse event to an image point.

The reverse function is `point2screen`.

#### point2screen X, Y, [ X, Y, ... ]

Performs translation of integer pairs as (X,Y)-points from image pixel offset to widget image coordinates. Takes into account zoom level, image alignments, and offsets. Returns an array of the same length as the input.

Useful for determining the screen location of an image point.

The reverse function is `screen2point`.

**watch_load_progress IMAGE**

When called, the image viewer begins to track the progress of the IMAGE being loaded ( see the **load** entry in the *Prima::Image* section ) and incrementally displays the loading picture. As soon as IMAGE begins to load, it replaces the existing the `image` property value. Example:

```
$i = Prima::Image-> new;
$viewer-> watch_load_progress( $i);
$i-> load('huge.jpg');
$viewer-> unwatch_load_progress;
```

A similar functionality is present in the *Prima::Dialog::ImageDialog* section.

**unwatch_load_progress CLEAR_IMAGE=1**

Stops monitoring the image loading progress. If CLEAR_IMAGE is 0, the leftovers of the incremental loading stay intact in `image` property. Otherwise, `image` is set to `undef`.

**zoom_round ZOOM**

Rounds the zoom factor to `zoomPrecision` precision, returns the rounded zoom value. The algorithm is the same as used internally in the `zoom` property.

## 4.14 Prima::InputLine

Input line widget

### Synopsis

```
use Prima qw(InputLine Application);
Prima::InputLine-> new( text => 'Hello world!');
run Prima;
```


## Description

The class provides the basic functionality of an input line, including hidden input, read-only state, selection, and clipboard operations. The input line text data is stored in the the *text* entry property.

## API

### Events

#### Change

The notification is called when the the *text* entry property is changed, either interactively or as a result of a direct call.

#### Validate TEXT_REF

The notification is called right before the the *text* entry property is changed, either interactively or as a result of a direct call. The custom code has a chance to validate the text and/or provide some sort of interactive feedback.

See also: the *blink* entry

### Properties

#### alignment INTEGER

One of the following `ta::` constants, defining the text alignment:

```
ta::Left
ta::Right
ta::Center
```

Default value: `ta::Left`

#### autoHeight BOOLEAN

If 1, adjusts the height of the widget automatically when its font changes.

Default value: 1

#### autoSelect BOOLEAN

If 1, all the text is selected when the widget becomes focused.

Default value: 1

**autoTab BOOLEAN**

If 1, ignores the keyboard `kb::Left` and `kb::Right` commands, when these are received when the cursor is at the beginning or the end of text and cannot be moved farther. The result of this is that the default handler moves focus to a neighbor widget, in a way as if the Tab key was pressed.

Default value: 0

**borderWidth INTEGER**

Width of the border around the widget.

Default value: depends on the skin

**charOffset INTEGER**

Managets the current position of the cursor

**firstChar**

Selects the first visible cluster of text

**insertMode BOOLEAN**

Manages the typing mode - if 1, the typed text is inserted, if 0, the text overwrites the old text. When `insertMode` is 0 the cursor shape is thick and covers the whole character; when 1, it is of the default width.

Default toggle key: Insert

**maxLen INTEGER**

The maximal length of the text, that can be stored into the *text* entry or typed by the user.

Default value: 256

**passwordChar CHARACTER**

A character to be shown instead of the text letters when the *writeOnly* entry property value is 1.

Default value: `'*'`

**readOnly BOOLEAN**

If 1, the text cannot be edited by the user.

Default value: 0

**selection START, END**

Two integers, specifying the beginning and the end of the selected text, in clusters. A case with no selection is when START equals END.

**selStart INTEGER**

Selects the start of the text selection.

**selEnd INTEGER**

Selects the end of the text selection.

**textDirection BOOLEAN**

If set, indicates RTL text input.

**textLigation BOOLEAN**

If set, text may be rendered at better quality with ligation and kerning, however, that comes with a price that some ligatures may be indivisible and form clusters (f.ex. *ff* or *ffi* ligatures). The cursor cannot go inside such clusters, and thus one can only select them, delete them as a whole, or press Del/Backspace on the cluster's edge.

Toggle during runtime with Ctrl+Shift+L.

**wordDelimiters STRING**

Contains the string of characters that are used for locating a word break. Default STRING value consists of punctuation marks, space, tab, and \xff character.

**writeOnly BOOLEAN**

If 1, the input is not shown but mapped to the *passwordChar* entry characters. Useful for a password entry.

Default value: 0

### Methods

**blink %options**

Produces a short blink by setting the background to red color. Can be used to signal an invalid input, f ex from `on_validate`. `%options` allows the `backColor` and `color` entries.

**copy**

Copies the selected text, if any, to the clipboard.

Default key: Ctrl+Insert

**cut**

Cuts the selected text into the clipboard.

Default key: Shift+Delete

**delete**

Removes the selected text.

Default key: Delete

**paste**

Copies text from the clipboard and inserts it in the cursor position.

Default key: Shift+Insert

**select_all**

Selects all text

### Bi-directional input and output

When working on bidirectional texts, or text represented by complex script shaping, values returned from the methods `firstChar`, `charOffset`, `selection`, etc cannot be used to calculate text offsets f.ex. via `substr`. Note that these values are in clusters, not in characters (see the *Prima::Drawable::Glyphs* section for the description>. Also, the selection ranges of bidi text become not straightforward. Use the following methods whenever text manipulations are needed:

**char_at OFFSET**

Returns character at cluster OFFSET

**selection_strpos**

Returns range of characters covered by the selection.

## 4.15 Prima::KeySelector

Key combination widget and routines

### Description

The module provides a standard widget for selecting user-defined key combinations. The widget class allows import, export, and modification of key combinations. The module also provides a set of routines useful for the conversion of key combinations between various representations.

### Synopsis

```
my $ks = Prima::KeySelector-> create( );
$ks-> key( km::Alt | ord('X'));
print Prima::KeySelector::describe( $ks-> key );
```

### API

#### Properties

**key INTEGER**

Selects a key combination in integer format. The format is described in the **Hotkey** entry in the *Prima::Menu* section, and is a combination of the `km::XXX` key modifiers and is either a `kb::XXX` virtual key or a character code value.

The property allows almost, but not all possible combinations of key constants. Only the `km::Ctrl`, `km::Alt`, and `km::Shift` modifiers are allowed.

#### Methods

All methods must be called without the object as a first parameter.

**describe KEY**

Accepts KEY in integer format and returns a string description of the key combination in a human-readable format. Useful for supplying an accelerator text to a menu.

```
print Prima::KeySelector::describe( km::Shift|km::Ctrl|km::F10);
Ctrl+Shift+F10
```

**export KEY**

Accepts KEY in integer format and returns a string with a perl-evaluable expression, which after the evaluation resolves to the original KEY value. Useful for storing a key into text config files, where the value must be both human-readable and easily passed to a program.

```
print Prima::KeySelector::export( km::Shift|km::Ctrl|km::F10);
km::Shift|km::Ctrl|km::F10
```

**shortcut KEY**

Converts KEY from integer format to a string, acceptable by `Prima::AbstractMenu` input methods.

```
print Prima::KeySelector::shortcut( km::Ctrl|ord('X'));
^X
```

**translate_codes KEY, [ USE_CTRL = 0 ]**

Converts KEY in integer format to three integers in the format accepted by the the **KeyDown** entry in the *Prima::Widget* section event: code, key, and modifier. USE_CTRL is only relevant when the KEY first byte ( `KEY & 0xFF` ) is between 1 and 26, which means that the key is a combination of an alpha key with the control key. If USE_CTRL is 1, the code result is unaltered and is in range 1 - 26. Otherwise, the code result is converted to the character code ( 1 to ord('A'), 2 to ord('B'), etc ).

## 4.16 Prima::Menus

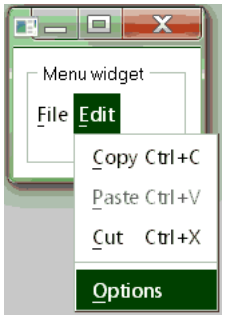
Menu widgets

### Description

This module contains classes that can create menu widgets used as regular widgets, without any special considerations about system-depended menus.

### Synopsis

```
use Prima qw(Application Menus);
my $w = Prima::MainWindow->new(
        accelItems => [['~File' => [
                ['Exit' => sub { exit } ],
        ]]],
        onMouseDown => sub {
                Prima::Menu::Popup->new(menu => $_[0]-> accelTable)->popup;
        },
        height => 100,
);
$w->insert( 'Prima::Menu::Bar',
        pack  => { fill => 'x', expand => 1},
        menu  => $w-> accelTable,
);
run Prima;
```



## 4.17 Prima::Label

Static text widget

### Description

The class is designed for the display of text and provides no user interaction. The text output capabilities include wrapping, horizontal and vertical alignment, and automatic widget resizing to match text extensions. If the text contains a tilde-escaped ( hot ) character, the label can explicitly focus the specified widget upon pressing of the character key; this feature is useful for the design of dialogs.

Labels can display rich text with links. See the *Prima::Drawable::Markup* section for more.

### Synopsis

```
use Prima qw(Label InputLine Application);
my $w = Prima::MainWindow->new;
$w->insert( 'Prima::Label',
        text      => 'Enter ~name:',
        focusLink => 'InputLine1',
        alignment => ta::Center,
        pack => { fill => 'x', side => 'top', pad => 10 },
);
$w->insert(
        'Prima::InputLine',
        text => '',
        pack => { fill => 'x', side => 'top', pad => 10 },
);
run Prima;
```


### API

#### Properties

**alignment INTEGER**

One of the following `ta::XXX` constants:

```
ta::Left
ta::Center
ta::Right
```

Selects the horizontal text alignment.

Default value: `ta::Left`

**autoHeight BOOLEAN**

If 1, the widget height is automatically changed as text extensions change.

Default value: 0

**autoWidth BOOLEAN**

If 1, the widget width is automatically changed as text extensions change.

Default value: 1

**focusLink WIDGET**

Points to a widget or a widget name (has to be a sibling widget), which is explicitly focused when the user presses the combination of the hotkey with the `Alt` key.

Prima::Label does not provide a separate property to access the hotkey value from the tilde-escaped string, however, it can be read from the {`accel`} variable.

Default value: `undef`.

**hotKey CHAR**

A key (defined by CHAR) that the label will react to if pressed if the label has the focus The combination ALT + Key works always whether the label has the focus or not

**linkColor COLOR**

The color of text in links.

The default value is taken from the *Prima::Widget::Link* section and is currently hardcoded as green. So far there is no support for the system link color.

**showAccelChar BOOLEAN**

If 0, the tilde ( `~` ) character is collapsed from the text, and the hot character is underlined. When the user presses the combination of the escaped character with the `Alt` key, the `focusLink` widget is explicitly focused.

If 1, the text is shown as is, and no hot character is underlined. Key combinations with the `Alt` key are not recognized. See also: `hotKey`.

Default value: 0

**showPartial BOOLEAN**

Used to determine if the last line of text should be drawn if it can not be vertically fit in the widget interior. If 1, the last line is shown even if not visible in full. If 0, only full lines are drawn.

Default value: 1

**textJustify $BOOL | { letter => 0, word => 0, kashida => 0, min_kashida => 0 } | %VALUES**

If set, justifies wrapped text according to the option passed in the hash ( see the **arabic_justify** entry in the *Prima::Drawable::Glyphs* section and the **interspace_justify** entry in the *Prima::Drawable::Glyphs* section ). Can accept three forms:

If an anonymous hash is used, overwrites all the currently defined options.

If `$BOOL` is used, treated as a shortcut for { `letter => $BOOL, word => $BOOL, kashida => $BOOL` }; consequent get-calls return a full hash, not the `$BOOL` value.

If the `%VALUES` form is used, overwrites only values found in `%VALUES`.

Only actual when `wordWrap` is set.

**textDirection BOOLEAN**

If set, indicates RTL text direction.

**wordWrap BOOLEAN**

If 0, the text is not wrapped unless new line characters are present in the text.

If 1, the text is wrapped if it can not be fit horizontally in the widget interior. The text is also wrapped over new lines.

Default value: 0

**valignment INTEGER**

One of the following `ta::XXX` constants:

```
ta::Top
ta::Middle or ta::Center
ta::Bottom
```

Selects the vertical text alignment.

Note: the `ta::Middle` value is not equal to `ta::Center`'s, however, both constants produce an equal effect here.

Default value: `ta::Top`

## 4.18 Prima::Lists

List widgets

### Description

The module provides several listbox classes that differ in the way items in the list widget are associated with data. The hierarchy of classes is as follows:

```
AbstractListViewer
        AbstractListBox
        ListViewer
                ProtectedListBox
                ListBox
```

The root class `Prima::AbstractListViewer` provides a common interface that is though not usable directly. The main differences between classes are centered around the way the items are stored. The simplest organization of a text-only item list, provided by `Prima::ListBox`, stores an array of text scalars in a widget. More elaborated storage and representation types are not realized, and the programmer is urged to use the more abstract classes to derive their own mechanisms. For example, for a list of items that contain text strings and icons see the **Prima::DirectoryListBox** entry in the *Prima::Dialog::FileDialog* section. To organize an item storage different from `Prima::ListBox` it is usually enough to overload either the `Stringify`, `MeasureItem`, and `DrawItem` events, or their method counterparts: `get_item_text`, `get_item_width`, and `draw_items`.

### Prima::AbstractListViewer

`Prima::AbstractListViewer` is a descendant of `Prima::Widget::GroupScroller`, and some of its properties are not described here.

The class provides an interface to generic list browsing functionality, plus functionality for text-oriented lists. The class is not usable directly.

#### Properties

**autoHeight BOOLEAN**

If 1, the item height is changed automatically when the widget font is changed; this is useful for text items. If 0, the item height is not changed; this is useful for non-text items.

Default value: 1

**count INTEGER**

An integer property, used to access the number of items in the list. Since it is tied to the item storage organization, and hence the possibility of changing the number of items, this property is often declared as read-only in descendants of `Prima::AbstractListViewer`.

**dragable BOOLEAN**

If 1, allows the items to be dragged interactively by pressing the Control key together with the left mouse button. If 0, item dragging is disabled.

Default value: 0

**drawGrid BOOLEAN**

If 1, vertical grid lines between columns are drawn with `gridColor`. Actual only in multi-column mode.

Default value: 1

**extendedSelect BOOLEAN**

Manages the way the user selects multiple items that is only actual when `multiSelect` is 1. If 0, the user must click each item to mark it as selected. If 1, the user can drag the mouse or use the Shift key plus arrow keys to perform range selection; the Control key can be used to select individual items.

Default value: 0

**focusedItem INDEX**

Selects the focused item index. If -1, no item is focused. It is mostly a run-time property, however, it can be set during the widget creation stage given that the item list is accessible at this stage as well.

Default value: -1

**gridColor COLOR**

Color used for drawing vertical divider lines for multi-column list widgets. The list classes support also the indirect way of setting the grid color, as well as the widget does, via the `colorIndex` property. To achieve this, the `ci::Grid` constant is declared ( for more detail see the **colorIndex** entry in the *Prima::Widget* section ).

Default value: `cl::Black`.

**integralHeight BOOLEAN**

If 1, only the items that fit vertically in the widget interiors are drawn. If 0, the partially visible items are drawn also.

Default value: 0

**integralWidth BOOLEAN**

If 1, only the items that fit horizontally in the widget interiors are drawn. If 0, the partially visible items are drawn also. Actual only in the multi-column mode.

Default value: 0

**itemHeight INTEGER**

Selects the height of the items in pixels. Since the list classes do not support items with variable heights, changes to this property affect all items.

Default value: default font height

**itemWidth INTEGER**

Selects the width of the items in pixels. Since the list classes do not support items with variable widths, changes to this property affect all items.

Default value: default widget width

**multiSelect BOOLEAN**

If 0, the user can only select one item, and it is reported by the `focusedItem` property. If 1, the user can select more than one item. In this case, the `focusedItem`'th item is not necessarily selected. To access the selected item list use the `selectedItems` property.

Default value: 0

**multiColumn BOOLEAN**

If 0, the items are arranged vertically in a single column and the main scroll bar is vertical. If 1, the items are arranged in several columns, each `itemWidth` pixels wide. In this case, the main scroll bar is horizontal.

**offset INTEGER**

Horizontal offset of an item list in pixels.

**topItem INTEGER**

Selects the first item drawn.

**selectedCount INTEGER**

A read-only property. Returns the number of selected items.

**selectedItems ARRAY**

ARRAY is an array of integer indices of selected items.

**vertical BOOLEAN**

Sets the general direction of items in multi-column mode. If 1, items increase down-to-right. Otherwise, right-to-down.

Doesn't have any effect in single-column mode. Default value: 1.

### Methods

**add_selection ARRAY, FLAG**

Sets item indices from ARRAY in selected or deselected state, depending on the FLAG value, correspondingly 1 or 0.

Only for the multi-select mode.

**deselect_all**

Clears the selection

Only for the multi-select mode.

**draw_items CANVAS, ITEM_DRAW_DATA**

Called from within the `Paint` notification to draw items. The default behavior is to call the `DrawItem` notification for every item in the ITEM_DRAW_DATA array. ITEM_DRAW_DATA is an array or arrays, where each array consists of parameters passed to the `DrawItem` notification.

This method is overridden in some descendant classes to increase the speed of drawing. For example, `std_draw_text_items` is the optimized routine for drawing text-based items. It is used in the `Prima::ListBox` class.

See the *DrawItem* entry for the description of the parameters.

**draw_text_items CANVAS, FIRST, LAST, STEP, X, Y, OFFSET, CLIP_RECT**

Called by `std_draw_text_items` to draw a sequence of text items with indices from FIRST to LAST, by STEP, on CANVAS, starting at point X, Y, and incrementing the vertical position with OFFSET. CLIP_RECT is a reference to an array of four integers given in the inclusive-inclusive coordinates that represent the active clipping rectangle.

Note that OFFSET must be an integer, otherwise bad effects will be observed when text is drawn below Y=0

**get_item_text INDEX**

Returns the text string assigned to the INDEXth item. Since the class does not assume the item storage organization, the text is queried via the `Stringify` notification.

**get_item_width INDEX**

Returns width in pixels of the INDEXth item. Since the class does not assume the item storage organization, the value is queried via the `MeasureItem` notification.

**is_selected INDEX**

Returns 1 if the INDEXth item is selected, 0 otherwise.

**item2rect INDEX, [ WIDTH, HEIGHT ]**

Calculates and returns four integers with rectangle coordinates of the INDEXth item. WIDTH and HEIGHT are optional parameters with pre-fetched dimensions of the widget; if not set, the dimensions are queried by calling the `size` property. If set, however, the `size` property is not called, thus some speed-up can be achieved.

**point2item X, Y**

Returns the index of an item that contains the point at (X,Y). If the point belongs to the item outside the widget's interior, returns the index of the first item outside the widget's interior in the direction of the point.

**redraw_items INDICES**

Redraws all items in the INDICES array.

**select_all**

Selects all items.

Only for the multi-select mode.

**set_item_selected INDEX, FLAG**

Sets the selection flag on the INDEXth item. If FLAG is 1, the item is selected. If 0, it is deselected.

Only for the multi-select mode.

**select_item INDEX**

Selects the INDEXth item.

Only for the multi-select mode.

**std_draw_text_items CANVAS, ITEM_DRAW_DATA**

An optimized method, draws text-based items. It is fully compatible with the `draw_items` interface and is used in the `Prima::ListBox` class.

The optimization is derived from the assumption that items maintain common background and foreground colors, that only differ in the selected and non-selected states. The routine groups drawing requests for selected and non-selected items, and then draws items with a reduced number of calls to the `color` property. While the background is drawn by the routine itself, the foreground ( usually text ) is delegated to the `draw_text_items` method, so that the text positioning and eventual decorations would be easier to implement.

ITEM_DRAW_DATA is an array of arrays of scalars, where each array contains parameters of the `DrawItem` notification. See the *DrawItem* entry for the description of the parameters.

**toggle_item INDEX**

Toggles selection of the INDEXth item.

Only for the multi-select mode.

**unselect_item INDEX**

Deselects the INDEXth item.

Only for the multi-select mode.

### Events

#### Click

Called when the user presses the return key or double-clicks on an item. The index of the item is stored in `focusedItem`.

#### DragItem OLD_INDEX, NEW_INDEX

Called when the user finishes the drag of an item from OLD_INDEX to NEW_INDEX position. The default action rearranges the item list according to the dragging action.

#### DrawItem CANVAS, INDEX, X1, Y1, X2, Y2, SELECTED, FOCUSED, PRELIGHT, COLUMN

Called when the INDEXth item is to be drawn on CANVAS. X1, Y1, X2, Y2 define the item rectangle in widget coordinates where the item is to be drawn. SELECTED, FOCUSED, and PRELIGHT are boolean flags, if the item must be drawn correspondingly in selected and focused states, with or without the prelight effect.

#### MeasureItem INDEX, REF

Stores width in pixels of the INDEXth item into the REF scalar reference. This notification must be called from within the `begin_paint_info/end_paint_info` block.

#### SelectItem INDEX, FLAG

Called when an item changes its selection state. INDEX is the index of the item, FLAG is its new selection state: 1 if it is selected, 0 if it is not.

#### Stringify INDEX, TEXT_REF

Stores the text string associated with the INDEXth item into TEXT_REF scalar reference.

## Prima::AbstractListBox

The same as its ascendant `Prima::AbstractListViewer` except that it does not the propagate `DrawItem` message, assuming that all items must be drawn as text strings.

## Prima::ListViewer

The class implements an item storage mechanism but leaves the definition of the format of the item to the programmer. The items are accessible via the `items` property and several other helper routines.

The class also defines user navigation by accepting character keyboard input and jumping to the items that have text assigned with the first letter that matches the input.

`Prima::ListViewer` is derived from `Prima::AbstractListViewer`.

### Properties

#### autoWidth BOOLEAN

Selects if the item width must be recalculated automatically when either the font or item list is changed.

Default value: 1

#### count INTEGER

A read-only property; returns the number of items.

#### items ARRAY

Accesses the storage array of the items. The format of items is not defined, it is merely treated as one scalar per index.

### Methods

**add_items ITEMS**

Appends an array of ITEMS to the end of the item list.

**calibrate**

Recalculates all item widths. Adjusts `itemWidth` if `autoWidth` is set.

**delete_items INDICES**

Deletes items from the list. INDICES can be either an array or a reference to an array of item indices.

**get_item_width INDEX**

Returns the width in pixels of the INDEXth item from the internal cache.

**get_items INDICES**

Returns an array of items. INDICES can be either an array or a reference to an array of item indices. Depending on the caller context, the results are different: in the array context the item list is returned; in scalar - only the first item from the list.

**insert_items OFFSET, ITEMS**

Inserts an array of items at the OFFSET index in the list. Offset must be a valid index; to insert items at the end of the list use the `add_items` method.

ITEMS can be either an array or a reference to an array of items.

**replace_items OFFSET, ITEMS**

Replaces existing items at the OFFSET index in the list. The offset must be a valid index.

ITEMS can be either an array or a reference to an array of items.

## Prima::ProtectedListBox

A semi-demonstrational class derived from `Prima::ListViewer`, implements certain protections for every item during drawing. Assuming that several item drawing algorithms can be used in the same widget, `Prima::ProtectedListBox` provides a safety layer between these. If an algorithm selects a font or a color and does not restore the old value, this does not affect the outlook of other items.

This functionality is implemented by overloading the `draw_items` method and also all graphic properties.

## Prima::ListBox

Descendant of `Prima::ListViewer`, declares that an item must be a single text string. Incorporating all the functionality of its predecessors, provides the standard workhorse listbox widget.

### Synopsis

```
my $lb = Prima::ListBox-> create(
        items       => [qw(First Second Third)],
        focusedItem => 2,
        onClick     => sub {
                print $_[0]-> get_items( $_[0]-> focusedItem), " is selected\n";
        }
);
```

### Methods

**get_item_text INDEX**

Returns the text string associated with the INDEXth item. Since the item storage organization is implemented, does so without calling the `Stringify` notification.

## 4.19 Prima::MDI

Top-level window emulation

### Description

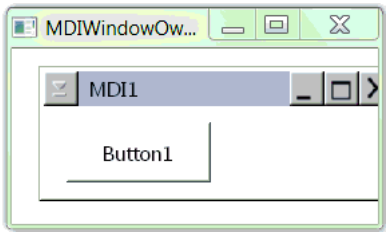
MDI stands for Multiple Document Interface and is a Microsoft Windows user interface that consists of multiple non-top-level windows belonging to an application window. The module contains classes that provide similar functionality; sub-window widgets realize a set of operations similar to those of the real top-level windows, - iconize, maximize, cascade, etc.

The basic classes required to use the MDI are `Prima::MDIOwner` and `Prima::MDI`, which are, correspondingly, sub-window owner class and sub-window class. `Prima::MDIWindowOwner` is the same as `Prima::MDIOwner` but is a `Prima::Window` descendant: both owner classes are different only in the class they are derived from. Their second ascendant is the `Prima::MDIMethods` package that contains all the owner class functionality.

Usage of the `Prima::MDI` class extends beyond the multi-document paradigm. The `Prima::DockManager` module uses the class as a base of the dockable toolbar window class (see the *Prima::DockManager* section.

### Synopsis

```
use Prima qw(Application MDI Buttons);

my $owner = Prima::MDIWindowOwner-> new;
my $mdi   = $owner-> insert( 'Prima::MDI');
$mdi-> client-> insert( 'Prima::Button' => centered => 1 );

run Prima;
```




### Prima::MDI

Implements MDI window functionality. A subwindow widget consists of a title bar, title bar buttons, and a client widget. The latter must be used as an insertion target for all children widgets.

A subwindow can be moved and resized, both by mouse and keyboard. These functions, along with maximize, minimize, and restore commands are accessible via the popup menu anchored to the window toolbar. The default set of commands is as follows

```
Close window                 - Ctrl+F4
Restore window               - Ctrl+F5 or double-click on the title bar
Maximize window              - Ctrl+F10 or double-click on the title bar
Go to the next MDI window    - Ctrl+Tab
Go to the previous MDI window - Ctrl+Shift+Tab
Invoke popup menu            - Ctrl+Space
```

The class mimics the API of the `Prima::Window` class, and to some extent, the *Prima::Window* section and this page share the same information.

### Properties

#### borderIcons INTEGER

Manages window decorations, which are buttons on the title bar and the title bar itself. Can be 0 or a combination of the following `mbi::XXX` constants that are a superset of the `bi::XXX` constants ( see the **borderIcons** entry in the *Prima::Window* section ) and are interchangeable.

```
mbi::SystemMenu  - system menu button with an icon is shown
mbi::Minimize    - minimize button
mbi::Maximize    - maximize and restore buttons
mbi::TitleBar    - window title
mbi::Close       - close button
mbi::All         - all of the above
```

Default value: `mbi::All`

#### borderStyle INTEGER

One of the `bs::XXX` constants that define the window border style:

```
bs::None      - no border
bs::Single    - thin border
bs::Dialog    - thick border
bs::Sizeable  - thick border with interactive resize capabilities
```

The `bs::Sizeable` is a unique mode. If selected, the user can resize the window interactively. The other border styles disallow resizing and affect the border width and design only.

Default value: `bs::Sizeable`

#### client OBJECT

Selects the client widget at run time. When changing the client, the old client's children are not reparented to the new client. The property cannot be used to set the client during the MDI window creation; use the `clientClass` and `clientProfile` properties instead.

When setting a new client object, note that it has to be named `MDIClient`, and that the window will be automatically destroyed after the client is destroyed.

#### clientClass STRING

Assigns the client widget class.

Create-only property.

Default value: `Prima::Widget`

#### clientProfile HASH

Assigns a hash of properties passed to the client during the creation.

Create-only property.

#### dragMode SCALAR

A three-state variable that manages the visual feedback style when the user moves or resizes a window. If 1, the window is moved or resized simultaneously with the user's mouse or keyboard actions. If 0, a marquee rectangle is drawn, which is moved or resized as the user sends the commands; the window is only positioned and/or resized after the dragging session is successfully finished. If `undef`, the system-dependant dragging style is used. ( See the **get_system_value** entry in the *Prima::Application* section ).

The dragging session can be aborted by hitting the Esc key or calling the `sizemove_cancel` method.

Default value: `undef`.

**icon HANDLE**

Selects a custom image to be drawn in the left corner of the toolbar. If 0, the default image ( menu button icon ) is drawn.

Default value: 0

**iconMin HANDLE**

Selects the minimized button image in the normal state.

**iconMax HANDLE**

Selects the maximized button image in the normal state.

**iconClose HANDLE**

Selects the close button image in the normal state.

**iconRestore HANDLE**

Selects the restore button image in the normal state.

**iconMinPressed HANDLE**

Selects the minimize button image in the pressed state.

**iconMaxPressed HANDLE**

Selects the maximize button image in the pressed state.

**iconClosePressed HANDLE**

Selects the close button image in the pressed state.

**iconRestorePressed HANDLE**

Selects the restore button image in the pressed state.

**tileable BOOLEAN**

Selects whether the window is allowed to participate in cascading and tiling auto-arrangements, performed correspondingly by the `cascade` and `tile` methods. If 0, the window position is not affected by these methods.

Default value: 1

**titleHeight INTEGER**

Selects the height of the title bar in pixels. If 0, the default system value is used.

Default value: 0

**windowState STATE**

A three-state property that manages the state of a window. STATE can be one of three `ws::XXX` constants:

```
ws::Normal
ws::Minimized
ws::Maximized
```

The property can be changed either by an explicit set-mode call or by the user. In either case, a `WindowState` notification is triggered.

The property has three convenience wrappers: `maximize()`, `minimize()`, and `restore()`.

The `ws::Fullscreen` constant is not supported, and there's no corresponding `fullscreen()` method.

Default value: `ws::Normal`

See also: `WindowState`

### Methods

**arrange_icons**

Arranges geometrically the minimized sibling MDI windows.

**cascade**

Arranges sibling MDI windows so they form a cascade-like structure: the lowest window is expanded to the full owner window inferior rectangle, the next window occupies the inferior rectangle of the first window, etc.

Only windows with the `tileable` property set to 1 are arranged.

**client2frame X1, Y1, X2, Y2**

Returns a rectangle that the window would occupy if its client rectangle is assigned to the X1, Y1, X2, Y2 rectangle.

**frame2client X1, Y1, X2, Y2**

Returns a rectangle that the window client would occupy if the window rectangle is assigned to the X1, Y1, X2, Y2 rectangle.

**get_client_rect [ WIDTH, HEIGHT ]**

Returns a rectangle in the window coordinate system that the client would occupy if the window extensions are WIDTH and HEIGHT. If WIDTH and HEIGHT are undefined, the current window size is used.

**keyMove**

Initiates window moving session, navigated by the keyboard.

**keySize**

Initiates window resizing session, navigated by the keyboard.

**mdis**

Returns an array of sibling MDI windows.

**maximize**

Maximizes the window. A shortcut for `windowState(ws::Maximized)`.

**minimize**

Minimizes the window. A shortcut for `windowState(ws::Minimized)`.

**post_action STRING**

Posts an action to the window; the action is deferred and executed in the next message loop. This is used to avoid unnecessary state checks when the action-executing code returns. The current implementation accepts the following string commands: `min`, `max`, `restore`, `close`.

**repaint_title [ STRING = `title` ]**

Invalidates the part of the title bar corresponding to the STRING, which can be one of the following:

```
left    - redraws the menu button
right   - redraws minimize, maximize, and close buttons
title   - redraws the title
```

**restore**

Restores the window to the normal state from the minimized or maximized state. A shortcut for `windowState(ws::Normal)`.

**sizemove_cancel**

Cancels active moving or resizing session and returns the window to the previous state

**tile**

Arranges sibling MDI windows so they form a grid-like structure where all windows occupy equal space, if possible.

Only windows with the `tileable` property set to 1 are processed.

**xy2part X, Y**

Maps a point in the (X,Y) coordinates into a string corresponding to the part of the window: title bar, button, or a part of the border. The latter can be returned only if `borderStyle` is set to `bs::Sizeable`. The possible return values are:

```
border   - window border; the window is not sizeable
client   - client widget
caption  - title bar; the window is not movable
title    - title bar; the window is movable
close    - close button
min      - minimize button
max      - maximize button
restore  - restore button
menu     - menu button
desktop  - the point does not belong to the window
```

In addition, if the window is sizeable, the following constants can be returned, indicating the part of the border:

```
SizeN    - upper side
SizeS    - lower side
SizeW    - left side
SizeE    - right side
SizeSW   - lower left corner
SizeNW   - upper left corner
SizeSE   - lower right corner
SizeNE   - upper right corner
```

### Events

#### Activate

Triggered when the user activates the window. The activation mark usually resides on the window that has the keyboard focus.

The module does not provide a dedicated activation function; the `select()` call can be used for this.

#### Deactivate

Triggered when the user deactivates the window. A window is usually marked inactive when it contains no keyboard focus.

The module does not provide a dedicated de-activation function; the `deselect()` call can be used instead.

#### WindowState STATE

Triggered when the window state is changed, either by an explicit `windowState()` call or by the user. STATE is the new window state, one of three `ws::XXX` constants.

## Prima::MDIMethods

### Methods

The package contains methods for MDI window owners. Add `Prima::MDIMethods` as a base to your class to inherit this functionality if neither `Prima::MDIOwner` nor `Prima::MDIWindowOwner` suit your needs.

#### mdi_activate

Repaints window titles in all children MDI windows.

#### mdis

Returns an array of children MDI windows.

#### arrange_icons

The same as `Prima::MDI::arrange_icons`.

#### cascade

The same as `Prima::MDI::cascade`.

#### tile

The same as `Prima::MDI::tile`.

## Prima::MDIOwner

A predeclared descendant class derived from `Prima::Widget` and `Prima::MDIMethods`.

## Prima::MDIWindowOwner

A pre-declared descendant class derived from `Prima::Window` and `Prima::MDIMethods`.

## SEE ALSderived from

the *Prima* section, the *Prima::Widget* section, the *Prima::Window* section, the *Prima::DockManager* section, *examples/mdi.pl*

## 4.20 Prima::Notebooks

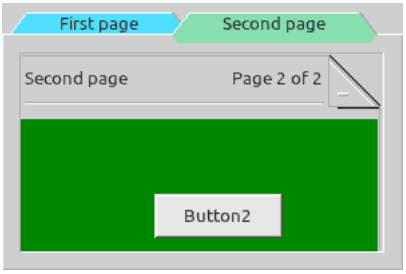
Multipage widgets

### Description

The module contains several widgets useful for organizing multipage containers, *notebooks*. `Prima::Notebook` provides the basic functionality of such a widget container. `Prima::TabSet` is a page selector control, and `Prima::TabbedNotebook` combines these two into a ready-to-use multipage control with interactive navigation.

### Synopsis

```
use Prima qw(Notebooks Buttons Application);
my $nb = Prima::TabbedNotebook-> new(
        tabs => [ 'First page', 'Second page', 'Second page' ],
        size => [ 300, 200 ],
);
$nb-> insert_to_page( 1, 'Prima::Button' );
$nb-> insert_to_page( 2,
        [ 'Prima::Button', bottom => 10  ],
        [ 'Prima::Button', bottom => 150 ],
);
$nb-> Notebook-> backColor( cl::Green );
run Prima;
```




### Prima::Notebook

#### Properties

Provides basic widget container functionality. Acts as a merely grouping widget, hiding and showing the children widgets when the `pageIndex` property is changed.

**defaultInsertPage INTEGER**

Selects the page where widgets attached the by `insert` call are assigned. If set to `undef`, the default page is the current page.

Default value: `undef`.

**pageCount INTEGER**

Selects the number of pages. If the number of pages is reduced, the widgets that belong to the rejected pages are removed from the notebook's storage.

**pageIndex INTEGER**

Selects the index of the current page. Valid values are from 0 to `pageCount - 1`.

### Methods

**attach_to_page INDEX, @WIDGETS**

Attaches WIDGETS to INDEXth page. The widgets not necessarily must be children of the notebook widget. If the INDEXth page is not current, the widgets get hidden and disabled; otherwise their state is not changed.

**contains_widget WIDGET**

Searches for WIDGET in the attached widgets list. If found, returns two integers: location page index and widget list index. Otherwise returns an empty list.

**delete_page [ INDEX = -1, REMOVE_CHILDREN = 1 ]**

Deletes the INDEXth page, and detaches the widgets associated with it. If REMOVE_CHILDREN is 1, the detached widgets are destroyed.

**delete_widget WIDGET**

Detaches WIDGET from the widget list and destroys the widget.

**detach_from_page WIDGET**

Detaches WIDGET from the widget list.

**insert CLASS, %PROFILE [[ CLASS, %PROFILE], ... ]**

Creates one or more widgets with the `owner` property set to the caller widget, and returns the list of references to the newly created objects.

See the **insert** entry in the *Prima::Widget* section for details.

**insert_page [ INDEX = -1 ]**

Inserts a new empty page at INDEX. The valid range is from 0 to `pageCount`; setting INDEX equal to `pageCount` is equivalent to appending a page to the end of the page list.

**insert_to_page INDEX, CLASS, %PROFILE, [[ CLASS, %PROFILE], ... ]**

Inserts one or more widgets to the INDEXth page. The semantics of setting CLASS and PROFILE, as well as the return values are fully equivalent to the `insert` method.

See the **insert** entry in the *Prima::Widget* section for details.

**insert_transparent CLASS, %PROFILE, [[ CLASS, %PROFILE], ... ]**

Inserts one or more widgets to the notebook widget, but does not add widgets to the widget list, so the widgets are not flipped together with pages. Useful for setting omnipresent ( or *transparent* ) widgets, visible on all pages.

The semantics of setting CLASS and PROFILE, as well as the return values are fully equivalent to the `insert` method.

See the **insert** entry in the *Prima::Widget* section for details.

**move_widget WIDGET, INDEX**

Moves WIDGET to the INDEXth page.

**widget_get WIDGET, PROPERTY**

Returns PROPERTY value of WIDGET. If PROPERTY is affected by the page flipping mechanism, the internal flag value is returned instead.

**widget_set WIDGET, %PROFILE**

Calls `set` on WIDGET with PROFILE and updates the internal `visible`, `enabled`, `current`, and `geometry` properties if these are present in PROFILE.

See the **set** entry in the *Prima::Object* section.

**widgets_from_page INDEX**

Returns list of widgets associated with the INDEXth page.

### Events

**Change OLD_PAGE_INDEX, NEW_PAGE_INDEX**

Called when the `pageIndex` value is changed from OLD_PAGE_INDEX to NEW_PAGE_INDEX. Current implementation invokes this notification while the notebook widget is in the locked state so no redraw requests are honored during the execution of the notification.

### Bugs

Since the notebook operates directly on children widgets' `::visible` and `::enable` properties, there is a problem when a widget associated with a non-active page must be explicitly hidden or disabled. As a result, such a widget would become visible and enabled anyway. This happens because Prima API does not cache property requests. For example, after the execution of the following code

```
$notebook-> pageIndex(1);
my $widget = $notebook-> insert_to_page( 0, ... );
$widget-> visible(0);
$notebook-> pageIndex(0);
```

`$widget` will still become visible. As a workaround, the `widget_set` method can be suggested, to be called together with the explicit state calls. Changing the

```
$widget-> visible(0);
```

code to

```
$notebook-> widget_set( $widget, visible => 0);
```

solves the problem, but introduces an inconsistency in API.

## Prima::TabSet

The `Prima::TabSet` class implements the functionality of an interactive page switcher. A widget is presented as a set of horizontal bookmark-styled tabs with text identifiers.

### Properties

**colored BOOLEAN**

A boolean property, selects whether each tab uses unique color ( OS/2 Warp 4 style ), or all tabs are drawn with `backColor`.

Default value: 1

**colorset ARRAY**

Allows to specify custom colors for the tabs.

Used only when `colored` is set to 1.

**firstTab INTEGER**

Selects the first ( leftmost ) visible tab.

**focusedTab INTEGER**

Selects the currently focused tab. This property value is almost always equal to `tabIndex` except when the widget is navigated by arrow keys, and the tab selection does not occur until the user presses the return key.

**topMost BOOLEAN**

Selects the way the widget is oriented. If 1, the widget is drawn as if it resides on top of another widget. If 0, it is drawn as if it is at the bottom.

Default value: 1

**tabIndex INDEX**

Selects the INDEXth tab. When changed, the `Change` notification is triggered.

**tabs ARRAY**

An array of text scalars. Each scalar corresponds to a tab and is displayed correspondingly. The class supports single-line text strings only; newline characters are not respected.

### Methods

**get_item_width INDEX**

Returns width in pixels of the INDEXth tab.

**tab2firstTab INDEX**

Returns the index of the tab that will be drawn leftmost if the INDEXth tab is to be displayed.

**insert_tab TEXT, [ POSITION = -1 ]**

Inserts a new tab text at the given position, which is at the end by default

**delete_tab POSITION**

Removes the tab from the given position

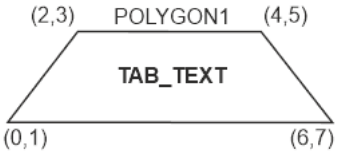
### Events

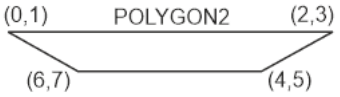
**Change**

Triggered when the `tabIndex` property is changed.

**DrawTab CANVAS, INDEX, COLOR_SET, POLYGON1, POLYGON2**

Called when the INDEXth tab is to be drawn on CANVAS. COLOR_SET is an array reference that consists of four cached color values: foreground, background, dark 3d color, and light 3d color. POLYGON1 and POLYGON2 are array references that contain four points as integer pairs in (X,Y)-coordinates. POLYGON1 keeps the coordinates of the larger polygon of a tab, while POLYGON2 of the smaller. Text is displayed inside the larger polygon:



Depending on the `topMost` property value, POLYGON1 and POLYGON2 change their mutual vertical orientation.

The notification is always called from within the `begin_paint/end_paint` block.

**MeasureTab INDEX, REF**

Stores the width of the INDEXth tab in pixels into the REF scalar value. This notification must be called from within the `begin_paint_info/end_paint_info` block.

## Prima::TabbedNotebook

The class combines the functionality of `Prima::TabSet` and `Prima::Notebook`, providing the interactive multipage widget functionality. The page indexing scheme has two levels: the first level is equivalent to the tabs provided by `Prima::TabSet`. Each first-level tab, in turn, may contain one or more second-level pages, which can be switched using native `Prima::TabbedNotebook` controls.

The first-level tabs are referred to as *tabs*, and the second-level as *pages*.

### Properties

The class forwards the following properties of `Prima::TabSet`, which are described in the *Prima::TabSet* section: `colored`, `colorset`

**defaultInsertPage INTEGER**

Selects the page where widgets attached by the `insert` call are assigned to. If set to `undef`, the default page is the current page.

Default value: `undef`.

**notebookClass STRING**

Assigns the notebook widget class.

Create-only property.

Default value: `Prima::Notebook`

**notebookProfile HASH**

Assigns a hash of properties passed to the notebook widget during the creation.

Create-only property.

**notebookDelegations ARRAY**

Assigns a list of delegated notifications to the notebook widget.

Create-only property.

**orientation INTEGER**

Selects one of the following tno::XXX constants

**tno::Top**

The TabSet will be drawn at the top of the widget.

**tno::Bottom**

The TabSet will be drawn at the bottom of the widget.

Default value: tno::Top

**pageIndex INTEGER**

Selects the INDEXth page or a tabset widget ( the second-level tab ). When this property is triggered, `tabIndex` can change its value, and the `Change` notification is triggered.

**style INTEGER**

Selects one of the following tns::XXX constants

**tns::Standard**

The widget will have a raised border surrounding it and a +/- control at the top for moving between pages.

**tns::Simple**

The widget will have no decorations (other than a standard border). It is recommended to have only one second-level page per tab with this style.

Default value: tns::Standard

**tabIndex INTEGER**

Selects the INDEXth tab on the tabset widget using the first-level tab numeration.

**tabs ARRAY**

Manages the number and names of notebook pages. ARRAY is an anonymous array of text scalars where each corresponds to a single first-level tab and a single notebook page, however, with a single exception. To define second-level tabs, the same text string must be repeated as many times as many second-level tabs are needed. For example, the code

```
$nb-> tabs('1st', ('2nd') x 3);
```

results in the creation of a notebook of four pages and two first-level tabs. The tab `'2nd'` contains three second-level pages.

The property implicitly operates the underlying notebook's `pageCount` property. When changed at run-time, its effect on the children widgets is therefore the same. See the *pageCount* entry for more information.

**tabsetClass STRING**

Assigns the tab set widget class.

Create-only property.

Default value: `Prima::TabSet`

**tabsetProfile HASH**

Assigns a hash of properties passed to the tab set widget during the creation.

Create-only property.

**tabsetDelegations ARRAY**

Assigns a list of delegated notifications to the tab set widget.

Create-only property.

### Methods

The class forwards the following methods of `Prima::Notebook`, which are described in the *Prima::Notebook* section: `attach_to_page`, `insert_to_page`, `insert`, `insert_transparent`, `delete_widget`, `detach_from_page`, `move_widget`, `contains_widget`, `widget_get`, `widget_set`, `widgets_from_page`.

**tab2page INDEX**

Returns the second-level tab index that corresponds to the INDEXth first-level tab.

**page2tab INDEX**

Returns the first-level tab index that corresponds to the INDEXth second-level tab.

**insert_page TEXT, [ POSITION = -1 ]**

Inserts a new page with text at the given position, which is at the end by default. If the TEXT is the same as the existing tab left or right from POSITION, the page is joined to the existing tab as a page; otherwise, a new tab is created.

**delete_page POSITION**

Removes the page from the given position.

### Events

**Change OLD_PAGE_INDEX, NEW_PAGE_INDEX**

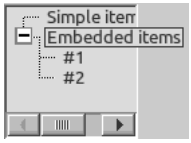
Triggered when the `pageIndex` property changes its value from OLD_PAGE_INDEX to NEW_PAGE_INDEX.

## 4.21 Prima::Outlines

Tree view widgets

### Synopsis

```
use Prima qw(Outlines Application);
my $outline = Prima::StringOutline-> create(
        items => [
                [ 'Simple item' ],
                [ 'Embedded items', [['#1'], ['#2']]],
        ],
);
$outline-> expand_all;
run Prima;
```




```
my $outline = Prima::StringOutline-> create(
        iconStyle => 'triangle',
        ...
);
```


### Description

The module provides a set of widget classes designed to display tree-like structures. `Prima::OutlineViewer` presents a generic class that contains the basic functionality and defines the interface for the class descendants, which are `Prima::StringOutline`, `Prima::Outline`, and `Prima::DirectoryOutline`.

### Prima::OutlineViewer

Presents a generic interface for browsing tree-like lists. Each node in a linked list represents an item. The format of the node is predefined, and is an anonymous array with the following definitions of its indices:

**#0**

Item id in an unspecified format. The simplest implementation, `Prima::StringOutline`, treats the scalar as a text string. The more complex classes store references to arrays or hashes here. See the `items` article of a concrete class for the format of the node record.

**#1**

Reference to a child node. `undef` if there is none.

**#2**

A boolean flag, which selects if the node is to be shown as expanded, e.g. all of its immediate children are visible.

**#3**

Width of an item in pixels.

The indices above 3 should not be used because eventual changes to the implementation of the class may use these. The general idea is that the data at index #0 should be self-sufficient to define an item.

To support a custom format of the node the following notifications should be overloaded: `DrawItem`, `MeasureItem`, and `Stringify`. Since `DrawItem` is called for every item, a gross method `draw_items` can be overloaded instead. See also the *Prima::StringOutline* section and the *Prima::Outline* section.

The class employs two ways to address an item, index-wise and item-wise. The index-wise counts only the visible ( non-expanded ) items and is represented by an integer index. The item-wise addressing cannot be expressed by an integer index, and the full node structure is used as a reference. It is important to use a valid reference here since the class does not always perform the check if the node belongs to the internal node list due to speed reasons.

`Prima::OutlineViewer` is a descendant of `Prima::Widget::GroupScroller` and `Prima::Widget::MouseScroller`, so some of their properties and methods are not described here.

The class is not usable directly.

### Properties

**autoHeight INTEGER**

If set to 1, changes `itemHeight` automatically according to the widget font height. If 0, does not influence anything. When `itemHeight` is set explicitly, changes value to 0.

Default value: 1

**dragable BOOLEAN**

If 1, allows the items to be dragged interactively by pressing the Control key together with the left mouse button. If 0, item dragging is disabled.

Default value: 1

**drawLines BOOLEAN**

If 1, draws dotted tree lines left to the items.

Default value: 1

**extendedSelect BOOLEAN**

Manages the way the user selects multiple items and is only actual when `multiSelect` is 1. If 0, the user must click each item to mark it as selected. If 1, the user can drag the mouse or use the `Shift` key plus arrow keys to perform range selection; the `Control` key can be used to select individual items.

Default value: 0

**focusedItem INTEGER**

Selects the focused item index. If -1, no item is focused. It is mostly a run-time property, however, it can be set during the widget creation stage given that the item list is accessible at this stage as well.

**iconCollapsed ICON**

Sets the image that is to be displayed when a tree branch is collapsed

**iconExpanded ICON**

Sets the image that is to be displayed when a tree branch is expanded

**iconStyle STYLE**

Sets visual style, one of: `default`, `plusminus`, or `triangle`.

The default style is set in `$Prima::Outlines::default_style` and is currently 'plusminus', however, it can be overridden by `skin`. The default style for the current default skin `flat` is 'triangle'.

**indent INTEGER**

Width in pixels of the indent between item levels.

Default value: 12

**itemHeight INTEGER**

Selects the height of the items in pixels. Since the outline classes do not support items with various heights, changes to this property affect all items.

Default value: default font height

**items ARRAY**

Provides access to the items as an anonymous array. The format of an item is described in the opening article ( see the *Prima::OutlineViewer* section ).

Default value: []

**multiSelect BOOLEAN**

If 0, the user can only select one item, which is also reported by the `focusedItem` property. If 1, the user can select more than one item. In this case, the `focusedItem`'th item is not necessarily selected. To access the selected item list, use the `selectedItems` property.

Default value: 0

**offset INTEGER**

Horizontal offset of the item list in pixels.

**selectedItems ARRAY**

ARRAY is an array of integer indices of the selected items. Note, that these are the items visible on the screen only. The property doesn't handle the selection status of the collapsed items.

The widget keeps the selection status on each node, visible and invisible ( e.g. the node is invisible if its parent node is collapsed). However, `selectedItems` accounts for the visible nodes only; to manipulate the node status or both visible and invisible nodes, use `select_item`, `unselect_item`, and `toggle_item` methods.

**showItemHint BOOLEAN**

If 1, allows activation of a hint label when the mouse pointer is hovered above an item that does not fit horizontally into the widget inferiors. If 0, the hint is never shown.

See also: the *makehint* entry.

Default value: 1

**topItem INTEGER**

Selects the first item drawn.

### Methods

**add_selection ARRAY, FLAG**

Sets item indices from ARRAY in selected or deselected state, depending on the FLAG value, correspondingly 1 or 0.

Note, that these are the items visible on the screen only. The method doesn't handle the selection status of the collapsed items.

Only for the multi-select mode.

**adjust INDEX, EXPAND**

Performs expansion ( 1 ) or collapse ( 0 ) of the INDEXth item, depending on the EXPAND boolean flag value.

**calibrate**

Recalculates the node tree and item dimensions. Used internally.

**delete_items [ NODE = undef, OFFSET = 0, LENGTH = undef ]**

Deletes LENGTH children items of NODE at OFFSET. If NODE is `undef`, the root node is assumed. If LENGTH is `undef`, all items after OFFSET are deleted.

**delete_item NODE**

Deletes NODE from the item list.

**deselect_all**

Removes selection from all items.

Only for multi-select mode.

**draw_items CANVAS, PAINT_DATA**

Called from within the `Paint` notification to draw items. The default behavior is to call the `DrawItem` notification for every visible item. PAINT_DATA is an array of arrays, where each consists of the parameters passed to the `DrawItem` notification.

This method is overridden in some descendant classes, to increase the speed of the drawing routine.

See the *DrawItem* entry for PAINT_DATA parameters description.

**get_index NODE**

Traverses all items for NODE and finds if it is visible. If it is, returns two integers: the first is the item index and the second is the item depth level. If the node is not visible, `-1, undef` is returned.

**get_index_text INDEX**

Returns the text string associated with the INDEXth item. Since the class does not assume the item storage organization, the text is queried via the `Stringify` notification.

**get_index_width INDEX**

Returns the width in pixels of the INDEXth item, which is a cached result of the `MeasureItem` notification, stored under index #3 in a node.

**get_item INDEX**

Returns two scalars corresponding to the INDEXth item: the node reference and its depth level. If INDEX is outside the list boundaries, an empty array is returned.

**get_item_parent NODE**

Returns two scalars, corresponding to the NODE: its parent node reference and offset of the NODE in the parent's immediate children list.

**get_item_text NODE**

Returns the text string associated with the NODE. Since the class does not assume the item storage organization, the text is queried via the `Stringify` notification.

**get_item_width NODE**

Returns width in pixels of the INDEXth item, which is a cached result of the `MeasureItem` notification, stored under index #3 in a node.

**expand_all [ NODE = undef ].**

Expands all nodes under NODE. If NODE is `undef` the root node is assumed. If the tree is large, the execution can take a significant amount of time.

**insert_items NODE, OFFSET, @ITEMS**

Inserts one or more ITEMS under NODE with OFFSET. If NODE is `undef`, the root node is assumed.

**iterate ACTION, FULL**

Traverses the item tree and calls the ACTION subroutine on each node. If the FULL boolean flag is 1, all nodes are traversed. If 0, only the expanded nodes are traversed.

ACTION subroutine is called with the following parameters:

**#0**

Node reference

**#1**

Parent node reference; if `undef`, the node is the root.

**#2**

Node offset in the parent item list.

**#3**

Node index.

**#4**

Node depth level. 0 means the root node.

**#5**

A boolean flag, set to 1 if the node is the last child in the parent node list, set to 0 otherwise.

**#6**

Visibility index. When `iterate` is called with `FULL = 1`, the index is the item index as seen on the screen. If the item is not visible, the index is `undef`.

When `iterate` is called with `FULL = 1`, the index is always the same as `node index`.

**is_selected INDEX, ITEM**

Returns 1 if an item is selected, 0 if it is not.

The method can address the item either directly ( ITEM ) or by its INDEX in the screen position.

**makehint SHOW, INDEX**

Controls hint label of the INDEXth item. If a boolean flag SHOW is set to 1, the `showItemHint` property is 1, and the item index does not fit horizontally in the widget inferiors, then the hint label is shown. By default, the label is removed automatically as soon as the user moves the mouse pointer away from the item. If SHOW is set to 0, the hint label is hidden immediately.

**point2item Y, [ HEIGHT ]**

Returns the index of an item that occupies the horizontal axis at Y in the widget coordinates. If HEIGHT is specified, it must be the widget height; if it is not, the value is fetched by calling `Prima::Widget::height`. If the value is known, passing it to `point2item` thus achieves some speed-up.

**select_all**

Selects all items.

Only for multi-select mode.

**set_item_selected INDEX, ITEM, FLAG**

Sets the selection flag of an item. If FLAG is 1, the item is selected. If 0, it is deselected.

The method can address the item either directly ( ITEM ) or by its INDEX. Only for the multi-select mode.

**select_item INDEX, ITEM**

Selects an item.

The method can address the item either directly ( ITEM ) or by its INDEX. Only for the multi-select mode.

**toggle_item INDEX, ITEM**

Toggles selection of an item.

The method can address the item either directly ( ITEM ) or by its INDEX. Only for the multi-select mode.

**unselect_item INDEX, ITEM**

Deselects an item.

The method can address the item either directly ( ITEM ) or by its INDEX. Only for the multi-select mode.

**validate_items ITEMS**

Traverses an array of ITEMS and changes every node so that eventual scalars above index #3 are deleted. Also adds default values to a node if it contains less than 3 scalars.

### Events

**Expand NODE, EXPAND**

Called when NODE is expanded ( 1 ) or collapsed ( 0 ). The EXPAND boolean flag reflects the action taken.

**DragItem OLD_INDEX, NEW_INDEX**

Called when the user finishes the drag of an item from OLD_INDEX to NEW_INDEX position. The default action rearranges the item list according to the dragging action.

**DrawItem CANVAS, NODE, X1, Y1, X2, Y2, INDEX, SELECTED, FOCUSED, PRELIGHT**

Called when the INDEXth item contained in NODE is to be drawn on CANVAS. X1, Y1, X2, Y2 coordinates define the exterior rectangle of the item in widget coordinates. SELECTED, FOCUSED, and PRELIGHT boolean flags are set to 1 if the item is selected, focused, or pre-lighted, respectively; 0 otherwise.

**MeasureItem NODE, LEVEL, REF**

Stores the width of the NODE item in pixels into the REF scalar reference. LEVEL is the node depth as returned by `get_item` for the reference. This notification must be called from within the `begin_paint_info/end_paint_info` block.

**SelectItem [[INDEX, ITEM, SELECTED], [INDEX, ITEM, SELECTED], ...]**

Called when an item gets selected or deselected. The array passed contains a set of arrays for each item where each contains either an integer INDEX or the ITEM, or both. In case the INDEX is undef, the item is invisible; if the ITEM is undef, then the caller didn't bother to call `get_item` for speed reasons, and the receiver should call this function. The SELECTED flag contains the new value of the item.

**Stringify NODE, TEXT_REF**

Stores text string associated with the NODE item into the TEXT_REF scalar reference.

## Prima::StringOutline

A descendant of the `Prima::OutlineViewer` class, provides a standard single-text-item widget. The items can be defined by supplying a text as the first scalar in the node array structure:

```
$string_outline-> items([ 'String', [ 'Descendant' ]]);
```

## Prima::Outline

A variant of `Prima::StringOutline`, with the only difference that the text is stored not in the first scalar in a node but as a first scalar in an anonymous array, which in turn is the first node scalar. The class does not define either format or the number of scalars in the array, and as such presents a half-abstract class.

## Prima::DirectoryOutline

Provides a standard widget with the item tree mapped to the directory structure, so that each item is mapped to a directory. Depending on the type of the host OS, there is either a single root directory ( unix ), or one or more disk drive root items ( win32 ).

The node format is defined as follows:

**#0**

Directory name, string.

**#1**

Parent path; an empty string for the root items.

**#2**

Icon width in pixels, integer.

**#3**

Drive icon; defined only for the root items under Windows to reflect the drive type ( hard, floppy, etc ).

### Properties

**closedGlyphs INTEGER**

The number of horizontal equal-width images in the `closedIcon` property.

Default value: 1

**closedIcon ICON**

Provides an icon representation for the collapsed items.

**openedGlyphs INTEGER**

The number of horizontal equal-width images in the `openedIcon` property.

Default value: 1

**openedIcon OBJECT**

Provides an icon representation for the expanded items.

**path STRING**

Runtime-only property. Selects the current file system path.

**showDotDirs BOOLEAN Selects if the directories with the first dot character are shown in the tree view. The treatment of the dot-prefixed names as hidden is traditional to unix and is of doubtful use under Windows.**

Default value: 0

### Methods

**files [ FILE_TYPE ]**

If the FILE_TYPE value is not specified, the list of all files in the current directory is returned. If FILE_TYPE is given, only the files of the types are returned. The FILE_TYPE is a string, one of those returned by `Prima::Utils::getdir` ( see the **getdir** entry in the *Prima::Utils* section ).

**get_directory_tree PATH**

Reads the file structure under PATH and returns a newly created hierarchy structure in the class node format. If the `showDotDirs` property value is 0, the dot-prefixed names are not included.

Used internally inside the `Expand` notification.

## 4.22 Prima::PodView

POD browser widget

### Synopsis

```
use Prima qw(Application PodView);

my $window = Prima::MainWindow-> create;
my $podview = $window-> insert( 'Prima::PodView',
        pack => { fill => 'both', expand => 1 }
);
$podview-> open_read;
$podview-> read("=head1 NAME\n\nI'm also a pod!\n\n");
$podview-> close_read;

run Prima;
```


### Description

Prima::PodView contains a formatter ( in terms of *perlpod* ) and a viewer of the POD content. It heavily employs its ascendant class the *Prima::TextView* section, and is in turn the base class for the toolkit's default help viewer the *Prima::HelpViewer* section.

### Usage

The package consists of several logically separated parts. These include file locating and loading, formatting, and navigation.

#### Content methods

The basic access to the content is not bound to the file system. The POD content can be supplied without any file to the viewer. Indeed, the file loading routine `load_file` is a mere wrapper to the following content-loading functions:

**open_read %OPTIONS**

Clears the current content and enters the reading mode. In this mode, the content can be appended by repeatedly calling the `read` method that pushes the raw POD content to the parser.

**read TEXT**

Supplies the TEXT string to the parser. Parses basic indentation, but the main formatting is performed inside the *add* entry and the *add_formatted* entry.

Must be called only within the open_read/close_read brackets

**close_read**

Closes the reading mode and starts the text rendering by calling `format`. Returns `undef` if there is no POD context, 1 otherwise.

### Rendering

The rendering is started by the `format` call which returns almost immediately, initiating the mechanism of delayed rendering, which is often time-consuming. `format`'s only parameter KEEP_OFFSET is a boolean flag, which, if set to 1, remembers the current location on a page, and when the rendered text approaches the location, scrolls the document automatically.

The rendering is based on a document model, generated by the open_read/close_read session. The model is a set of the same text blocks defined by the *Prima::TextView* section, except that the header length is only three integers:

```
pod::M_INDENT       - the block X-axis indent
pod::M_TEXT_OFFSET  - same as BLK_TEXT_OFFSET
pod::M_FONT_ID      - 0 or 1, because PodView's fontPalette contains only two fonts -
                  variable ( 0 ) and fixed ( 1 ).
```

The actual rendering is performed in `format_chunks`, where model blocks are transformed into text blocks, wrapped, and pushed into the TextView-provided storage. In parallel, links and the corresponding event rectangles are calculated at this stage.

### Topics

Prima::PodView provides the `::topicView` property, which manages whether the man page is viewed by topics or as a whole. When a page is in the single topic mode, the {`modelRange`} array selects the model blocks that include the topic to be displayed. That way the model stays the same while text blocks inside the widget can be changed.

### Styles

In addition to styles provided by the *Prima::Drawable::Pod* section, Prima::PodView defines `colorMap` entries for `pod::STYLE_LINK` , `pod::STYLE_CODE`, and `pod::STYLE_VERBATIM`:

```
COLOR_LINK_FOREGROUND
COLOR_CODE_FOREGROUND
COLOR_CODE_BACKGROUND
```

The default colors in the styles are mapped into these entries.

### Link and navigation methods

Prima::PodView provides the hand-icon mouse pointer that highlights links. Also, the link documents or topics are loaded in the widget when the user presses the mouse button on the link. the *Prima::Widget::Link* section is used for the implementation of the link mechanics.

If the page is loaded successfully, depending on the `::topicView` property value, either the `select_topic` or `select_text_offset` method is called.

The family of file and link access functions consists of the following methods:

**load_file MANPAGE**

Loads the manpage if it can be found in the PATH or perl installation directories. If unsuccessful, displays an error.

**load_link LINK**

LINK is a text in the format of *perlpod* `L<>` link: "manpage/section". Loads the manpage, if necessary, and selects the section.

**load_bookmark BOOKMARK**

Loads the bookmark string prepared by the the *make_bookmark* entry function. Used internally.

**load_content CONTENT**

Loads content into the viewer. Returns `undef` if there is no POD context, 1 otherwise.

**make_bookmark [ WHERE ]**

Combines the information about the currently viewing page source, topic, and text offset, into a storable string. WHERE, an optional string parameter, can be either omitted, in such case the current settings are used, or be one of the 'up', 'next', or 'prev' strings.

The 'up' string returns a bookmark to the upper level of the manpage.

The 'next' and 'prev' return a bookmark to the next or the previous topics in the manpage.

If the location cannot be stored or defined, `undef` is returned.

**Events**

**Bookmark BOOKMARK**

When a new topic is navigated by the user, this event is triggered with the current topic to have it eventually stored in the bookmarks or user history.

**Link LINK_REF, BUTTON, MOD, X, Y**

When the user clicks on a link, this event is called with the link address, mouse button, modification keys, and coordinates.

**NewPage**

Called after new content is loaded

## 4.23 Prima::ScrollBar

Scroll bars

### Description

`Prima::ScrollBar` implements standard vertical and horizontal scrollbars

### Synopsis

```
use Prima::ScrollBar;

my $sb = Prima::ScrollBar->new( owner => $group, %rest_of_profile);
my $sb = $group-> insert( 'ScrollBar', %rest_of_profile);

my $isAutoTrack = $sb-> autoTrack;
$sb-> autoTrack( $yesNo);

my $val = $sb-> value;
$sb-> value( $value);

my $min = $sb-> min;
my $max = $sb-> max;
$sb-> min( $min);
$sb-> max( $max);
$sb-> set_bounds( $min, $max);

my $step = $sb-> step;
my $pageStep = $sb-> pageStep;
$sb-> step( $step);
$sb-> pageStep( $pageStep);

my $partial = $sb-> partial;
my $whole = $sb-> whole;
$sb-> partial( $partial);
$sb-> whole( $whole);
$sb-> set_proportion( $partial, $whole);

my $size = $sb-> minThumbSize;
$sb-> minThumbSize( $size);

my $isVertical = $sb-> vertical;
$sb-> vertical( $yesNo);

my ($width,$height) = $sb-> get_default_size;
```

### API

#### Properties

**autoTrack BOOLEAN**

Tells the widget if it should send the `Change` notification during mouse tracking events. Generally, it should only be set to 0 on very slow computers.

The default value: 1

**growMode INTEGER**

The default value is `gm::GrowHiX`, i.e. the scrollbar will try to maintain the constant distance from its right edge to its owner's right edge as the owner changes its size. This is useful for horizontal scrollbars.

**height INTEGER**

The default value is $Prima::ScrollBar::stdMetrics[1], which is an operating system-dependent value determined with a call to `Prima::Application->get_default_scrollbar_metrics`. The height is affected because by default the horizontal `ScrollBar` will be created.

**max INTEGER**

Sets the upper limit for `value`.

The default value: 100.

**min INTEGER**

Sets the lower limit for `value`.

The default value: 0

**minThumbSize INTEGER**

A minimal thumb breadth in pixels. The thumb cannot have a main dimension lesser than this.

The default value: 21

**pageStep INTEGER**

This determines the increment/decrement to `value` during the operations that suppose to scroll by pages, for example clicking the mouse on the strip outside the thumb, or pressing `PgDn` or `PgUp`.

The default value: 10

**partial INTEGER**

This tells the scrollbar how many imaginary units the thumb should occupy. See `whole` below.

The default value: 10

**selectable BOOLEAN**

The default value is 0. If set to 1 the widget receives keyboard focus; when in focus, the thumb bar is blinking.

**step INTEGER**

This determines the minimal increment/decrement to `value` during mouse/keyboard interaction.

The default value is 1

**value INTEGER**

A basic scrollbar property; reflects the imaginary position between `min` and `max`, which corresponds directly to the position of the thumb.

The default value is 0

**vertical BOOLEAN**

Determines the main scrollbar style. Set this to 1 when the scrollbar style is vertical, 0 - horizontal. The property can be changed at run-time, so the scrollbars can morph from horizontal to vertical and vice versa.

The default value is 0

**whole INTEGER**

This tells the scrollbar how many imaginary units correspond to the whole length of the scrollbar. This value has nothing in common with `min` and `max`. You may think of the combination of `partial` and `whole` as the proportion between the visible size of something (document, for example) and the whole size of that "something".

The default value is 100.

### Methods

**get_default_size**

Returns two integers, the default platform-dependant width of a vertical scrollbar and the height of a horizontal scrollbar.

### Events

**Change**

The `Change` notification is sent whenever the thumb position of the scrollbar is changed, subject to certain limitations when `autoTrack` is 0. The notification is sent when appropriate, regardless of whether due to the user interaction or a side effect of some method the programmer has called.

**Track**

If `autoTrack` is 0, called when the user changes the thumb position with the mouse.

## Example

```
use Prima;
use Prima::Application name => 'ScrollBar test';
use Prima::ScrollBar;

my $w = Prima::Window->new(
        text => 'ScrollBar test',
        size => [300,200]);

my $sb = $w-> insert( ScrollBar =>
        width => 280,
        left => 10,
        bottom => 50,
        onChange => sub {
                $w-> text( $_[0]-> value);
        });

run Prima;
```

## 4.24 Prima::Sliders

Sliding bars, spin buttons, dial widgets, etc

### Description

The module contains a set of unrelated widget classes that provide input and/or output of an integer value. That is the only thing common in these classes, which are:

```
Prima::AbstractSpinButton
        Prima::SpinButton
        Prima::AltSpinButton

Prima::SpinEdit

Prima::Gauge
Prima::PrigressBar

Prima::AbstractSlider
        Prima::Slider
        Prima::CircularSlider
```

### Prima::AbstractSpinButton

Provides a generic interface to the spin-button class functionality that includes events and range definition properties. Neither `Prima::AbstractSpinButton` nor its descendants store the integer value. These provide a mere possibility for the user to send the incrementing and decrementing commands.

The class is not usable directly.

#### Properties

**state INTEGER**

The property manages a common internal state that doesn't have an exact meaning, as it is only defined in the descendant classes. For example, the state can be set to non-zero when the user performs a mouse drag action.

#### Events

**Increment DELTA**

Called when the user presses a part of the widget that is responsible for incrementing or decrementing commands. DELTA is an integer value that indicates how the associated value must be modified.

**TrackEnd**

Called when the user finished the mouse transaction.

### Prima::SpinButton

A rectangular spin button that consists of three parts, divided horizontally. The upper and the lower parts are push buttons associated with singular increment and decrement commands. The middle part, when dragged by the mouse, fires the `Increment` events with delta value, based on the vertical position of the mouse pointer.

## Prima::AltSpinButton

A rectangular spin button that consists of two push-buttons, associated with singular increment and decrement commands. Compared to `Prima::SpinButton`, the class is a bit less functional but has a more stylish look.

## Prima::SpinEdit


The widget contains a numerical input line and a spin button. The input line value can be changed in three ways - either as a direct traditional keyboard input, as a result of the spin button actions, or as the mouse wheel response. The class provides properties for value storage and range selection.

### Properties

**allowEmpty BOOLEAN**

If set, allows an empty string as a valid value

Default value: false

**circulate BOOLEAN**

Selects the value modification rule when the increment or decrement action hits a range limit. If 1, the value is changed to the opposite limit value ( for example, if the value is 100 in the range 2-100, and the user clicks on the 'increment' button, the value is changed to 2 ).

If 0, the value does not change.

Default value: 0

**editClass STRING**

Assigns the input line class.

A create-only property.

Default value: `Prima::InputLine`

**editDelegations ARRAY**

Assigns the input line list of the notifications.

A create-only property.

**editProfile HASH**

Assigns a hash of properties passed to the input line during the creation.

A create-only property.

**max INTEGER**

Sets the upper limit for `value`.

Default value: 100.

**min INTEGER**

Sets the lower limit for `value`.

Default value: 0

**pageStep INTEGER**

Determines the multiplication factor for incrementing and decrementing actions of the mouse wheel.

Default value: 10

**spinClass STRING**

Assigns the spin-button class.

A create-only property.

Default value: `Prima::AltSpinButton`

**spinProfile ARRAY**

Assigns the spin-button list of the delegated notifications.

A create-only property.

**spinDelegations HASH**

Assigns a hash of properties passed to the spin-button during the creation.

A create-only property.

**step INTEGER**

Determines the multiplication factor for incrementing and decrementing actions of the spin-button.

Default value: 1

**value INTEGER**

Selects the integer value in the range from `min` to `max`. The value is reflected in the input line.

Default value: 0.

### Methods

**set_bounds MIN, MAX**

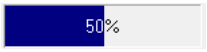
Simultaneously sets both `min` and `max` values.

### Events

**Change**

Called when `value` is changed.

## Prima::Gauge




An output-only widget class, displays a progress bar and an eventual percentage string. Useful as a progress indicator.

### Properties

**indent INTEGER**

Selects the width of the border around the widget.

Default value: 1

**max INTEGER**

Sets the upper limit for `value`.

Default value: 100.

**min INTEGER**

Sets the lower limit for `value`.

Default value: 0

**relief INTEGER**

Selects the style of the border around the widget. Can be one of the following `gr::XXX` constants:

```
gr::Sink    - 3d sunken look
gr::Border  - uniform black border
gr::Raise   - 3d raised look
```

Default value: `gr::Sink`.

**threshold INTEGER**

Selects the threshold value used to determine if the changes to `value` are reflected immediately or are deferred until the value is changed more significantly. When 0, all calls to `value` result in an immediate repaint request.

Default value: 0

**value INTEGER**

Selects the integer value between `min` and `max`, reflected in the progress bar and eventual text.

Default value: 0.

**vertical BOOLEAN**

If 1, the widget is drawn vertically and the progress bar moves from bottom to top. If 0, the widget is drawn horizontally and the progress bar moves from left to right.

Default value: 0

### Methods

**set_bounds MIN, MAX**

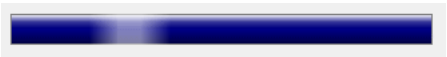
Simultaneously sets both `min` and `max` values.

### Events

**Stringify VALUE, REF**

Converts the integer VALUE into a string format and stores it in the REF scalar reference. Default stringifying conversion is identical to a call to `sprintf("%2d%%")`.

## Prima::ProgressBar



Displays a progress bar

### Properties

**max INTEGER**

Sets the upper limit for `value`.

Default value: 100.

**min INTEGER**

Sets the lower limit for `value`.

Default value: 0

**value INTEGER**

Selects the integer value between `min` and `max`, reflected in the progress bar and an eventual text.

Default value: 0.

### Methods

**set_bounds MIN, MAX**

Simultaneously sets both `min` and `max` values.

## Prima::AbstractSlider

The class provides the basic functionality of a sliding bar, equipped with *tick marks*. The tick marks are supposed to be drawn alongside the main sliding axis or the dialing circle, and provide visual feedback for the user.

The class is not usable directly.

### Properties

**autoTrack BOOLEAN**

A boolean flag, selects the way notifications are executed when the user mouse-drags the sliding bar. If 1, the `Change` notification is executed as soon as `value` is changed. If 0, `Change` is deferred until the user finishes the mouse drag; instead, the `Track` notification is executed when the bar is moved.

This property can be used when the `Change` notification handler performs very slowly, so the eventual fast mouse interactions would not thrash down the program.

Default value: 1

**increment INTEGER**

A step range value used in `scheme` for marking the key ticks. See the *scheme* entry for details.

Default value: 10

**max INTEGER**

Sets the upper limit for `value`.

Default value: 100.

**min INTEGER**

Sets the lower limit for `value`.

Default value: 0

**readOnly BOOLEAN**

If 1, the user cannot change the value by moving the bar or otherwise.

Default value: 0

**ticks ARRAY**

Selects the tick marks representation along the sliding axis or the dialing circle. ARRAY consists of hashes, each for one tick. The hash must contain at least a `value` key with an integer value. The two additional keys `height` and `text`, select the height of a tick mark in pixels, and the text is drawn near the mark, correspondingly.

If ARRAY is `undef`, no ticks are drawn.

**scheme INTEGER**

`scheme` is a property that creates a set of tick marks using one of the predefined scale designs selected by the `ss::XXX` constants. Each constant produces a different scale; some make use of the `increment` integer property that selects a step that is used to place additional text marks. As an example, the `ss::Thermometer` design with the default `min`, `max`, and `increment` values would look like this:

```
0   10   20        100
|    |    |          |
|||||||||||||....|||
```

The module defines the following constants:

```
ss::Axis           - 5 minor ticks per increment
ss::Gauge          - 1 tick per increment
ss::StdMinMax      - 2 ticks at the ends of the bar
ss::Thermometer    - 10 minor ticks per increment, longer text ticks
```

When the `tick` property is explicitly set, `scheme` is reset to `undef`.

**snap BOOLEAN**

If 1, `value` cannot accept values that are not on the tick scale. When such a value is attempted to be set, it is rounded to the closest tick mark. If 0, `value` can accept any integer value in the range from `min` to `max`.

Default value: 0

**step INTEGER**

An integer delta for singular increment and decrement commands, and also a threshold for `value` when the `snap` value is 0.

Default value: 1

**value INTEGER**

Selects an integer value between `min` and `max` and the corresponding sliding bar position.

Default value: 0.

### Events

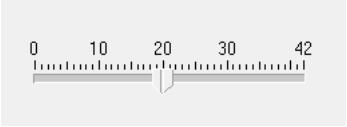
#### Change

Called when the `value` value is changed, with one exception: if the user moves the sliding bar while `autoTrack` is 0, the `Track` notification is called instead.

#### Track

Called when the user moves the sliding bar while the `autoTrack` value is 0; this notification is a substitute to `Change`.

## Prima::Slider




Presents a linear sliding bar, movable along a linear shaft.

### Properties

#### borderWidth INTEGER

In horizontal mode, sets extra margin space between the slider line and the widget boundaries. Can be used for fine-tuning text labels from `ticks()`, where the default spacing (0) or spacing procedure (drop overlapping labels) does not produce decent results.

#### ribbonStrip BOOLEAN

If 1, the parts of the shaft are painted with different colors to increase visual feedback. If 0, the shaft is painted with the single default background color.

Default value: 0

#### shaftBreadth INTEGER

The breadth of the shaft in pixels.

Default value: 6

#### tickAlign INTEGER

One of the `tka::XXX` constants that correspond to the position of the tick marks:

```
tka::Normal        - ticks are drawn on the left or the top of the shaft
tka::Alternative   - ticks are drawn on the right or at the bottom of the shaft
tka::Dual          - ticks are drawn both ways
```

The ticks' orientation ( left or top, right or bottom ) is dependent on the `vertical` property value.

Default value: `tka::Normal`

#### vertical BOOLEAN

If 1, the widget is drawn vertically, and the slider moves from bottom to top. If 0, the widget is drawn horizontally, and the slider moves from left to right.

Default value: 0

### Methods

#### pos2info X, Y

Translates integer coordinates pair ( X, Y ) into the value corresponding to the scale, and returns three scalars:

**info INTEGER**

If `undef`, the user-driven positioning is not possible ( `min` equals to `max` ).

If 1, the point is located on the slider.

If 0, the point is outside the slider.

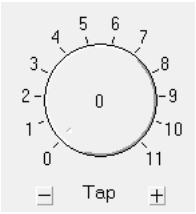
**value INTEGER**

If `info` is 0 or 1, contains the corresponding `value`.

**aperture INTEGER**

Offset in pixels along the shaft axis.

## Prima::CircularSlider




Presents a slider widget with a dialing circle and two increment/decrement buttons. The tick marks are drawn around the perimeter of the dial; the current value is displayed in the center of the dial.

### Properties

#### buttons BOOLEAN

If 1, the increment / decrement buttons are shown at the bottom of the dial, and the user can change the value either by the dial or by the buttons. If 0, the buttons are not shown.

Default values: 0

#### stdPointer BOOLEAN

Determines the style of a value indicator ( pointer ) on the dial. If 1, it is drawn as a black triangular mark. If 0, it is drawn as a small circular knob.

Default value: 0

### Methods

#### offset2data VALUE

Converts integer value in the range from `min` to `max` into the corresponding angle, and returns two floating-point values: cosine and sine of the angle.

#### offset2pt X, Y, VALUE, RADIUS

Converts integer value in the range from `min` to `max` into the point coordinates, with the RADIUS and dial center coordinates X and Y. Return the calculated point coordinates as two integers in the (X,Y) format.

**xy2val X, Y**

Converts widget coordinates X and Y into value in the range from `min` to `max` and returns two scalars: the value and the boolean flag, which is set to 1 if the (X,Y) point is inside the dial circle, and to 0 otherwise.

**Events**

**Stringify VALUE, REF**

Converts integer VALUE into a string format and stores it in the REF scalar reference. The resulting string is displayed in the center of the dial.

The default conversion routine simply copies VALUE to REF as is.

## 4.25 Prima::Spinner

Spinner animation widget

### Synopsis

```
use Prima qw(Application Buttons Spinner);

my $mw = Prima::MainWindow->new(
      size      => [200, 400],
      text      => 'Button Example'
);

my $spinner = $mw->insert('Spinner',
      style     => 'drops',
      size      => [200,400],
      growMode  => gm::Center
);

my $button = $mw->insert(
      'Button',
      text      => 'Start/Stop',
      checkable => 1,
      checked   => 1,
      origin    => [0,0],
      onClick   => sub { $spinner->toggle },
      growMode  => gm::XCenter
);

run Prima;
```


### Description

Prima::Spinner provides a simple spinning animation in three different designs and the opportunity to specify the colors of the spinning animation. This is useful to show the progress of a running process.

### Usage

#### Properties

**active [BOOLEAN]**

Manages whether the spinning animation is active or not.

**color COLOR**

Inherited from the *Prima::Widget* section. `color` manages the basic foreground color. For the spinner widget, this means the background color of the circle or the color of the drops.

**hiliteColor COLOR**

Inherited from the *Prima::Widget* section. The color is used to draw alternate foreground areas with high contrast. For the spinner widget, this defines the color of the arc. Only for the *circle* style.

**showPercent BOOLEAN**

If set, displays completion percent as text. Only for the *circle* style.

**start**

Same as `active(1)`

**stop**

Same as `active(0)`

**style STRING**

`style` can be 'drops', 'circle' or 'spiral'. `drops` shows drops with fading colors. The `circle` style features an arc moving around a circle. `spiral` shows a spinning spiral. The default is 'drops'.

**value INT**

An integer value between 0 and 100, shows completion percentage. Only for the *circle* style.

**toggle**

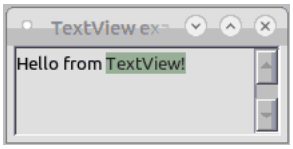
Same as `active(!active)`

## 4.26 Prima::TextView

Rich text browser widget

### Synopsis

```
use strict;
use warnings;
use Prima qw(TextView Application);

my $w = Prima::MainWindow-> create(
    name => 'TextView example',
);

my $t = $w->insert(TextView =>
    text     => 'Hello from TextView!',
    pack     => { expand => 1, fill => 'both' },
);

# Create a single block that renders all the text using the default font
my $tb = tb::block_create();
my $text_width_px = $t->get_text_width($t->text);
my $font_height_px = $t->font->height;
$tb->[tb::BLK_WIDTH]  = $text_width_px;
$tb->[tb::BLK_HEIGHT] = $font_height_px;
$tb->[tb::BLK_BACKCOLOR] = cl::Back;
$tb->[tb::BLK_FONT_SIZE] = int($font_height_px) + tb::F_HEIGHT;
# Add an operation that draws the text:
push @$tb, tb::text(0, length($t->text), $text_width_px);

# Set the markup block(s) and recalculate the ymap
$t->{blocks} = [$tb];
$t->recalc_ymap;

# Additional step needed for horizontal scroll as well as per-character
# selection:
$t->paneSize($text_width_px, $font_height_px);

run Prima;
```




### Description

Prima::TextView accepts blocks of formatted text and provides basic functionality for text scrolling and user selection. The text strings are stored as one large text chunk accessible with the `::text` and `::textRef` properties. A block of formatted text is an array with a fixed-length header and following commands. Each command is formed as an opcode followed by a fixed number of arguments. The block header contains the text offset, which text commands implicitly add to when addressing text strings by the offsets in their arguments.

The package `tb::` provides the block constants and simple functions for creating and accessing blocks, opcodes, and commands.

### Capabilities

Prima::TextView is mostly about text block functions and helpers. It provides functions for wrapping text blocks, calculating block dimensions, and drawing and converting coordinates from (X,Y) to a block position. The class functionality is focused on the text functionality, and although any custom graphic of arbitrary complexity can be embedded in a text block, the internal coordinate system is ( TEXT_OFFSET, BLOCK ) , where TEXT_OFFSET is the text offset from the beginning of a block and BLOCK is an index of a block.

The functionality does not imply any particular text layout - this is up to the class descendants, they must provide their own layout policy. The only policy Prima::TextView requires is that the blocks' BLK_TEXT_OFFSET field must be strictly increasing, and the block text chunks must not overlap. The text gaps are allowed though.

A text block basic drawing function handles the commands changing of color, backColor, and font, and the painting of text strings. Other types of graphics can be achieved by supplying custom code.

#### block_draw CANVAS, BLOCK, X, Y

The `block_draw` method draws BLOCK on the CANVAS in screen coordinates (X,Y). It may be used not only inside begin_paint/end_paint brackets; CANVAS can be an arbitrary `Prima::Drawable` descendant.

#### block_walk BLOCK, %OPTIONS

Cycles through the block opcodes, calls the relevant callbacks on each. The callbacks can be supplied in %OPTIONS.

### Coordinate system methods

Prima::TextView employs two own coordinate systems: *document-based* (X,Y) and *block-based* (TEXT_OFFSET,BLOCK). Each block's text offset is also referred to as *big text offset* vs *small text offset* that is used by individual commands; the small text offset always is added to the block's big text offset to address the string in the widget's text scalar.

The document coordinate system is isometric and measured in pixels. Its origin is located in the imaginary point of the beginning of the document ( not in the first block! ), in the upper-left pixel. X increases to the right, and Y increases down. The block header values BLK_X and BLK_Y use document coordinates, and the widget's pane extents ( regulated the by `::paneSize`, `::paneWidth` and `::paneHeight` properties ) are also in the document coordinates.

The block coordinate system is anisometric - its second axis BLOCK, is an index of a text block in the widget's blocks storage, `$self->{blocks}`, and its first axis TEXT_OFFSET is a text offset from the beginning of the block.

Below are described different coordinate system converters:

#### screen2point, point2screen X, Y

`screen2point` accepts (X,Y) in the screen coordinates ( O is the lower left widget corner ) and returns (X,Y) in document coordinates ( O is the upper left corner of the document ). `point2screen` does the reverse transformation.

#### xy2info X, Y

Accepts (X,Y) is document coordinates, returns (TEXT_OFFSET,BLOCK) coordinates where TEXT_OFFSET is the text offset from the beginning of a block ( not of the whole text! ) , and BLOCK is an index of a block.

#### info2xy TEXT_OFFSET, BLOCK

Accepts (TEXT_OFFSET,BLOCK) coordinates and returns (X,Y) in document coordinates of a block.

**text2xoffset TEXT_OFFSET, BLOCK**

Returns the X coordinate where TEXT_OFFSET begins in a block. BLOCK is the index of the latter.

**info2text_offset**

Accepts (TEXT_OFFSET,BLOCK) coordinates and returns the text offset from the beginning of the whole text.

**text_offset2info TEXT_OFFSET**

Accepts big text offset and returns (TEXT_OFFSET,BLOCK) coordinates

**text_offset2block TEXT_OFFSET**

Accepts big text offset and returns the BLOCK coordinate.

### Text selection

The text selection is performed automatically when the user selects a text region with the mouse. The selection is stored in (TEXT_OFFSET,BLOCK) coordinate pair and is accessible via the `::selection` property. If its value is assigned to (-1,-1,-1,-1) then this indicates that there is no selection. For convenience, the `has_selection` method is introduced.

Also, `get_selected_text` returns the text within the selection (or undef with no selection ), and `copy` copies automatically the selected text into the clipboard. The latter action is bound to the `Ctrl+Insert` key combination.

A block with TEXT_OFFSET set to -1 will be treated as not containing any text, and therefore will not be able to get selected.

### Event rectangles

Partly as an option for future development, partly as a hack a concept of *event rectangles* was introduced. Currently, the {`contents`} private variable points to an array of objects equipped with the `on_mousedown`, `on_mousemove`, and `on_mouseup` methods. These are called by the widget mouse events so that the overloaded classes can define the interactive content without overloading the actual mouse events ( which is although easy but is dependent on the implementation of Prima::TextView's mouse handlers ).

As an example, the *Prima::PodView* section uses the event rectangles to catch the mouse events over the document links. Theoretically, every 'content' can be bound with a separate logical layer; the concept was designed with an HTML browser in mind, so such layers can be thought of as links, image maps, layers, external widgets, etc in the HTML world.

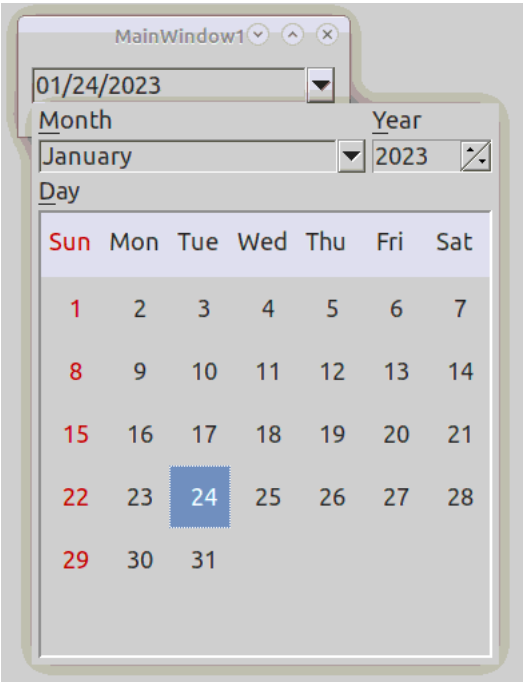
Currently, the `Prima::TextView::EventRectangles` class is provided for such usage. Its property `::rectangles` contains an array of rectangles, and the `contains` method returns an integer value, whether the passed coordinates are inside one of its rectangles or not; in the first case it is the rectangle index.

## 4.27 Prima::Widget::Date

Standard date picker widget

### Synopsis

```
use Prima qw(Application Widget::Date);

my $mw = Prima::MainWindow->new;
$mw->insert( 'Widget::Date' =>
        pack => { fill => 'x', pad => 20 },
);

run Prima;
```




### Description

Standard date picker widget, derived from the `Prima::InputLine` class.

### API

#### Methods

**date2str DATE**

Converts the DATE to a string representation according to the current `format` string

**default_format**

Returns a string to be used in `format` where the string is constructed to reflect the formatting of the regional date preferences.

See also: `man 3 strftime, %x` .

**str2date STRING**

Tries to extract the date from the STRING, assuming it is constructed according to the current `format` string. Doesn't fail but values that could not be extracted are assigned to today's day/month/year instead.

**today**

Returns today's date in widgets [D,M,Y] format

**validate_date D, M, Y**

Checks if D, M, Y form a valid date, and adjusts the values if not. Returns the corrected values.

**Properties**

**date DAY, MONTH, YEAR | [ DAY, MONTH, YEAR ]**

Accepts three integers / arrayref with three integers in the format of `localtime`. DAY can be from 1 to 31, MONTH from 0 to 11, YEAR from 0 to 199.

Default value: today's date.

**day INTEGER**

Selects the day of the month.

**format STRING**

The format string is used when converting the date to its visual interpretation, also with regional preferences, f ex YYYY-MM-DD or DD/MM/YY. The syntax of the format is verbatim as this, i e it recognizes fixed patterns YYYY, YY, MM, and DD, replacing them with the date values.

**month**

Selects the month.

**year**

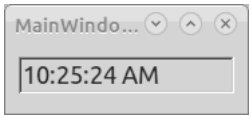
Selects the year.

## 4.28 Prima::Widget::Time

Standard time input widget

### Synopsis

```
use Prima qw(Application Widget::Time);

my $mw = Prima::MainWindow->new;
$mw->insert( 'Widget::Time' =>
        pack => { fill => 'x', pad => 20 },
);

run Prima;
```




### Description

Standard time input widget derived from the `Prima::InputLine` class.

### API

#### Methods

**time2str TIME**

Converts the TIME to a string representation according to the current `format` string

**default_format**

Returns a string to be used in `format` where the string is constructed to reflect the formatting of the regional time preferences.

See also: `man 3 strftime, %X` .

**str2time STRING**

Tries to extract time from STRING, assuming it is constructed according to the current `format` string. Doesn't fail but values that could not be extracted are assigned to the current second/minute/hour instead.

**validate_time S, M, H**

Checks whether S, M, H form a valid point in time, adjusts the values if not. Returns the corrected values.

#### Properties

**format STRING**

The format string is used when converting the time to its visual interpretation, also with regional preferences, f ex hh:mm:ss or hh:mm:AA . The syntax of the format is verbatim as this, i e it recognizes fixed patterns hh, mm, ss, aa, and AA, replacing them with the time values.

(`aa` is for <am / pm>, `AA` is for <AM / PM>).

**hour**

Selects the hour.

**minute**

Selects the minute.

**second INTEGER**

Selects the second

**time SEC, MIN, HOUR | [ SEC, MIN, HOUR ]**

Accepts three integers / arrayref with three integers in the format of `localtime`. SEC and MIN can be from 0 to 59, and HOUR from 0 to 23.

Default value: today's time.

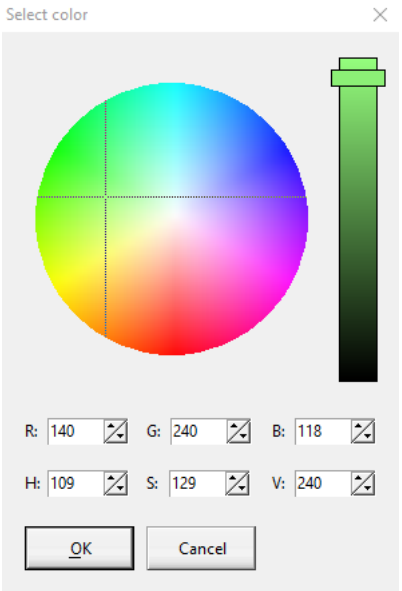
# 5 Standard dialogs

## 5.1 Prima::Dialog::ColorDialog

Standard color selection facilities

### Synopsis

```
use Prima qw(Dialog::ColorDialog Application);

my $p = Prima::Dialog::ColorDialog-> new(
        quality => 1,
);
printf "color: %06x", $p-> value if $p-> execute == mb::OK;
```




### Description

The module contains two packages, `Prima::Dialog::ColorDialog` and `Prima::ColorComboBox`, used as standard tools for the interactive color selection. `Prima::ColorComboBox` is a modified combo widget that provides selecting colors from a predefined palette, but also can invoke a `Prima::Dialog::ColorDialog` window.

## Prima::Dialog::ColorDialog

### Properties

**grayscale BOOLEAN**

If set, allows only gray colors

**quality BOOLEAN**

The setting can increase the visual quality of the dialog if run on paletted displays.

Default value: 0

**value COLOR**

Selects the color represented by the color wheel and other dialog controls.

Default value: `cl::White`

### Methods

**hsv2rgb HUE, SATURATION, LUMINOSITY**

Converts a color from HSV to RGB format and returns three 8-bit integer values, red, green, and blue components.

**rgb2hsv RED, GREEN, BLUE**

Converts color from RGB to HSV format and returns three numerical values, hue, saturation, and luminosity components.

**xy2hs X, Y, RADIUS**

Maps X and Y coordinate values onto a color wheel with RADIUS in pixels. The code uses RADIUS = 119 for mouse position coordinate mapping. Returns three values, - hue, saturation, and error flag. If the error flag is set, the conversion is failed.

**hs2xy HUE, SATURATION**

Maps hue and saturation onto a 256-pixel wide color wheel, and returns X and Y coordinates of the corresponding point.

**create_wheel SHADES, BACK_COLOR**

Creates a color wheel with the number of SHADES given, drawn on a BACK_COLOR background. Returns a `Prima::DeviceBitmap` object.

**create_wheel_shape SHADES**

Creates a circular 1-bit mask with a radius derived from SHAPES. SHAPES must be the same as passed to the *create_wheel* entry. Returns a `Prima::Image` object.

### Events

**BeginDragColor $PROPERTY**

Called when the user starts dragging a color from the color wheel by the left mouse button and an optional combination of Alt, Ctrl, and Shift keys. $PROPERTY is one of the `Prima::Widget` color properties, and depends on a combination of the following keys:

```
Alt             backColor
Ctrl            color
Alt+Shift       hiliteBackColor
Ctrl+Shift      hiliteColor
Ctrl+Alt        disabledColor
Ctrl+Alt+Shift  disabledBackColor
```

The default action reflects the property to be changed in the dialog title

**Change**

The notification is called when the the *value* entry property is changed, either interactively or as a result of a direct call.

**EndDragColor $PROPERTY, $WIDGET**

Called when the user releases the mouse button over a Prima widget. The default action sets `$WIDGET->$PROPERTY` to the selected color value.

### Variables

**$colorWheel**

Contains the cached result of the the *create_wheel* entry method.

**$colorWheelShape**

Contains the cached result of the the *create_wheel_shape* entry method.

## Prima::ColorComboBox

### Events

**Colorify INDEX, COLOR_PTR**

`nt::Action` callback, designed to map combo palette index into an RGB color. INDEX is an integer from 0 to the *colors* entry - 1, COLOR_PTR is a reference to the result scalar where the notification is expected to store the resulting color.

### Properties

**colors INTEGER**

Defines the amount of colors in the fixed palette of the combo box.

**grayscale BOOLEAN**

If set, allows only gray colors

**value COLOR**

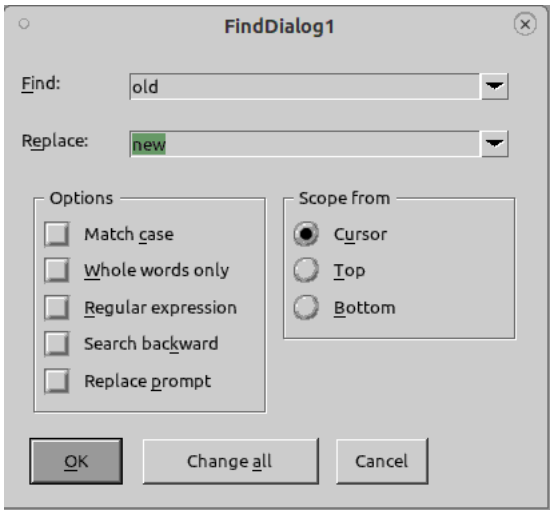
Contains the color selection as a 24-bit integer value.

## 5.2 Prima::Dialog::FindDialog

The standard dialogs FindDialog and ReplaceDialog to find and replace text

### Synopsis

```
use Prima qw(Dialog::FindDialog Application);

my $dlg = Prima::Dialog::FindDialog-> new( findStyle => 0);
my $res = $dlg-> execute;
if ( $res == mb::Ok) {
        print $dlg-> findText, " is to be found\n";
} elsif ( $res == mb::ChangeAll) {
        print "all occurrences of ", $dlg-> findText,
                " are to be replaced by ", $dlg-> replaceText;
}
```




### Description

The module provides two classes Prima::Dialog::FindDialog and Prima::Dialog::ReplaceDialog. The Prima::Dialog::ReplaceDialog class is the same as Prima::Dialog::FindDialog except that its default the *findStyle* entry property value is set to 0. One can cache and reuse the dialog object, changing its the *findStyle* entry value to 0 and 1, so that only one instance of Prima::Dialog::FindDialog is used in the program.

The module does not provide the actual search algorithm; this must be implemented by the programmer. The toolkit includes some help - the part of the algorithm for the `Prima::Edit` class is implemented in the `Prima::Edit/find` method, and another part in the *examples/editor.pl* example program. The the *Prima::HelpWindow* section class also uses the module but implements its own searching algorithm.

### API

#### Properties

All the properties reflect values that the user can change interactively, - except the *findStyle* entry.

**findText STRING**

Selects the text string to be found.

Default value: ”

**findStyle BOOLEAN**

If 1, the dialog provides only the 'find text' interface. If 0, the dialog provides also the'replace text' interface.

Default value: 1 for `Prima::Dialog::FindDialog`, 0 for `Prima::Dialog::ReplaceDialog`.

**options INTEGER**

A combination of the `fdo::` constants. For the detailed description see the **find** entry in the *Prima::Edit* section.

```
fdo::MatchCase
fdo::WordsOnly
fdo::RegularExpression
fdo::BackwardSearch
fdo::ReplacePrompt
```

Default value: 0

**replaceText STRING**

Selects the text string to replace the found text.

Default value: ”

**scope**

One of the `fds::` constants. Represents the scope of the search: it can be started from the cursor position, from the top of the text, or from the bottom.

```
fds::Cursor
fds::Top
fds::Bottom
```

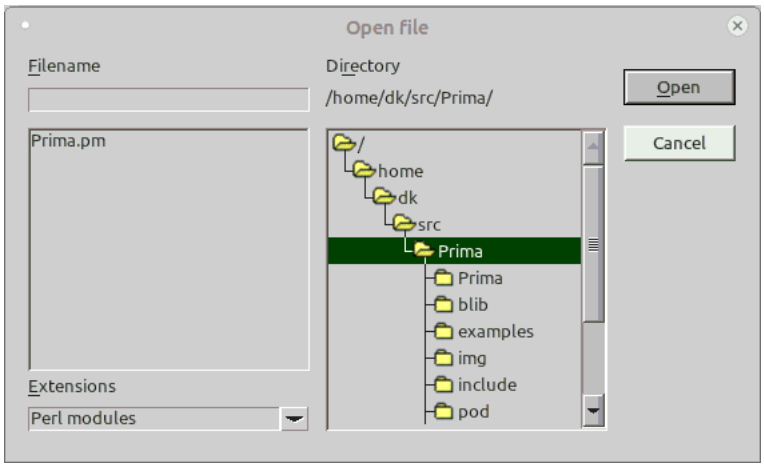
Default value: `fds::Cursor`

## 5.3 Prima::Dialog::FileDialog

File system-related widgets and dialogs

### Synopsis

```
# open file
use Prima qw(Application Dialog::FileDialog);

my $open = Prima::Dialog::OpenDialog-> new(
        filter => [
                ['Perl modules' => '*.pm'],
                ['All' => '*']
        ]
);
print $open-> fileName, " is to be opened\n" if $open-> execute;

# save file
my $save = Prima::Dialog::SaveDialog-> new(
        fileName => $open-> fileName,
);
print $save-> fileName, " is to be saved\n" if $save-> execute;

# open several files
$open-> multiSelect(1);
print $open-> fileName, " are to be opened\n" if $open-> execute;
```




### Description

The module contains standard open file, save file, and change directory dialogs; plus special widgets for file and drive selection that are used in the dialogs

### Prima::DirectoryListBox

A directory list box. Shows the list of subdirectories.

### Properties

**closedGlyphs INTEGER**

The number of horizontal equal-width images, contained in the the *closedIcon* entry property.

Default value: 1

**closedIcon ICON**

Provides the icon for the directories contained in the current directory.

**indent INTEGER**

A positive integer number of pixels, the offset of the hierarchy outline.

Default value: 12

**openedGlyphs INTEGER**

The number of horizontal equal-width images, contained in the the *openedIcon* entry property.

Default value: 1

**openedIcon OBJECT**

Provides the icon for the directories contained above the current directory.

**path STRING**

Runtime-only property. Selects the file path.

**showDotDirs BOOLEAN**

Selects if the directories with the first dot character are shown. The dot-prefixed files are traditionally hidden in unix, so under Windows, this property is not useful.

Default value: 1

### Methods

**files FILE_TYPE**

Returns the list of files filtered by FILE_TYPE. The FILE_TYPE is a string, one of those returned by `Prima::Utils::getdir` ( see the **getdir** entry in the *Prima::Utils* section.

## Prima::DriveComboBox

Drive selector combo-box for non-unix systems

### Properties

**firstDrive DRIVE_LETTER**

Create-only property.

Default value: 'A:'

DRIVE_LETTER can be set to another value to start the drive enumeration. Some OSes can probe eventual diskette drives inside the drive enumeration routines, so it might be reasonable to set DRIVE_LETTER to the `C:` string for responsiveness increase.

**drive DRIVE_LETTER**

Selects the drive letter.

Default value: 'C:'

## Prima::Dialog::FileDialog

Provides the standard file dialog where the user can navigate in the file system and select one or many files. The class can operate in two modes - 'open' and 'save'; these modes are triggered internally by the *Prima::Dialog::OpenDialog* section and the *Prima::Dialog::SaveDialog* section. Some properties behave differently depending on the mode that is stored in the the *openMode* entry property.

### Properties

**createPrompt BOOLEAN**

If 1, and the selected file is nonexistent, asks the user if the file is to be created.

Only actual when the *openMode* entry is 1.

Default value: 0

**defaultExt STRING**

Selects the file extension, appended to the file name typed by the user, if the extension is not given.

Default value: ”

**directory STRING**

Selects the currently selected directory

**fileMustExist BOOLEAN**

If 1, ensures that the file typed by the user exists before closing the dialog.

Default value: 1

**fileName STRING, ...**

For the single-file selection, assigns the selected file name. For the multiple-file selection, on get-calls returns a list of the selected files; on set-calls accepts a single string where the file names are separated by the space character. The eventual space characters must be quoted.

**filter ARRAY**

Contains an array of arrays of string pairs, where each pair describes a file type. The first scalar in the pair is the description of the type; the second is a file mask.

Default value: `[[ 'All files' => '*']]`

**filterIndex INTEGER**

Selects the index in the the *filter* entry array, which is the currently selected file type.

**multiSelect BOOLEAN**

Selects whether the user can select several ( 1 ) or one ( 0 ) file.

See also: the *fileName* entry.

**noReadOnly BOOLEAN**

If 1, fails to open a file when it is read-only.

Default value: 0

Only actual when the *openMode* entry is 0.

**noTestFileCreate BOOLEAN**

If 0, tests if a file that the user selected can be created.

Default value: 0

Only actual when the *openMode* entry is 0.

**overwritePrompt BOOLEAN**

If 1, asks the user if the file selected is to be overwritten.

Default value: 1

Only actual when the *openMode* entry is 0.

**openMode BOOLEAN**

Create-only property.

Selects whether the dialog operates in 'open' ( 1 ) mode or 'save' ( 0 ) mode.

**pathMustExist BOOLEAN**

If 1, ensures that the path typed by the user exists before closing the dialog.

Default value: 1

**showDotFiles BOOLEAN**

Selects if the directories with the first dot character are shown.

Default value: 0

**showHelp BOOLEAN**

A create-only property. If 1, the 'Help' button is inserted in the dialog.

Default value: 0

**sorted BOOLEAN**

Selects whether the file list appears sorted by name ( 1 ) or not ( 0 ).

Default value : 1

**system BOOLEAN**

A create-only property. If set to 1, `Prima::Dialog::FileDialog` returns an instance of the `Prima::sys::XXX::FileDialog` system-specific file dialog, if available for the *XXX* platform.

The `system` property knows only how to map the `FileDialog`, `OpenDialog`, and `SaveDialog` classes onto the system-specific file dialog classes; the inherited classes are not affected and cannot be replaced by the system dialog.

**Methods**

**reread**

Re-reads the currently selected directory.

## Prima::Dialog::OpenDialog

A descendant of the *Prima::Dialog::FileDialog* section tuned for open-dialog functionality.

## Prima::Dialog::SaveDialog

A descendant of the *Prima::Dialog::FileDialog* section tuned for save-dialog functionality.

## Prima::Dialog::ChDirDialog

Provides standard dialog with interactive directory selection.

**Properties**

**directory STRING**

Selects the directory

**showDotDirs**

Selects if the directories with the first dot character are shown

Default value: 0

**showHelp**

Create-only property. If 1, the 'Help' button is inserted in the dialog.

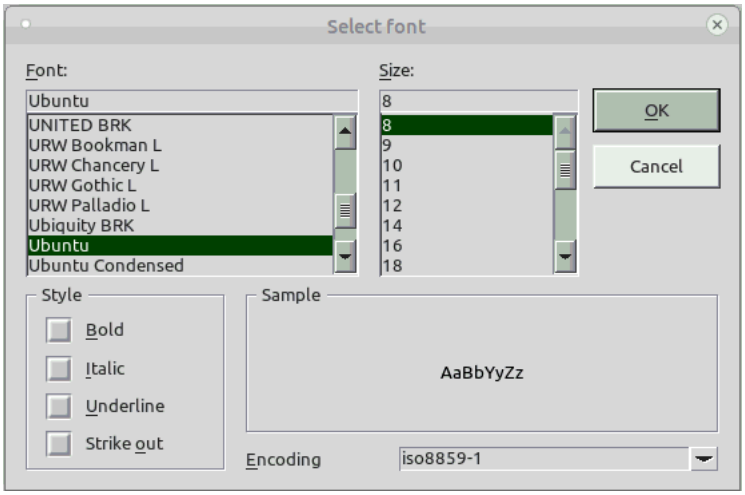
Default value: 0

## 5.4 Prima::Dialog::FontDialog

Standard font dialog

### Synopsis

```
use Prima qw(Application Dialog::FontDialog);
my $f = Prima::Dialog::FontDialog-> create;
return unless $f-> execute == mb::OK;
$f = $f-> logFont;
print "$_:$f->{$_}\n" for sort keys %$f;
```




### Description

The dialog provides standard font selection by name, style, size, and encoding. The selected font is returned by the the *logFont* entry property.

### API

#### Properties

**fixedOnly BOOLEAN**

Selects whether only the fonts of fixed pitch ( 1 ) or all fonts ( 0 ) are displayed in the selection list.

Default value: 0

**logFont FONT**

Provides access to the interactive font selection as a hash reference. FONT format is fully compatible with `Prima::Drawable::font`.

**sampleText STRING**

Sample line of text drawn with the currently selected font

Default value: AaBbYyZz

**showHelp BOOLEAN**

A create-only property.

Specifies if the help button is displayed in the dialog.

Default value: 0

**Events**

**BeginDragFont**

Called when the user starts dragging a font from the font sample widget by the left mouse button.

The default action reflects the dragging status in the dialog title

**EndDragFont $WIDGET**

Called when the user releases the mouse button over a Prima widget. The default action applies the currently selected font to $WIDGET.

## 5.5 Prima::Dialog::ImageDialog

Image file open and save dialogs

### Description

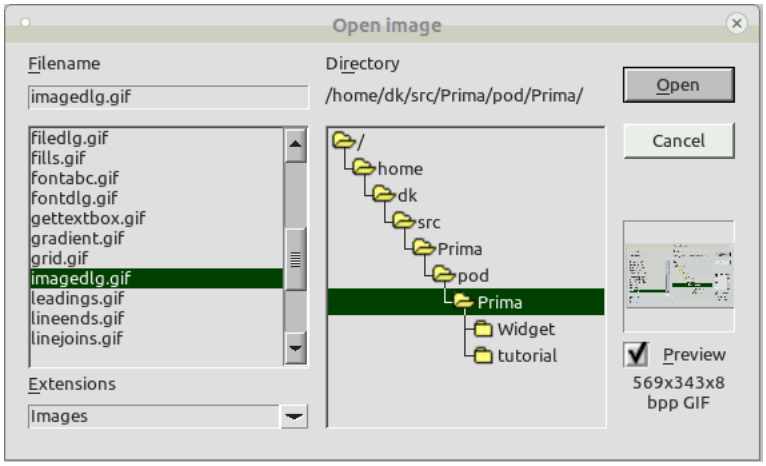
The module provides standard dialogs specially adjusted for image loading and saving.

### Prima::Dialog::ImageOpenDialog

Provides a preview widget, allowing the user to view the image file before loading, and the selection of a frame index for the multi-framed image files. Instead of the `execute` call, the the *load* entry method is used to invoke the dialog and returns the loaded image as a `Prima::Image` object. The loaded object contains the {`extras`} hash with the file and image information filled by the loader. See the *Prima::image-load* section for more information.

#### Synopsis

```
use Prima qw(Application Dialog::ImageDialog);
my $dlg = Prima::Dialog::ImageOpenDialog-> new;
my $img = $dlg-> load;
return unless $img;
print "$_:$img->{extras}->{$_}\n" for sort keys %{$img-> {extras}};
```




#### Properties

**preview BOOLEAN**

Selects if the preview functionality is active. The user can switch it on and off interactively.

Default value: 1

#### Methods

**load %PROFILE**

Executes the dialog, and, if successful, loads the image file and frame selected by the user. Returns the loaded image as a `Prima::Image` object. PROFILE is a hash, passed to the `Prima::Image::load` method. In particular, it can be used to disable the default loading of extra information in the {`extras`} hash variable or to specify a non-default loading option.

For example, {extras}->{className} = 'Prima::Icon' would return the loaded image as an icon object. See the *Prima::image-load* section for more.

load can report the progress of the image loading to the caller, and/or to an instance of Prima::ImageViewer, if desired. If either (or both) onHeaderReady and onDataReady notifications are specified, these are called from the respective event handlers of the image being loaded ( see the **Loading with progress indicator** entry in the *Prima::image-load* section for details). If the profile key progressViewer is supplied, its value is treated as a Prima::ImageViewer instance, and it is used to display the loading progress. See the **watch_load_progress** entry in the *Prima::ImageViewer* section.

#### Events

#### HeaderReady IMAGE

See the **HeaderReady** entry in the *Prima::Image* section.

#### DataReady IMAGE, X, Y, WIDTH, HEIGHT

See the **DataReady** entry in the *Prima::Image* section.

## Prima::Dialog::ImageSaveDialog

Provides the standard image save dialog where the user can select the desired image format, the bit depth, and other format-specific options. The format-specific options can be set if a dialog for the file format is provided by the toolkit. The standard toolkit dialogs reside under the Prima::Image namespace, in the *Prima/Image* subdirectory. For example, Prima::Image::gif provides the selection of a transparent color, and Prima::Image::jpeg provides the image quality control. If the image passed to the the *image* entry property contains the {extras} hash variable, its data are used as the default values. In particular, the {extras}->{codecID} field, responsible for the file format, affects the default file format selection.

#### Synopsis

```
my $dlg = Prima::Dialog::ImageSaveDialog-> new;
return unless $dlg-> save( $image );
print "saved as ", $dlg-> fileName, "\n";
```

#### Properties

#### image IMAGE

Selects the image to be saved. The property is to be used for the standard invocation of dialog, via the execute method. It is not needed when the execution and saving are invoked via the the *save* entry method.

#### Methods

#### save IMAGE, %PROFILE

Invokes the dialog, and, if the execution is successful, saves the IMAGE according to the user selection and PROFILE hash. PROFILE is not used as a source of the default options, but is passed directly to the Prima::Image::save call, possibly overriding the selection of the user.

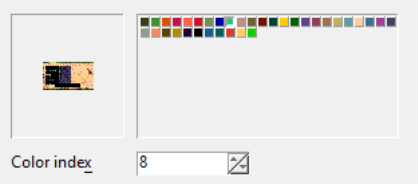
Returns 1 on success, 0 on error. If the error occurs, the user is notified before the method returns.

## 5.6 Prima::Image::TransparencyControl

Standard dialog for selecting transparent color when saving an image to a file.

### Description

The module contains two classes - `Prima::Image::BasicTransparencyDialog` and `Prima::Image::TransparencyControl`. The former provides the dialog used by image codec-specific save options dialogs to select a transparent color index when saving an image to a file. `Prima::Image::TransparencyControl` is the widget class that displays the image palette and allows color rather than index selection.




### Prima::Image::TransparencyControl

#### Properties

**index INTEGER**

Selects the palette index

**image IMAGE**

Selects the image, reads its palette, and displays it in such a manner that only a color that is found in the palette can be selected by the user.

#### Events

**Change**

Triggered when the user changes the `index` property.

### Prima::Image::BasicTransparencyDialog

#### Methods

**transparent BOOLEAN**

If 1, the transparent color widgets are enabled, and the user can select the transparent color index in the image palette. If 0, the widgets are disabled; the image file is saved with no transparent color index.

The property can also be toggled interactively by a checkbox.

## 5.7 Prima::MsgBox

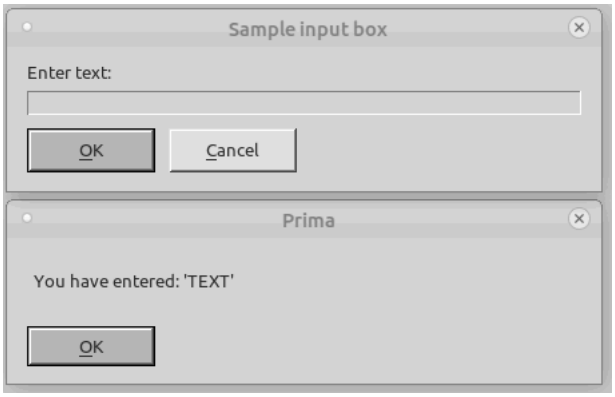
Standard message and input dialog boxes

### Description

The module contains methods that start standard simple message dialogs

### Synopsis

```
use Prima qw(Application);
use Prima::MsgBox qw(input_box message);

my $text = input_box( 'Sample input box', 'Enter text:', '') // '(none)';
message( \ "You have entered: 'B<Q<< $text >>>'", mb::Ok);
```




### API

**input_box TITLE, LABEL, INPUT_STRING, [ BUTTONS = mb::OkCancel, %PROFILES ]**

Invokes a standard dialog box, that contains an input line, a text label, and buttons that end the dialog session. The dialog box uses the TITLE string to display as the window title, the LABEL text to draw next to the input line, and INPUT_STRING, which is the text present in the input box. Depending on the value of the BUTTONS integer parameter, which can be a combination of the `mb::XXX` constants, different combinations of push buttons can be displayed in the dialog.

The PROFILE parameter is a hash, that contains customization parameters for the buttons and the input line. To access the input line parameters the `inputLine` hash key is used. See the *Buttons and profiles* entry for more information on BUTTONS and PROFILES.

Returns different results depending on the call context. In the array context returns two values: the result of `Prima::Dialog::execute` which is either `mb::Cancel` or one of the `mb::XXX` constants of the dialog buttons; and the text entered. The input text is not restored to its original value if the dialog is canceled. In the scalar context returns the text entered, if the dialog ended with `mb::OK` or `mb::Yes` result, or `undef` otherwise.

**message TEXT, [ OPTIONS = mb::Ok | mb::Error, %PROFILES ]**

Same as the `message_box` call, with the application name passed as the title string.

**message_box TITLE, TEXT, [ OPTIONS = mb::Ok | mb::Error, %PROFILES ]**

Invokes the standard dialog box that contains a text label, a predefined icon, and buttons to end the dialog session. The dialog box uses the TITLE string to display as the window title, and the TEXT to display as the main message. The value of the OPTIONS integer parameter is combined from two different sets of `mb::XXX` constants. The first set is the button constants `mb::OK`, `mb::Yes`, etc. See the *Buttons and profiles* entry for the details. The second set consists of the following constants:

```
mb::Error
mb::Warning
mb::Information
mb::Question
```

While there can be several constants of the first set, only one constant from the second set can be selected. Depending on the message type constant, one of the predefined icons is displayed and one of the system sounds is played; if no message type constant is selected, no icon is displayed and no sound is emitted. In case no sound is desired, a special constant `mb::NoSound` can be used.

The PROFILE parameter is a hash that contains customization parameters for the buttons. See the *Buttons and profiles* entry for the details.

Returns the result of `Prima::Dialog::execute` which is either `mb::Cancel` or one of `mb::XXX` constants of the specified dialog buttons.

**signal_dialog $TITLE, $ERROR, $STACK_TRACE**

The standard minimalistic exception dialog shown by default when `Prima::Application.guiException` is 1 and an exception is thrown. Could be reused for other purposes, by supplying a title, error message, and stack trace. If the stack trace is not defined, the corresponding button is not shown.

### Buttons and profiles

The message and input boxes provide several predefined buttons that correspond to the following `mb::XXX` constants:

```
mb::OK
mb::Cancel
mb::Yes
mb::No
mb::Abort
mb::Retry
mb::Ignore
mb::Help
```

To provide more flexibility, the PROFILES hash parameter can be used. In this hash, the following predefined keys tell the dialog methods about certain customizations:

**defButton INTEGER**

Selects the default button in the dialog, i.e. the button that reacts on the return key. Its value must be to an `mb::` constant of the desired button. If this option is not set, the leftmost button is selected as the default.

**helpTopic TOPIC**

Selects the help TOPIC invoked in the help viewer window if the `mb::Help` button is pressed by the user. If this option is not set, the *Prima* section is displayed.

**inputLine HASH**

Only for `input_box`.

Contains the profile hash passed to the input line as creation parameters.

**buttons HASH**

To modify a button, an integer key with the corresponding `mb::XXX` constant can be set with the hash reference under the `buttons` key. The hash is the profile passed to the button as creation parameters. For example, to change the text and behavior of a button, the following construct can be used:

```
Prima::MsgBox::message( 'Hello', mb::OkCancel,
        buttons => {
                mb::Ok, {
                        text     => '~Hello',
                        onClick  => sub { Prima::message('Hello indeed!'); }
                }
        }
);
```

If it is not desired that the dialog must be closed when the user presses a button, its `::modalResult` property ( see the *Prima::Buttons* section ) must be reset to 0.

**owner WINDOW**

If set, the dialog owner is set to WINDOW, otherwise to `$::main_window`. Necessary to maintain window stack order under some window managers, to disallow other windows to be brought over the message box.

**wordWrap BOOLEAN=undef**

`message_box` can display the message in two modes. In `wordWrap = 1` where the text is expected to be relatively short, plus or minus several lines, the user can resize the dialog if for some reason the text is too big. In `wordWrap = 0` mode there is added a scroller, so that even if the text indeed is too big, even when the dialog is maximized.

By default, the function analyzes the message text and decides which of the two modes is suited best. An explicit override is possible with this flag.

## 5.8 Prima::Dialog::PrintDialog

Standard printer setup dialog

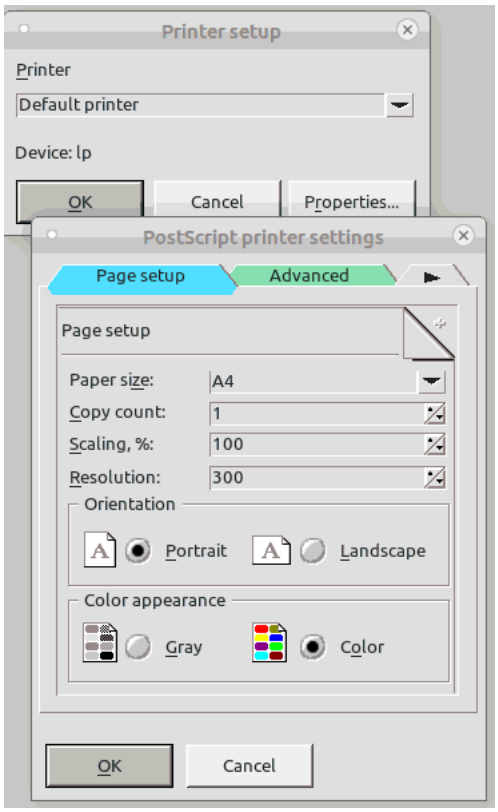
### Description

Provides the standard dialog that allows the user to select the printer and its options. The toolkit does not provide the in-depth management of the printer options; this can only be accessed by executing the printer-specific setup window, called by the `Prima::Printer::setup_dialog` method. The class invokes this method when the user presses the 'Properties' button. Otherwise the class provides only selection from the printer list.

When the dialog finishes successfully the selected printer is set as current by setting the `Prima::Printer::printer` property. This technique allows direct use of the user-selected printer and its properties without prior knowledge of the selection process.

### Synopsis

```
use Prima qw(Dialog::PrintDialog Application);

my $dlg = Prima::Dialog::PrintDialog-> new;
if ( $dlg-> execute) {
        my $p = $dlg-> printer;
        if ( $p-> begin_doc ) {
                $p-> text_out( 'Hello world', 10, 10);
                $p-> end_doc;
        }
}
$dlg-> destroy;
```



# 6 Drawing helpers

## 6.1 Prima::Drawable::Antialias

Alternative API for antialiased shapes

### Description

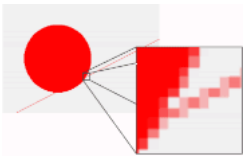
Prima offers drawing antialiased lines and shapes, which is rather slow but provides better visual feedback.

The module augments the `Prima::Drawable` drawing functionality by adding the `new_aa_surface` function, which features two plotting methods, `polyline` and `fillpoly`, identical to the ones in `Prima::Drawable`.

The emulation method in the module used to be the backend and the only implementation of antialiased shapes, but as Prima now supports that internally, this module is not used anymore. It is still functional though and can be used as an alternative.

### Synopsis

```
$canvas-> new_aa_surface-> polyline([0, 0, 100, 100]);

$canvas-> new_path(subpixel => 1)-> ellipse(100,100,100)->fill;
```



### API

#### Methods

**new $CANVAS**

Creates a new AA surface object. The object is cheap to keep and reuse.

**fillpoly $POLY [ $FILLMODE ]**

Paints an antialiased polygon shape. The following properties from the `$CANVAS` are respected: color, backColor, fillPattern, fillPatternOffset, rop2.

**polyline $POLY**

Plots an antialiased polyline. The following properties from the `$CANVAS` are respected: color, backColor, linePattern, lineWidth, lineEnd, lineJoin, miterLimit, rop2

## 6.2 Prima::Drawable::CurvedText

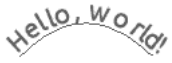
Fit text to path

### Description

The module registers a single function `curved_text_out` in the `Prima::Drawable` namespace. The function plots a line of text along the path defined as a set of points. Various options regulate the behavior of the function when individual glyphs collide with either the path boundaries or each other.

### Synopsis

```
use Prima qw(Application Drawable::CurvedText);
my $spline = [qw(100 100 150 150 200 100)];
$::application-> begin_paint;
$::application-> spline($spline);
$::application-> curved_text_out( 'Hello, world!', $::application-> render_spline( $spline ));
```




#### curved_text_out $TEXT, $POLYLINE, %OPTIONS

`$TEXT` is a line of text, no special treatment is given to tab and newline characters. The text is plotted over `$POLYLINE` path that is an array of coordinate numeric pairs, in the same format as the `Prima::Drawable::polyline` method expects.

The plotting begins by drawing the first glyph at the first point in the path unless specified otherwise with the `offset` option. The glyph is plotted with the angle perpendicular to the first path segment; therefore the path may contain floating point numbers if better angle accuracy is desired.

When the text cannot be fit along a segment it is plotted along the next segment in the path. Depending on the `bevel` boolean option, the next glyph is either drawn on the next segment with the angle corresponding to the tangent of that segment (value 0) or is drawn with the normal text concatenation offset, with the angle averaged between tangents of the two segments it is plotted between (value 1). The default value of the `bevel` option is 1.

The glyph positioning rules differ depending on the `collisions` integer option. If the option is set to 0 (default), the next glyph position always corresponds to the glyph width as projected to the path. This means that glyphs *will* overlap when plotted inside segments forming an acute angle. Also, when plotting along a reflex angle, the glyphs will be visually more distant from each other than when plotted along a straight line.

Simple collision detection can be turned on by setting the `collisions` option to 1 so that no two adjacent glyphs may overlap. The glyphs will be placed together with a minimal distance between them, when possible. With this option set, the function will behave slower. This detection works only for the adjacent glyphs; if the detection of all glyphs in the text is needed, the `collisions` value 2 turns that on. This option may be needed when, for example, the text is plotted inside an acute angle, and the upper parts of glyphs plotted along one segment will overlap the lower parts of glyphs plotted along the other one. Setting `collisions` to 2 will slow the function even more.

The function internally creates an array of tuples where each contains the text string, plotting angle, and the X,Y coordinates for the text to be plotted. If called in the array context, the function returns this array. In the scalar context, the function returns the success flag that is the result of the last call to the `text_out` function.

Options:

**bevel BOOLEAN=true**

If set, the glyphs between two adjoining segments will be plotted with a beveled angle. Otherwise, the glyphs will strictly follow the tangents of the segments in the path.

**callback CODE($SELF, $POLYLINE, $CHUNKS)**

If set, the callback is called with `$CHUNKS` after the calculations are made but before the text is plotted. `$CHUNKS` is an array of tuples where each consists of text, angle, and X,Y coordinates for each text. The callback is free to modify the array.

**collisions INTEGER=0**

If 0, collision detection is disabled, and text glyphs are plotted strictly along the path. If 1, no two adjacent glyphs may overlap, and no two adjacent glyphs will be situated further away from each other than is necessary. If 2, the same functionality as with 1, and also no two glyphs in the whole text will overlap.

**nodraw BOOLEAN=false**

If set, calculate glyph positions but do not draw them.

**offset INTEGER=0**

Sets a pixel offset from the beginning of the path where the first glyph is plotted. If the offset is negative, it is calculated from the end of the path.

**skiptail BOOLEAN=false**

If set, the remainder of the text that is left after the path is completely traversed is not shown. Otherwise (default), the tail text is shown with the angle used to plot the last glyph (if bevelling was requested) or the angle perpendicular to the last path segment (otherwise).

## 6.3 Prima::Drawable::Glyphs

Helper routines for bi-directional text input and complex scripts output

### Synopsis

```
use Prima;
$::application->begin_paint;
$application->text_out('אבס123',100,100);

123אבס
```

### Description

The class implements an abstraction layer by organizing arrays filled with information about glyphs as a structure that can be used to render text strings. Objects of the class are created and returned by the `Prima::Drawable::text_shape` method, see more in the **text_shape** entry in the *Prima::Drawable* section. A `Prima::Drawable::Glyphs` object is a blessed array reference that can contain either two, four, or five packed arrays with 16-bit integers, representing, correspondingly, a set of glyph indexes, a set of character indexes, a set of glyph advances, a set of glyph position offsets per glyph, and a font index. Additionally, the class implements several sets of helper routines that aim to address common tasks when displaying glyph-based text.

#### Structure

Each sub-array is an instance of the `Prima::array` class, an effective plain memory structure that provides a standard perl interface over a string scalar filled with fixed-width integers.

The following methods provide read-only access to these arrays:

**glyphs**

Contains a set of unsigned 16-bit integers where each is a glyph number corresponding to the font that was used for shaping the text. The glyph numbers are only applicable to the font used in the shaping process. Zero is usually treated as a default glyph in vector fonts when shaping cannot map a character; in bitmap fonts, this number is usually the same as `defaultChar`.

The glyphs array is recognized as a special case when is sent to the `text_out` or `get_text_width` methods that can process it natively. In this case, no special advances and glyph positions are taken into account.

Each glyph is not necessarily mapped to a text character, and quite often is not, even in English left-to-right texts. F ex character combinations like `"ff"`, `"fi"`, and `"fl"` may be mapped to single ligature glyphs. When right-to-left, *RTL*, text direction is taken into account, the glyph positions may change, too. See `indexes` below that addresses the mapping of glyphs to characters.

**indexes**

Contains a set of unsigned 16-bit integers where each is a text offset corresponding to the text used in the shaping process. Each glyph position points to the first character in the text that maps to the glyph.

There can be more than one character per glyph, such as the above example with the `"ff"` ligature. There can also be cases with more than one character per more than one glyph, f ex in indic scripts. In these cases it is easier to operate neither by character offsets nor

by glyph offsets, but rather by *clusters*, where each cluster is an individual syntax unit that contains one or more characters per one or more glyphs.

In addition to the text offset, each index value can be flagged with a `to::RTL` bit, signifying that the character in question has RTL direction. This is not necessarily semitic characters from RTL languages that only have that attribute set; spaces in these languages are normally attributed with the RTL bit too, sometimes also numbers. The use of explicit direction control characters from the U+20XX block can result in any character being assigned or not assigned the RTL bit.

The array has an extra item added to its end, the length of the text that was used for the shaping. This helps calculate the cluster length in characters, especially of the last one, where the difference between indexes is, basically, the cluster length.

The array is not used for text drawing or calculation, but only for conversion between character, glyph, and cluster coordinates (see `Coordinates` below).

**advances**

Contains a set of unsigned 16-bit integers where each is a pixel distance of how much space the corresponding glyph occupies. Where the advances array is not present, or was force-filled by `advances` options by the `text_shape` method, a glyph advance value is basically the sum of a, b, and c widths of the corresponding glyph. However there are cases when depending on the shaping input, these values can differ.

One of those cases is the combining graphemes, where the text consisting of two characters, `"A"` and the combining grave accent U+300 should be drawn as a single "À" symbol, and where the font doesn't have that single glyph but rather two individual glyphs `"A"` and `"‘"`. Even though the grave glyph has its own advance for standalone usage, in this case, it should be ignored; this is achieved by the shaper setting the advance of the `"‘"` to zero.

The array content is respected by the `text_out` and `get_text_width` methods, and its content can be changed at will to produce gaps in the text quite easily. F ex `Prima::Edit` uses that to display tab characters as spaces with the 8x advance.

**positions**

Contains a set of pairs of signed 16-bit integers where each is an X and Y pixel offset for each glyph. Like in the previous example with the "À" symbol, the grave glyph `"‘"` may be positioned differently on the vertical axis in "À" and "à" graphemes, for example.

The array is respected by `text_out` (but not by `get_text_width`).

**fonts**

Contains a set of unsigned 16-bit integers where each is an index in the font substitution list (see the **font_mapper** entry in the *Prima::Drawable* section). Zero means the current font.

The font substitution is applied by the `text_shape` method when the `polyfont` option is set (it is by default), and when the shaper cannot match all characters in the text to the glyphs using the current font. If the current font contains all the needed glyphs, this entry is not present at all.

The array is respected by the `text_out` and `get_text_width` methods.

### Coordinates

In addition to the natural character coordinates, where each index is a text offset that can be directly used in the `substr` perl function, the `Prima::Drawable::Glyphs` class offers two additional coordinate systems that help abstract the object data for the display and navigation.

The glyph coordinate system is a rather straightforward copy of the character coordinate system, where each number is an offset in the `glyphs` array. Similarly, these offsets can be used

to address individual glyphs, indexes, advances, and positions. However, these are not easy to use when one needs, for example, to select a grapheme with a mouse, or break a set of glyphs in such a way that a grapheme is not broken. These use cases can be managed more easily in the cluster coordinate system.

The cluster coordinates represent a virtually superimposed set of offsets where each corresponds to a set of one or more characters displayed by one or more glyphs. The most useful functions below operate in this system.

### Visual selection

The coordinates that are best used for implementing the visual selection are either characters or clusters, but not glyphs. The charater-based selection makes it trivial to extract or replace the selected text, while the cluster-based makes it easier to manipulate (f ex with Shift- arrow keys) the selection itself.

The class supports both, by operating on *selection maps* or *selection chunks*, where each represents the same information but in different ways. For example, consider an embedded number in a bidi text. For the sake of clarity, I'll use Latin characters here. Let's imagine a text scalar containing these characters:

```
ABC123
```

where *ABC* is a right-to-left text that, if rendered on the screen should be displayed as

```
123CBA
```

(and the indexes, i e the offsets of the first characters for each glyph, are (3,4,5,2,1,0) ).

Next, the user clicks the mouse between the glyphs A and B (in the text offset of 1), drags the mouse to the left, and finally stops between the characters 2 and 3 (in the text offset of 4). The resulting selection then should not be, as one might naively expect, this:

```
123CBA
--^^^-
```

but this instead:

```
123CBA
^^-^^-
```

because the next character after C is 1, and the *range* of the selected sub-text is from characters 1 to 4.

The class offers means to encode such information in a *map*, i.e. an array of integers `1,1,0,1,1,0`, where each entry is either 0 or 1 depending on whether the cluster is or is not selected. Alternatively, the same information can be encoded in *chunks*, or RLE sets, as an array `0,2,1,2,1`, where the first integer signifies the number of non-selected clusters to display, the second - the number of selected clusters, the third the non-selected again, etc. If the first character belongs to the selected chunk, the first integer in the result is set to 0.

### Bidi input

When sending an input to a widget to type some text, the otherwise trivial case of figuring out at which position the text should be inserted (or removed, for that matter), becomes interesting when there are characters with mixed input direction.

F ex it is indeed trivial, when the Latin text is `AB`, and the cursor is positioned between `A` and `B`, to figure out that whenever the user types `C`, the result should become `ACB`. Likewise, when the text is RTL and both text and input are Arabic, the result is the same. However when f.ex. the text is `A1`, which is displayed as `1A` because of the RTL shaping, and the cursor is positioned

between the `1` (LTR) and `A` (RTL) glyphs, it is not clear whether that means the new input should be appended after `1` and become `A1C`, or after `A`, and become, correspondingly, `AC1`.

There is no easy solution for this problem, and different programs approach this differently, where some go as far as to provide two cursors for both input directions. The class offers its own solution that uses some primitive heuristics to detect whether the cursor belongs to the left or the right glyph. This is the area that can be enhanced, and any help from native users of the languages that use the right-to-left writing system can be greatly appreciated.

## API

### abc $CANVAS, $INDEX

Returns the a, b, c metrics from the glyph `$INDEX`

### advances

A read-only accessor to the advances array, see the *Structure* entry above.

### clone

Clones the object

### cluster2glyph $FROM, $LENGTH

Maps the range of clusters starting with `$FROM` with size `$LENGTH` into the corresponding range of glyphs. Undefined `$LENGTH` calculates the range from `$FROM` to the object's end.

### cluster2index $CLUSTER

Returns character offset of the first character in the cluster `$CLUSTER`.

Note: result may contain `to::RTL` flag.

### cluster2range $CLUSTER

Returns character offset of the first character in the cluster `$CLUSTER` and the number of characters in the cluster.

### clusters

Returns an array of integers where each is the offset of the first character in each cluster.

### cursor2offset $AT_CLUSTER, $PREFERRED_RTL

Given the cursor is positioned next to the cluster `$AT_CLUSTER`, runs simple heuristics to calculate what character offset it corresponds to. The `$PREFERRED_RTL` flag is used when object data does not have enough information to decide the text direction.

See the *Bidi input* entry above.

### def $CANVAS, $INDEX

Returns the d, e, f metrics from the glyph `$INDEX`

### fonts

A read-only accessor to the font indexes, see the *Structure* entry above.

### get_box $CANVAS

Return box metrics of the glyph object.

See the **get_text_box** entry in the *Prima::Drawable* section.

### get_sub $FROM, $LENGTH

Extracts and clones a new object that contains data from cluster offset `$FROM` with cluster length `$LENGTH`.

**get_sub_box $CANVAS, $FROM, $LENGTH**

Calculate the box metrics of the glyph string from the cluster `$FROM` with size `$LENGTH`.

**get_sub_width $CANVAS, $FROM, $LENGTH**

Calculate the pixel width of the glyph string from the cluster `$FROM` with size `$LENGTH`.

**get_width $CANVAS, $WITH_OVERHANGS**

Returns the width of the glyph objects, with overhangs if requested.

**glyph2cluster $GLYPH**

Return the cluster that contains `$GLYPH`.

**glyphs**

A read-only accessor to the glyph indexes array, see the *Structure* entry above.

**glyph_lengths**

Returns an array where each glyph position is the number of how many glyphs the corresponding cluster occupies

**index2cluster $INDEX, $ADVANCE = 0**

Returns the cluster that contains the character offset `$INDEX`.

Set the `$ADVANCE` 1 to add the RTL-dependent advance to the resulting cluster

**indexes**

A read-only accessor to the indexes, see the *Structure* entry above.

**index_lengths**

Returns an array where each glyph position is the number of how many characters the corresponding cluster occupies

**justify CANVAS, TEXT, WIDTH, %OPTIONS**

An umbrella call for `justify_interspace` if `$OPTIONS{letter}` or `$OPTIONS{word}` is set; for `justify_arabic` if `$OPTIONS{kashida}` is set; and for `justify_tabs` if `$OPTIONS{tabs}` is set.

Returns a boolean flag whether the glyph object was changed or not.

**justify_arabic CANVAS, TEXT, WIDTH, %OPTIONS**

Performs justifications of Arabic TEXT with kashida to the given WIDTH, returns either a success flag, or a new text with explicit *tatweel* characters inserted.

```
my $text = "\x{6a9}\x{634}\x{6cc}\x{62f}\x{647}";
my $g = $canvas->text_shape($text) or return;
$canvas->text_out($g, 10, 50);
$g->justify_arabic($canvas, $text, 200) or return;
$canvas->text_out($g, 10, 10);
```

Inserts tatweels only between Arabic letters that did not form any ligatures in the glyph object, max one tatweel set per word (if any). Does not apply the justification if the letters

in the word are rendered as LTR due to embedding or explicit shaping options; only does justification on RTL letters. If for some reason newly inserted tatweels do not form a monotonically increasing series after shaping, skips the justifications in that word.

Note: Does not use the JSTF font table, on Windows results may be different from the native rendering.

Options:

If justification is found to be needed, eventual ligatures with newly inserted tatweel glyphs are resolved via a call to `text_shape(%OPTIONS)` - so any needed shaping options, such as `language`, may be passed there.

**as_text BOOL = 0**

If set, returns the new text with inserted tatweels, or undef if no justification is possible.

If unset, runs in-place justification on the caller glyph object, and returns the boolean success flag.

**min_kashida INTEGER = 0**

Specifies the minimal width of a kashida strike to be inserted.

**kashida_width INTEGER**

During the calculation, the width of the tatweel glyph is needed - unless supplied by this option, it is calculated dynamically. Also, when called in the list context, and succeeds, returns a `1, kashida_width` tuple that can be reused in subsequent calls.

**justify_interspace CANVAS, TEXT, WIDTH, %OPTIONS**

Performs an in-place inter-letter and/or inter-word justification of TEXT to the given WIDTH. Returns either a boolean flag whether there were any changes made, or, the new text with explicit space characters inserted.

Options:

**as_text BOOL = 0**

If set, returns new text with inserted spaces, or undef if no justification is possible.

If unset, runs in-place justification on the caller glyph object, and returns the boolean success flag.

**letter BOOL = 1**

If set, runs an inter-letter spacing on all glyphs.

**max_interletter FLOAT = 1.05**

When the inter-letter spacing is applied, it is applied first, so that the width of the resulting text line can take up to `$OPTIONS{max_interletter} * glyph_width` pixels.

The inter-word spacing does not have such a limit, and in the worst case can produce two words moved to the left and the right edges of the enclosing 0 - WIDTH-1 rectangle.

**space_width INTEGER**

`as_text` mode: during the calculation, the width of the space glyph may be needed. Unless supplied by `$OPTIONS{space_width}`, it is calculated dynamically. Also, when called in the list context, and succeeds, returns the `1, space_width` tuple that can be reused in subsequent calls.

**word BOOL = 1**

If set, runs an inter-word spacing by extending advances on all space glyphs.

**min_text_to_space_ratio FLOAT = 0.75**

If the `word` option set, does not run inter-word justification if the text-to-space ratio is too small (to not spread the text too thin)

**justify_tabs CANVAS, TEXT, %OPTIONS**

Expands the tab characters as `$OPTIONS{tabs}` (default:8) spaces.

Needs the advance of the space glyph to replace the tab glyph. If no `$OPTIONS{glyph}` and `$OPTIONS{width}` are specified, calculates them.

Returns a boolean flag whether there were any changes made. On success, if called in the list context, returns also the space glyph ID and space glyph width for eventual use on the later calls.

**left_overhang**

The first integer from the `overhangs` result.

**log2vis**

Returns a map of integers where each character position corresponds to the glyph position. The name is a rudiment from pure fribidi shaping, where `log2vis` and `vis2log` were mapper functions with the same functionality.

**n_clusters**

Calculates how many clusters are there in the object

**new @ARRAYS**

Creates a new object. Is not used directly, created automatically inside the `text_shape` method.

**new_array NAME**

Creates an array suitable for direct insertion to the object, if manual construction of the object is needed. F ex one may set the missing `fonts` array like this:

```
$obj->[ Prima::Drawable::Glyphs::FONTS() ] = $obj->new_array('fonts');
$obj->fonts->[0] = 1;
```

The newly created array is filled with zeros.

**new_empty**

Creates a new empty object.

**overhangs**

Calculates two widths for overhangs at the beginning and at the end of the glyph string. This is used in the emulation of the `get_text_width` method with the `to::AddOverhangs` flag.

**positions**

A read-only accessor to the positions array, see the *Structure* entry above.

**reorder_text TEXT**

Returns a visual representation of `TEXT` assuming it was the input of the `text_shape` call that created the object.

**reverse**

Creates a new object that has all arrays reversed. Used for calculation of the pixel offset from the right end of a glyph string.

**right_overhang**

The second integer from the `overhangs` result.

**selection2range $CLUSTER_START $CLUSTER_END**

Converts cluster selection range into text selection range

**selection_chunks_clusters, selection_chunks_glyphs $START, $END**

Converts text selection given as the visual range between `$START` and `$END` into a set of integers (*chunks*), where each is the number or selected or not-selected clusters or glyphs. The first chunk is a number of non-selected items and is 0 if the first cluster or glyph is selected.

**selection_diff $OLD, $NEW**

Given two chunk sets in the format as returned by `selection_chunks_clusters` or `selection_chunks_glyphs`, calculates the new set of chunks where each integer value corresponds to the number of the clusters or glyphs affected by the transition from the `$OLD` to `$NEW` visual selection. The first chunk is the number of non-affected items and is 0 if the first cluster or glyph is affected by the selection change.

Can be used for efficient repaints when the user interactively changes text selection, to redraw only the changed regions.

**selection_map_clusters, selection_map_glyphs $START, $END**

Same as `selection_chunks_XXX`, but instead of RLE chunks returns a full array for each cluster/glyph, where each entry is a boolean value corresponding to whether that cluster/glyph is to be displayed as selected or not.

**selection_walk $CHUNKS, $FROM, $TO = length, $SUB**

Walks the selection chunks array, returned by `selection_chunks`, between `$FROM` and `$TO` clusters/glyphs. Calls the provided `$SUB->($offset, $length, $selected)` for each chunk where each call contains 2 integers - the chunk offset and its length, and a boolean flag whether the chunk is selected or not.

Can be also used on a result of `selection_diff`, in which case the `$selected` flag shows whether the chunk is affected by the selection change or not.

**sub_text_out $CANVAS, $FROM, $LENGTH, $X, $Y**

An optimized version of `$CANVAS->text_out( $self->get_sub($FROM, $LENGTH), $X, $Y )`.

**sub_text_wrap $CANVAS, $FROM, $LENGTH, $WIDTH, $OPT, $TABS**

An optimized version of `$CANVAS->text_wrap( $self->get_sub($FROM, $LENGTH), $WIDTH, $OPT, $TABS )`. The result is also converted to chunks.

**text_length**

Returns the length of the text that was shaped and that produced the object.

**x2cluster $CANVAS, $X, $FROM, $LENGTH**

Given the sub-cluster from `$FROM` with size `$LENGTH`, calculates how many clusters would fit in `$X` pixels.

**_debug**

Dumps the glyph object content in a readable format.

## 6.4 Prima::Drawable::Gradient

Gradient fills for primitives

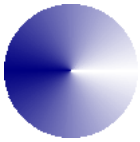
### Description

Prima offers simple gradient services to draw gradually changing colors. A gradient is made by setting at least two colors and optionally a set of points that, when projected, generate the transition curve between the colors.

The module augments the `Prima::Drawable` drawing functionality by adding the `new_gradient` function.

### Synopsis

```
$canvas-> new_gradient(
        palette => [ cl::White, cl::Blue, cl::White ],
)-> sector(50,50,100,100,0,360);
```



### API

#### Methods

**clone %OPTIONS**

Creates a new gradient object with %OPTIONS replaced.

**new $CANVAS, %OPTIONS**

Creates a new gradient object. The following %OPTIONS are available:

**dither BOOLEAN = 0**

When set, applies not only gradient colors but also different fill patterns to create an even smoother transition effect between adjacent colors. Works significantly slower.

**palette @COLORS**

Each color is a `cl::` value. The gradient is calculated as a polyline where each of its vertices corresponds to a certain blend between two adjacent colors in the palette. F.ex. the simplest palette going from `cl::White` to `cl::Black` over a transition line `0..1` (default), produces a pure white color at the start and a pure black color at the end, with all available shades of gray in between.

**poly @VERTICES**

A set of 2-integer polyline vertices where the first integer is a coordinate (x, y, or whatever is required by the drawing primitive) between 0 and 1, and the second is the color blend value between 0 and 1.

Default: ((0,0),(1,1))

**spline \@VERTICES, %OPTIONS**

Serving the same purpose as the `poly` option but the vertices are projected first to a B-spline curve using the *render_spline* entry and `%OPTIONS`. The resulting polyline is treated as `poly`.

**vertical BOOLEAN**

Only used in the the *bar* entry primitive, to set the gradient direction.

**widgetClass INTEGER**

Points to the widget class to resolve generic colors like `cl::Back` that may differ between widget classes.

See also: the *bar* entry, the *stripes* entry .

**bar X1, Y1, X2, Y2, VERTICAL = 0**

Draws a filled rectangle with (X1,Y1) - (X2,Y2) extents

Context used: fillPattern, rop, rop2

**colors BREADTH**

Returns a list of gradient colors for each step from 1 to BREADTH. When `dither` is set, each color is an array of three items, - the two adjacent colors and an integer value between 0 and 63 that reflects the amount of blending needed between the colors.

**ellipse X, Y, DIAM_X, DIAM_Y**

Draws a filled ellipse with the center in (X,Y) and diameters (DIAM_X,DIAM_Y)

Context used: fillPattern, rop, rop2

**sector X, Y, DIAM_X, DIAM_Y, START_ANGLE, END_ANGLE**

Draws a filled sector with the center in (X,Y) and diameters (DIAM_X,DIAM_Y) from START_ANGLE to END_ANGLE

Context used: fillPattern, rop, rop2

**stripes BREADTH**

Returns an array consisting of integer pairs, where the first one is a color value, and the second is the breadth of the color strip. the *bar* entry uses this information to draw a gradient fill, where each color strip is drawn with its own color. Can be used for implementing other gradient-aware primitives (see *examples/f_fill.pl* )

## 6.5 Prima::Drawable::Markup

Allow markup in widgets

### Synopsis

```
use Prima qw(Application Buttons);
use Prima::Drawable::Markup q(M);
my $m = Prima::MainWindow->new;
$m-> insert( Button =>
    text   => Prima::Drawable::Markup->new(markup => "B<Bold> bU<u>tton"),
    hotKey => 'u',
    pack   => {},
);
$m->insert( Button => pack => {}, text => M "I<Italic> button" );
$m->insert( Button => pack => {}, text => \ "Not an Q<I<italic>> button" );

run Prima;
```


### Description

`Prima::Drawable::Markup` adds the ability to recognize POD-like markup to Prima widgets. Supported markup sequences are `B` (bold text), `I` (italic text), `U` (underlined text), `F` (change font), `S` (change font size), `C` (change foreground color), `G` (change background color), `M` (move pointer), `W` (disable wrapping), and `P` (picture).

The `F` sequence is used as follows: *n*|*text*, where `n` is a 0-based index into the `fontPalette`.

The `S` sequence is used as follows: `n|text`, where `n` is the number of points relative to the current font size. The font size may optionally be preceded by `+` or `-`.

The `C` and `G` sequences are used as follows: `c|text`, where `c` is either: a color in any form accepted by Prima, including the `cl` constants (`Black Blue Green Cyan Red Magenta Brown LightGray DarkGray LightBlue LightGreen LightCyan LightRed LightMagenta Yellow White Gray`). Or, a 0-based index into the `colorPalette`. Also, `default` can be used to set the color that the canvas originally had. For `G` a special value `off` can be used to turn off the background color and set it as transparent.

The `M` command has three parameters, comma-separated: X, Y, and flags. X and Y are coordinates of how much to move the current pointer. By default X and are in pixels, and do not extend block width. `flags` is a set of characters, where each is:

```
m - set units to font height
p - set units to points
x - also extend the block width
```

The text inside the `W` sequence will not be wrapped during `text_wrap` calls.

The text inside the `Q` sequence will not be treated as markup.

The `P` sequence is used as follows:`P<n>`, where `n` is a 0-based index into the `picturePalette`.

The the *URL*|*text* entry sequence parsing results in making 1) `text` of color `linkColor`, and 2) wrapping the text with `OP_LINK` commands in the block, that do nothing by default but could be used by whoever uses the block. See the *Prima::Widget::Link* section for more and the *Prima::Label* section as an example.

The methods `text_out` and `get_text_width` are affected by `Prima::Drawable::Markup`. `text_out` will write formatted text to the canvas, and `get_text_width` will return the width of the formatted text. **NOTE**: These methods do not save the state between calls, so your markup cannot span lines (since each line is drawn or measured with a separate call).

The module can export a single method `M` which is a shortcut over the creation of a new markup object with default color, font, and image palettes. These can be accessed directly as `@COLORS`, `@FONTS`, `@IMAGES` correspondingly.

## API

The following properties are used:

**colorPalette([@colorPalette])**

Gets or sets the color palette to be used for `C` sequences within this widget. Each element of the array should be a `cl::` constant.

**fontPalette([@fontPalette])**

Gets or sets the font palette to be used for `F` sequences within this widget. Each element of the array should be a hashref suitable for setting a font.

**picturePalette([@picturePalette])**

Gets or sets the picture palette to be used for `P` sequences within this widget. Each element of the array should be a `Prima::Image` descendant.

## 6.6 Prima::Drawable::Metafile

Graphics recorder

### Description

Metafiles can record graphic primitives and replay them later on another canvas

### Synopsis

```
my $metafile = Prima::Drawable::Metafile->new( size => [30, 30] );
$metafile->begin_paint;
$metafile->rectangle(10,10,20,20);
$metafile->end_paint;

$metafile->execute( $another_drawable, 100, 100 );
```

### API

**call $SUB::($self,$canvas,@ARGS), @ARGS**

`$SUB` will be called when the metafile is executed, with the first two parameters the metafile and the target canvas, and `@ARGS` thereafter.

**clear**

When called without parameters, clears the content before proceeding. Otherwise same as `Drawable.clear`.

**execute CANVAS,X,Y**

Draws the content on the CANVAS with X,Y offset

**size X,Y**

Sets the metafile extensions; the content is not clipped by it.

## 6.7 Prima::Drawable::Path

Stroke and fill complex paths

### Description

The module augments the `Prima::Drawable`'s drawing and plotting functionality by implementing paths that allow arbitrary combinations of polylines, splines, and arcs, to be used for drawing or clipping shapes.

### Synopsis

```
# draws an elliptic spiral
my ( $d1, $dx ) = ( 0.8, 0.05 );
$canvas-> new_path->
        rotate(45)->
        translate(200, 100)->
        scale(200, 100)->
        arc( 0, 0, $d1 + $dx * 0, $d1 + $dx * 1, 0, 90)->
        arc( 0, 0, $d1 + $dx * 2, $d1 + $dx * 1, 90, 180)->
        arc( 0, 0, $d1 + $dx * 2, $d1 + $dx * 3, 180, 270)->
        arc( 0, 0, $d1 + $dx * 4, $d1 + $dx * 3, 270, 360)->
stroke;
```

### API

#### Primitives

All primitives come in two versions, with absolute and relative coordinates. The absolute version draws a graphic primitive so that its starting point (or a reference point) is at (0,0). The relative version, called with an 'r' (f.ex. `line` vs `rline`) has its starting point as the ending point of the previous primitive (or (0,0) if there's none).

**arc CENTER_X, CENTER_Y, DIAMETER_X, DIAMETER_Y, ANGLE_START, ANGLE_END, TILT = 0**

Adds an elliptic arc to the path. The arc is centered around the (CENTER_X,CENTER_Y) point.

Important: if the intention is an immediate rendering, especially with a 1-pixel line width, consider decreasing diameters by 1. This is because all arc calculations are made with the floating point precision, where the diameter is also given not in pixels but in geometrical coordinates, to allow for matrix transformations. Before rendering is performed, arcs are transformed into spline vertices and then the transformation matrix is applied, and by that time the notion of an arc diameter is lost to be successfully converted into pixel size minus one.

Read more about this in the **Antialiasing and alpha** entry in the *Prima::Drawable* section .

**close, open**

Closes the current shape and opens a new one. close() is the same as open() but makes sure the shape's first point is equal to its last point.

**circular_arc ANGLE_START, ANGLE_END**

Adds a circular arc to the path. Note that adding transformations will effectively make it into an elliptic arc, which is used internally by `arc` and `rarc`.

**chord CENTER_X, CENTER_Y, DIAMETER_X, DIAMETER_Y, ANGLE_START, ANGLE_END.**

Adds a chord to the path. Is there only for compatibility with `Prima::Drawable`.

**ellipse CENTER_X, CENTER_Y, DIAMETER_X, DIAMETER_Y = DIAMETER_X, TILT = 0**

Adds a full ellipse to the path.

**glyph INDEX, %OPTIONS**

Adds a glyph outline to the path. `%OPTIONS` are passed as is to the **renger_glyph** entry in the *Prima::Drawable* section, except the `fill` option.

Note that filled glyphs require `fillMode` without the `fm::Overlay` bit set, and also the `fill` option set to generate proper shapes with holes.

**line, rline @POINTS**

Adds a polyline to the path

**lines [X1, Y1, X2, Y2]..**

Adds a set of multiple, unconnected lines to the path. Is there only for compatibility with `Prima::Drawable`.

**moveto, rmoveto X, Y**

Stops plotting the current shape and moves the plotting position to X, Y.

**rarc DIAMETER_X, DIAMETER_Y, ANGLE_START, ANGLE_END, TILT = 0**

Adds an elliptic arc to the path so that the first point of the arc starts on the last point of the previous primitive, or (0,0) if there's none.

**rectangle X1, Y1, X2, Y2**

Adds a rectangle to the path. Is there only for compatibility with `Prima::Drawable`.

**round_rect X1, Y1, X2, Y2, MAX_DIAMETER**

Adds a round rectangle to the path.

**sector CENTER_X, CENTER_Y, DIAMETER_X, DIAMETER_Y, ANGLE_START, ANGLE_END**

Adds a sector to the path. Is there only for compatibility with `Prima::Drawable`.

**spline, rspline $POINTS, %OPTIONS.**

Adds a B-spline to the path. See the **spline** entry in the *Prima::Drawable* section for `%OPTIONS` descriptions.

**text TEXT, %OPTIONS**

Adds `TEXT` to the path. `%OPTIONS` are the same as in the **render_glyph** entry in the *Prima::Drawable* section, except that `unicode` is deduced automatically based on whether the `TEXT` has utf8 bit on or off. An extra option `cache` with a hash can be used to speed up the function with subsequent calls. The `baseline` option is the same as the **textOutBaseline** entry in the *Prima::Drawable* section.

Note that filled glyphs require `fillMode` without the `fm::Overlay` bit set, and also the `fill` option set to generate proper shapes with holes.

### Transformations

Transformation calls change the current path properties (matrix etc) so that all subsequent calls would use them until a call to `restore` is made. The `save` and `restore` methods implement the stacking mechanism so that local transformations can be made.

### Properties

**canvas DRAWABLE**

Provides access to the attached drawable object

**matrix A, B, C, D, Tx, Ty**

Applies a transformation matrix to the path. The matrix, as used by the module, is formed as follows:

```
A  B  0
C  D  0
Tx Ty 1
```

When applied to 2D coordinates, the transformed coordinates are calculated as

```
X' = AX + CY + Tx
Y' = BX + DY + Ty
```

**precision INTEGER**

Selects current precision for splines and arcs. See the **spline** entry in the *Prima::Drawable* section, `precision` entry.

**restore**

Pops the stack entry and replaces the current matrix and graphic properties with it.

**rotate ANGLE**

Adds rotation to the current matrix

**save**

Saves the current matrix and graphic properties on the stack.

**shear X, Y = X**

Adds shearing to the current matrix

**scale X, Y = X**

Adds scaling to the current matrix

**subpixel BOOLEAN**

Turns on and off slow but more precise floating-point calculation mode

Default: depends on the canvas antialiasing mode

**translate X, Y = X**

Adds an offset to the current matrix

### Operations

These methods perform path rendering and create an array of points that can be used for drawing

**clip %options**

Returns a 1-bit image with the clipping mask created from the path. `%options` can be used to pass the `fillMode` property that affects the result of the filled shape.

**contours**

Same as the *points* entry but further reduces lines into a set of 8-connected points, suitable to be traced pixel-by-pixel.

**extents**

Returns two points that box the path.

**last_matrix**

Returns the current transform matrix (CTM) after running all commands

**fill fillMode=undef**

Paints a filled shape over the path. If `fillMode` is set, it is used instead of the one selected on the canvas.

**fill_stroke fillMode=undef**

Paints a filled shape over the path with the background color. If `fillMode` is set, it is used instead of the one selected on the canvas. Thereafter, draws a polyline over the path.

**flatten PRESCALE**

Returns new objects where arcs are flattened into lines. The lines are rasterized with a scaling factor that is as close as possible to the device pixels, to be suitable for a call to the polyline() method. If the PRESCALE factor is set, it is used instead to premultiply coordinates of arc anchor points used to render the lines.

**points**

Runs all accumulated commands, returns rendered set of points suitable for the `Prima::Drawable::polyline` and `Prima::Drawable::fillpoly` methods.

**region MODE=fm::Winding|fm::Overlay, RGNOP=rgnop::Union**

Creates a region object from the path. If MODE is set, applies fill mode (see the **fillMode** entry in the *Prima::Drawable* section for more); if RGNOP is set, applies region set operation (see the **combine** entry in the *Prima::Region* section).

**stroke**

Draws a polyline over the path

**widen %OPTIONS**

Expands the path into a new path object containing outlines of the original path as if drawn with selected line properties. The values of `lineWidth`, `lineEnd`, `lineJoin`, and `linePattern` are read from `%OPTIONS`, or from the attached canvas when available. Supports the `miterLimit` option with values from 0 to 20.

Note: if the intention is to immediately render lines, decrease lineWidth by 1 (they are 1 pixel wider because paths are built around the assumption that pixel size is 0, which makes them scalable).

### Methods for custom primitives

**append PATH**

Copies all commands from another PATH object. The PATH object doesn't need to have balanced stacking brackets `save` and `restore`, and can be viewed as a macro.

**identity**

Returns the identity matrix

**matrix_apply @POINTS**

Applies the CTM to POINTS, returns the transformed points. If @POINTS is a list, returns the transformed points as a list; if it is an array reference, returns an array reference.

## 6.8 Prima::Drawable::Pod

POD parser and renderer

### Synopsis

```
use Prima::Drawable::Pod;
use Prima::PS::Printer;

my $pod = Prima::Drawable::Pod->new;
$pod-> load_pod_content("=head1 NAME\n\nI'm also a pod!\n\n");

my $printer = Prima::PS::PDF::File->new( file => 'pod.pdf');
$printer-> begin_doc or die $@;
$pod-> print($printer);
$printer-> end_doc;
```

### Description

Prima::Drawable::Pod contains a formatter ( in terms of *perlpod* ) and a renderer of the POD content. The POD text is converted in *model*, a set of text blocks in format described in the *Prima::Drawable::TextBlock* section. The model blocks are not directly usable though, and would need to be rendered to another set of text blocks, that in turn can be drawn on the screen, a printer, etc. The module also provides helper routines for these operations.

### Usage

The package consists of several logically separated parts. These include file locating and loading, formatting, and navigation.

#### Content methods

**load_pod_content CONTENT, %OPTIONS**

High-level POD content parser. `%OPTIONS` are same as in `open_read`.

**open_read %OPTIONS**

Clears the current content and enters the reading mode. In this mode, the content can be appended by repeatedly calling the `read` method that pushes the raw POD content to the parser.

**read TEXT**

Supplies the TEXT string to the parser. Parses basic indentation, but the main formatting is performed inside the *add* entry and the *add_formatted* entry.

Must be called only within the open_read/close_read brackets

**add TEXT, STYLE, INDENT**

Formats the TEXT string of a given STYLE ( one of the `pod::STYLE_XXX` constants) with the INDENT space.

Must be called only within the open_read/close_read brackets.

**add_formatted Format, TEXT**

Adds a pre-formatted TEXT with a given Format, supplied by the `=begin` or `=for` POD directives. Prima::PodView understands 'text' and 'podview' FORMATs; the latter format

is for Prima::PodView itself and contains a small number of commands for rendering images in documents.

The 'podview' commands are:

**cut**

Example:

```
=for podview <cut>

=for text just text-formatter info

        ....
        text-only info
        ...

=for podview </cut>
```

The <cut<gt> clause skips all POD input until canceled. It is used in conjunction with the following command, the *img* entry, to allow a POD manpage to provide both graphic ('podview', 'html', etc ) and text ( 'text' ) content.

**img [src="SRC"] [width="WIDTH"] [height="HEIGHT"] [cut="CUT"] [frame="FRAME"]**

An image inclusion command, where *src* is a relative or an absolute path to an image file. In case scaling is required, `width` and `height` options can be set. If the image is a multiframe image, the frame index can be set by the `frame` option. A special `cut` option, if set to a true value, activates the the *cut* entry behavior if ( and only if ) the image load operation is unsuccessful. This makes possible simultaneous use of 'podview' and 'text' :

```
=for podview <img src="graphic.gif" cut=1 >

=begin text

y     .
|  .
|.
+----- x

=end text

=for podview </cut>
```

In the example above 'graphic.gif' will be shown if it can be found and loaded, otherwise, the poor-man drawings will be selected.

If `src` is omitted, the image is retrieved from the `images` array, from the index `frame`.

It is also possible to embed images in the pod, by using a special `src` tag for base64-encoded images. The format should preferably be GIF, as this is Prima default format, or BMP for very small images, as it is supported without third-party libraries:

```
=for podview <img src="data:base64">
R0lGODdhAQABAIAAAAAAAAAAACwAAAAAQABAIAAAAAAAAACAkQBADs=
```

**close_read**

Closes the reading mode. Returns `undef` if there is no POD context, or a hash with `topic_id` (ID of the first topic containing the content) and the `success` flag otherwise.

### Topics

Topics reside in the {topics} array, where each is an array with the following indices of the pod::T_XXX constants:

```
pod::T_MODEL_START - start of topic
pod::T_MODEL_END   - end of a topic
pod::T_DESCRIPTION - topic name
pod::T_STYLE       - pod::STYLE_XXX constant
pod::T_ITEM_DEPTH  - depth of =item recursion
pod::T_LINK_OFFSET - offset in the links array
```

### Styles

The ::styles property provides access to the styles, applied to different pod text parts. These styles are:

```
pod::STYLE_CODE     - style for C<>
pod::STYLE_TEXT     - normal text
pod::STYLE_HEAD_1   - =head1
pod::STYLE_HEAD_2   - =head2
pod::STYLE_HEAD_3   - =head3
pod::STYLE_HEAD_4   - =head4
pod::STYLE_HEAD_5   - =head5
pod::STYLE_HEAD_6   - =head6
pod::STYLE_ITEM     - =item
pod::STYLE_LINK     - style for L<> text
pod::STYLE_VERBATIM - style for pre-formatted text
```

Each style is a hash with the following keys: fontId, fontSize, fontStyle, color, and backColor, fully analogous to the tb::BLK_DATA_XXX options. This functionality provides another layer of accessibility to the pod formatter.

### Rendering

The model loaded by the read functions is stored internally. It is independent of screen resolution, fonts, colors, etc. To be rendered or printed, the following functions can be used:

**begin_format %OPTIONS**

Starts formatting session. The following options are recognized:

**allow_width_overrun BOOLEAN=1**

If set, allows resulting block width to overrun the canvas width. If set, the actual width can be queried by calling the accumulated_width_overrun method. Otherwise forcibly breaks blocks explicitly marked to be not wrapped.

**colormap ARRAY**

Array of at least 5 color entries (default foreground color, default background color, link color, verbatime text color, and its background color). If unset, some sensible default values are used.

**fontmap ARRAY_OF_HASHES**

Set of at least 2 hashes each describing a font to be used for normal text (index 0) and verbatim text (index 1). If unset, some sensible default values are used.

**hmargin, vmargin**

Target device margins

**resolution ARRAY_OF_2**

Target device resolution

**width, height**

Target device size

**format_model $MODEL**

Renders a model block `$MODEL` and returns zero or more text blocks suitable for the drawing on the given canvas. Also the `block_draw` method can be used for the same purpose.

**end_format**

Ends formatting session

### Printing

The method `print` prints the pod content on a target canvas. Accepts the following options (along with all the other options from `begin_format`)

**canvas OBJECT**

The target device

**from, to INDEX**

Selects the model range to be printed

### Block export

The method `export_blocks` can render the model into a set of blocks that can be reused elsewhere. This functionality is used by `Prima::Label` that is able to display the pod content. Returns a `Prima::Drawable::PolyTextBlock` object that is a super-set of text blocks that also contains all necessary information (fonts, colors, etc) needed to pass on the block drawing routines and to be suitable as input for `text_out` method.

The method accepts the following options (along with all the other options from `begin_format`):

**canvas OBJECT**

The target device

**from, to INDEX**

Selects the model range to be printed

**max_height INTEGER**

Stops rendering after `max_height` pixel are occupied by the pod content

**trim_header BOOLEAN**

If set, removes the topic or page header, so that only the content itself is rendered

**trim_footer BOOLEAN**

Prunes empty newlines

**width INTEGER**

Desired render width in pixels

### Navigation

#### load_link LINK, %OPTIONS

Parses and loads POD content from LINK. If the LINK contains a section reference, loads only that section. Returns the success flag.

`%OPTIONS` are same as understood by the `load_pod_file` and `open_read`.

#### load_pod_file FILE, %OPTIONS

High-level POD file reader. `%OPTIONS` are same as in `open_read`.

#### parse_link LINK

The method `parse_link` accepts text in the format of *perlpod* `L<>` link: "manpage/section". Returns a hash with up to two items, `file` and `topic`. If the `file` is set, then the link contains a file reference. If the `topic` is set, then the link topic matches the currently loaded set of topic.

Note: if the file requested is not loaded, f.ex. by `load_pod_file`, then `topic` will not me matched. Issue another call to `parse_link` to match the topic if `file` is set.

## 6.9 Prima::Drawable::Subcanvas

Paint a hierarchy of widgets to any drawable

### Description

Needed for painting a screenshot on an image, printer, etc. Adds two methods to the `Prima::Drawable` namespace: the *paint_with_widgets* entry and the *screenshot* entry.

### Synopsis

```
use Prima qw(Application Button);
my $w = Prima::MainWindow-> create;
$w->insert( 'Button' );
$w->screenshot->save('a.bmp');
```

### Methods

**paint_with_widgets $canvas, $x=0, $y=0**

Given a `$canvas` is in the paint mode, traverses all widgets as they are seen on the screen, and paints them on the canvas with given `$x,$y` offsets.

**screenshot $canvas, %opt**

Syntax sugar over the paint_with_widgets. Creates an image with the `$self`'s, size, and calls `paint_with_widgets` with it. Returns the screenshot.

## 6.10 Prima::Drawable::TextBlock

Rich text representation

### API

#### Block header

A block's fixed header consists of `tb::BLK_START - 1` integer scalars, each of which is accessible via the corresponding `tb::BLK_XXX` constant. The constants are separated into two logical groups:

```
BLK_FLAGS
BLK_WIDTH
BLK_HEIGHT
BLK_X
BLK_Y
BLK_APERTURE_X
BLK_APERTURE_Y
BLK_TEXT_OFFSET
```

and

```
BLK_FONT_ID
BLK_FONT_SIZE
BLK_FONT_STYLE
BLK_COLOR
BLK_BACKCOLOR
```

The first group defines the offset constants that are used to address the values in the block header; the constants lie in the 0 - `tb::BLK_START - 1` range. The second group values line in the `tb::BLK_DATA_START` - `tb::BLK_DATA_END` range. This is done for eventual backward compatibility, if the future development changes the length of the header.

The fields from the first group define the text block dimension, aperture position, and text offset ( remember, the text is stored as one big chunk ). The second group defines the initial color and font settings. Prima::TextView needs all fields of every block to be initialized before displaying. The the *block_wrap* entry method can be used for the automated assigning of these fields.

#### Block parameters

The scalars after `tb::BLK_START` encode the commands to the block renderer. These commands have their own parameters which follow the command. The length of the command is encoded in the high 16-bit word of the command. The basic command set includes `OP_TEXT`, `OP_COLOR`, `OP_FONT`, `OP_TRANSPOSE`, and `OP_CODE`. The additional codes are `OP_WRAP` and `OP_MARK`, not used in drawing but are special commands to the *block_wrap* entry.

##### OP_TEXT - TEXT_OFFSET, TEXT_LENGTH, TEXT_WIDTH

`OP_TEXT` commands to draw a string, from the offset `tb::BLK_TEXT_OFFSET + TEXT_OFFSET`, with the length TEXT_LENGTH. The third parameter TEXT_WIDTH contains the width of the text in pixels. The scheme is made for simplification of an imaginary code, that would alter ( insert to, or delete part of ) the text; the updating procedure would not need to traverse all commands in all blocks, but only the block headers.

Relative to: `tb::BLK_TEXT_OFFSET`

### OP_COLOR - COLOR

OP_COLOR sets foreground or background color. To set the background, COLOR must be or-ed with the tb::BACKCOLOR_FLAG value. In addition to the two toolkit-supported color values ( RRGGBB and system color index ), COLOR can also be or-ed with the tb::COLOR_INDEX flag, in such case it is treated an index in the ::colormap property array.

Relative to: tb::BLK_COLOR, tb::BLK_BACKCOLOR.

### OP_FONT - KEY, VALUE

As a font is a complex property which includes font name, size, direction, etc fields, the OP_FONT KEY represents one of the three parameters - tb::F_ID, tb::F_SIZE, tb::F_STYLE. All three have different VALUE meanings.

Relates to: tb::BLK_FONT_ID, tb::BLK_FONT_SIZE, tb::BLK_FONT_STYLE.

#### F_STYLE

Contains a combination of the fs::XXX constants, such as fs::Bold, fs::Italic etc.

Default value: 0

#### F_SIZE

Contains the relative font size. The size is relative to the current font size. As such, 0 is a default value, and -2 is the default font decreased by 2 points. Prima::TextView provides no range checking ( but the toolkit does ), so while it is o.k. to set the negative F_SIZE values larger than the default font size, one must be careful when relying on the combined font size value .

If the F_SIZE value is added to the F_HEIGHT constant, then it is treated as font height in pixels rather than font size in points. The macros for these opcodes are named respectively tb::fontSize and tb::fontHeight, while the opcode is the same.

#### F_ID

All other font properties are collected under an 'ID'. ID is an index in the ::fontPalette property array, which contains font hashes with the other font keys initialized - name, encoding, and pitch. These three fields are required to be defined in the font hash; the other font fields are optional.

### OP_TRANSPOSE X, Y, FLAGS

Contains a mark for an empty space. The space is extended to the relative coordinates (X,Y), so the block extension algorithms take this opcode into account. If FLAGS does not contain tb::X_EXTEND, then in addition to the block expansion, the current coordinate is also moved to (X,Y). (OP_TRANSPOSE,0,0,0) and (OP_TRANSPOSE,0,0,X_EXTEND) are identical and are empty operators.

The X_DIMENSION_FONT_HEIGHT flag indicates that (X,Y) values must be multiplied by the current font height. Another flag X_DIMENSION_POINT does the same but multiplies by the current value of the the *resolution* entry property divided by 72 ( treats X and Y not as pixel but as point values).

OP_TRANSPOSE can be used for customized graphics, in conjunction with OP_CODE to assign a space, so the rendering algorithms do not need to be rewritten every time a new graphic is invented. For example, see how the *Prima::PodView* section implements images and bullet points.

### OP_CODE - SUB, PARAMETER

Contains a custom code pointer SUB with a parameter PARAMETER, passed when the block is about to be drawn. SUB is called with the following format:

```
( $widget, $canvas, $text_block, $font_and_color_state, $x, $y, $parameter);
```

$font_and_color_state ( or $state, through the code ) contains the state of font and color commands in effect, and is changed as the rendering algorithm advances through the block. The format of the state is the same as of the text block, and the F_ID, F_SIZE, the F_STYLE constants are the same as BLK_FONT_ID, BLK_FONT_SIZE, and BLK_FONT_STYLE.

The SUB code is executed only when the block is about to be drawn.

### OP_WRAP mode

OP_WRAP is only used in the the *block_wrap* entry method. mode is a flag, selecting the wrapping command.

```
WRAP_MODE_ON   - default, block commands can be wrapped
WRAP_MODE_OFF  - cancels WRAP_MODE_ON, commands cannot be wrapped
WRAP_IMMEDIATE - proceed with immediate wrapping, unless the ignoreImmediateWrap option i
```

the *block_wrap* entry does not support stacking for the wrap commands, so the (OP_WRAP,WRAP_MODE_ON,OP_WRAP,WRAP_MODE_ON,OP_WRAP,WRAP_MODE_OFF) command sequence has the same effect as the (OP_WRAP,WRAP_MODE_OFF) sequence. If mode is WRAP_MODE_ON, wrapping is disabled - all following commands are treated as non-wrappable until the (OP_WRAP,WRAP_MODE_OFF) command sequence is met.

### OP_MARK PARAMETER, X, Y

OP_MARK is only in effect in the the *block_wrap* entry method and is a user command. the *block_wrap* entry only sets (!) X and Y to the current coordinates when the command is met. Thus, OP_MARK can be used for arbitrary reasons, for example for saving the geometrical positions during the block wrapping.

These opcodes are far not enough for the full-weight rich text viewer. However, the new opcodes can be created using tb::opcode, which accepts the opcode length and returns the new opcode value.

### Rendering methods

### block_wrap %OPTIONS

block_wrap wraps a block into a given width in pixels. It returns one or more text blocks with fully formed headers. The returned blocks are located one below another, providing an illusion that the text itself is wrapped. It does not only traverse the opcodes and sees if the command fits in the given width; it also splits the text strings if these do not fit.

By default, the wrapping can occur either on a command boundary or by the spaces or tab characters in the text strings. The unsolicited wrapping can be prevented by using the OP_WRAP command brackets. The commands inside these brackets are not wrapped; the OP_WRAP commands are removed from the resulting blocks.

block_wrap copies all commands and their parameters as is, except the following:

- OP_TEXT's third parameter, TEXT_WIDTH, is disregarded, and is recalculated for every OP_TEXT command.

- If OP_TRANSPOSE's third parameter, X_FLAGS contains the X_DIMENSION_FONT_HEIGHT flag, the command coordinates X and Y are multiplied to the current font height, and the flag is cleared in the output block. The X_DIMENSION_PIXEL has a similar effect but the coordinates are multiplied by the current resolution divided by 72.

- OP_MARK's second and third parameters are assigned to the current (X,Y) coordinates.

- OP_WRAP is removed from the output.

**justify_interspace %OPTIONS**

Uses `$OPTIONS{width}` and `$OPTIONS{min_text_to_space_ratio}` to try to make inter-word spacing. Returns new block if successful, undef otherwise.

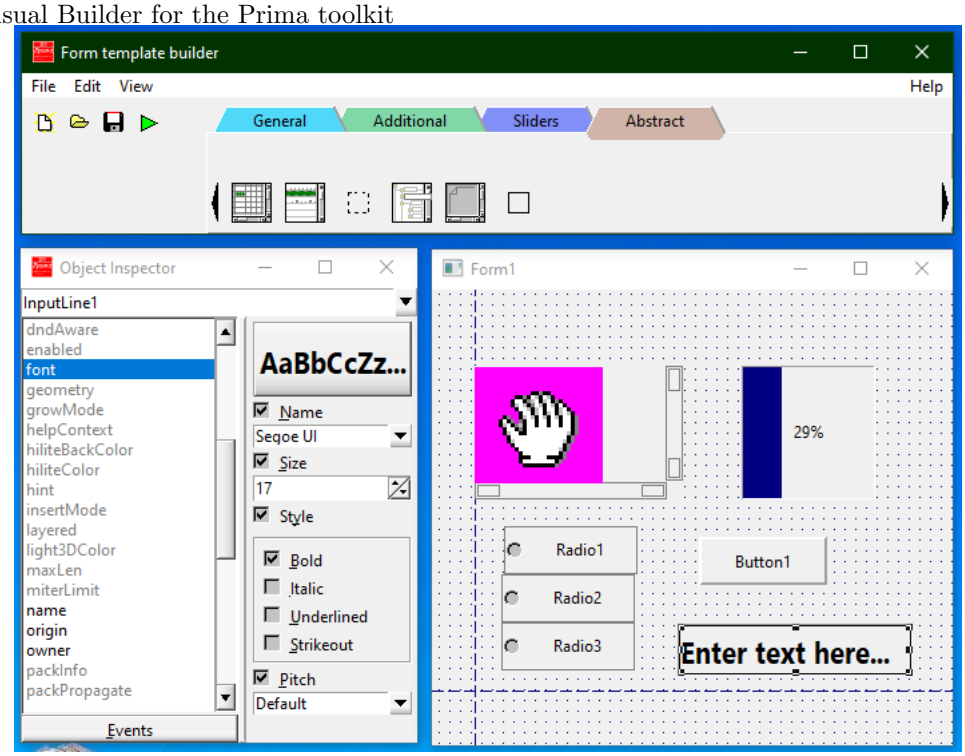
**walk BLOCK, %OPTIONS**

Cycles through the block opcodes, calls supplied callbacks on each.

# 7 Visual Builder

## 7.1 VB

Visual Builder for the Prima toolkit




### Synopsis

Run the `VB` command in your terminal

### Description

Visual Builder is a RAD-style suite for designing forms using the Prima toolkit. It provides a rich set of perl-based widgets which can be inserted into a window-based form by simple actions. The form can be stored in a file and loaded by either a user program or a simple wrapper, `utils/prima-fmview.pl`; the form can be also stored as a valid perl program.

A form file has the *.fm* extension and can be loaded fairly simply by using the the *Prima::VB::VBLoader* section module. The following code is the only content of the `prima-fmview.pl` program:

```
use Prima qw(Application VB::VBLoader);
```

```
my $ret = Prima::VBLoad( $ARGV[0] );
die "$@\n" unless $ret;
$ret-> execute;
```

Such code is usually sufficient for executing a form file.

## Help

The builder provides three main windows, that are used for interactive design. These are called *main panel*, *object inspector*, and *form window*. When the builder is started, the form window is empty.

The main panel consists of the menu bar, speed buttons, and the widget buttons. If the user presses a widget button and then clicks the mouse on the form window, the selected widget is inserted into the form and becomes a child of the form window. If the click was made on a visible widget in the form window, the newly inserted widget becomes a child of that widget. After the widget is inserted, its properties are accessible in the object inspector window.

The menu bar contains the following commands:

**File**

**New**

Closes the current form and opens a new empty form. If the old form was not saved, the user is asked if the changes made are to be saved.

This command is an alias to the 'new file' icon on the panel.

**Open**

Invokes the file open dialog so a *.fm* form file can be opened. After a successful file load, all form widgets are visible and available for editing.

This command is an alias to the 'open folder' icon on the panel.

**Save**

Stores the form into a file. The user here can select a type of the file to be saved. If the form is saved as a *.fm* form file then it can be re-loaded either in the builder or a user program ( see the *Prima::VB::VBLoader* section for details ). If the form is saved as a *.pl* program, then it can not be loaded; instead, the program can be run immediately without the builder or any supplementary code.

This command is an alias to the 'save on disk' icon on the panel.

**Save as**

Same as the *Save* entry, except that a new name or type of file is asked every time the command is invoked.

**Close**

Closes the form and removes the form window. If the form window was changed, the user is asked if the changes made are to be saved.

**Edit**

**Copy**

Copies the selected widgets into the clipboard so they can be inserted later by using the the *Paste* entry command. The form window itself can not be copied, only the widgets it contains.

**Paste**

Reads the information put by the builder the *Copy* entry command into the clipboard and inserts the widgets into the form window. The child-parent relation is kept by the names of the widgets; if the widget with the name of the parent of the clipboard-read widgets is not found, the widgets are inserted into the form window. The form window is not affected by this command.

**Delete**

Deletes the selected widgets. The form window itself can not be deleted.

**Select all**

Selects all of the widgets inserted in the form window except the form window itself.

**Duplicate**

Duplicates the selected widgets. The form window is not affected by this command.

**Align**

This menu item contains z-ordering actions that are performed on selected widgets. These are:

```
Bring to front
Send to back
Step forward
Step backward
Restore order
```

**Change class**

Changes the class of the selected widget. This is an advanced option and can lead to confusion or errors if the default widget class and the supplied class differ too much. It is used when the widget that has to be inserted is not present in the builder installation. Also, it is called implicitly when the loaded form does not contain a valid widget class; in such case the *Prima::Widget* class is assigned.

**Creation order**

Opens the dialog that manages the creation order of the widgets. It is not that important for the widget child-parent relation, since the builder tracks these, and does not allow a child to be created before its parent. However, the explicit order might be helpful in a case when, for example, the `tabOrder` property is left to its default value, so it is assigned according to the order of widget creation.

**Toggle lock**

Changes the lock status for selected widgets. The lock, if set, prevents widgets from being selected by the mouse to avoid occasional positional changes. This is useful when a widget is used as an owner for many sub-widgets.

The ctrl+mouse combination click locks and unlocks widgets.

**View**

**Object inspector**

Brings the object inspector window, if it was hidden or closed.

**Add widgets**

Opens the file dialog to install additional VB widgets. The modules are used for providing custom widgets and properties for the builder. As an example, the *Prima/VB/examples/Widgety.pm* module is provided with the builder and the toolkit. Look inside this file for the implementation details.

**Reset guidelines**

Resets the guidelines on the form window into the center.

**Snap to guidelines**

Specifies if the moving and resizing widget actions must treat the form window guidelines as snapping areas.

**Snap to grid**

Specifies if the moving and resizing widget actions must use the form window grid granularity instead of the pixel granularity.

**Run**

This command hides the form and object inspector windows and 'executes' the form as if it would be run by `prima-fmview.pl`. The execution session ends either by closing the form window or by calling the the *Break* entry command.

This command is an alias to the 'run' icon on the panel.

**Break**

Explicitly terminates the execution session initiated by the the *Run* entry command.

**Help**

**About**

Displays the information about the visual builder.

**Help**

Displays the information about the usage of the visual builder.

**Widget property**

Invokes the help viewer on the the *Prima::Widget* section manpage and tries to open the topic corresponding to the current selection of the object inspector property or event. While the manpage covers far not all ( but still many ) properties and events, it is still a little bit more convenient than nothing.

**Form window**

The form widget is the common parent for all widgets created by the builder. The form window provides the following basic navigation.

**Guidelines**

The form window contains two guidelines, the horizontal and the vertical, drawn as blue dashed lines. Dragging with the mouse can move these lines. If the menu option the *Snap to guidelines* entry is checked, the widgets' moving and sizing operations treat the guidelines as snapping areas.

**Selection**

A widget can be selected by clicking with the mouse on it. There can be more than one selected widget at a time, or none at all. To explicitly select a widget in addition to the already selected ones, hold the `shift` key while clicking on a widget. This combination also deselects the widget. To select all widgets on the form window, call the the *Select all* entry command from the menu. To prevent widgets from being occasionally selected, lock them with the "Edit/Toggle lock" command or Ctrl+mouse click.

**Moving**

Dragging the mouse can move the selected widgets. Widgets can be snapped to the grid or the guidelines during the move. If one of the moving widgets is selected in the object inspector window, the coordinate changes are reflected in the `origin` property.

If the `Tab` key is pressed during the move, the mouse pointer is changed between three states, each reflecting the currently accessible coordinates for dragging. The default accessible coordinates are both the horizontal and the vertical; the other two are the horizontal only and the vertical only.

### Sizing

The sizeable widgets can be dynamically resized. Only one widget at a time can be resized. If the resized widget is selected in the object inspector window, the size changes are reflected in the `size` property.

### Context menus

The right-click (or the other system-defined pop-up menu invocation command) provides the menu, identical to the main panel's the *Edit* entry submenu.

The alternative context menus can be provided with some widgets ( for example, `TabbedNotebook` ), and are accessible with the `control + right click` combination.

### Object inspector window

The inspector window reflects the events and properties of a widget. To explicitly select a widget, it must be either clicked by the mouse on the form window or selected in the widget combo box. Depending on whether properties or events are selected, the left panel of the inspector provides a list of properties or events, and the right panel - a value of the currently selected property or event. To toggle between the properties and the events use the button below the list.

The adjustable properties of a widget include an incomplete set of the properties returned by the class method `profile_default` ( for a detailed explanation see the *Prima::Object* section). Among these are such basic properties as `origin`, `size`, `name`, `color`, `font`, `visible`, `enabled`, `owner`, and many others. Each property can be selected by the property selector; in such case the name of a property is highlighted in the list - that means, that the property is initialized. To remove a property from the initialization list, double-click on it, so it is grayed again. Some very basic properties as `name` can not be deselected. This is because no widgets of the same name can coexist simultaneously in the builder.

The events, much like the properties, are accessible for direct change. All the events provide a small editor, so the custom code can be supplied. This code is executed when the form is run or loaded via the `Prima::VB::VBLoader` interface.

The full explanation of properties and events is not provided here. It is not even the goal of this document because the builder can work with the widgets irrespective of their property or event capabilities; this information is provided by the toolkit. To read what each property or event means, use the documentation on the class of interest; the *Prima::Widget* section is a good start because it encompasses the basic `Prima::Widget` functionality. The other widgets are documented in their respective modules, for example, the `Prima::ScrollBar` documentation can be found in the *Prima::ScrollBar* section.

## 7.2 Prima::VB::VBLoader

Visual Builder file loader

### Description

The module provides functionality for loading resource files created by Visual Builder. After a successful load, the newly created window with all children is returned.

### Synopsis

A simple way to use the loader is as follows:

```
use Prima qw(Application VB::VBLoader);
my $form = Prima::VBLoad( './your_resource.fm',
        Form1 => { centered => 1 },
);
die $@ unless $form;
$form-> execute;
```

All form widgets can be supplied with custom parameters, combined in a hash of hashes, and passed as the second parameter to the `VBLoad()` function. The example above supplies values for the `::centered` property to the `Form1` widget, which is the default name for a form window created by Visual Builder. All other widgets are accessible by their names in a similar fashion; after the creation, the widget hierarchy can be accessed in the standard way:

```
$form = Prima::VBLoad( $fi,
        ....
        StartButton => {
                onMouseOver => sub { die "No start buttons here\n" },
        }
);
...
$form-> StartButton-> hide;
```

In case a form is to be included not from another data source, the the *AUTOFORM_REALIZE* entry method can be used to transform the array of hashes that describes the form widget hierarchy into a set of widgets:

```
$form = AUTOFORM_REALIZE( [ Form1 => {
        class   => 'Prima::Window',
        parent  => 1,
        profile => {
                name => 'Form1',
                size => [ 330, 421],
        }], {});
```

There are several examples of how the form files are used in the toolkit; for instance, the *Prima/PS/setup.fm* dialog is used by the `Prima::PS::Setup` package.

### API

#### Methods

**check_version HEADER**

Scans HEADER that the first line of the .fm file for the version info. Returns two scalars - the first is a boolean flag, which is set to 1 if the file can be used and loaded, 0 otherwise. The second scalar is the version string.

### GO_SUB SUB [ @EXTRA_DATA ]

Depending on the value of the boolean flag `Prima::VB::VBLoader::builderActive` performs the following: if it is 1, the SUB text is returned as is. If it is 0, evaluates it in the `sub{}` context and returns the code reference. If the evaluation fails, EXTRA_DATA is stored in the `Prima::VB::VBLoader::eventContext` array and the exception is re-thrown. `Prima::VB::VBLoader::builderActive` is an internal flag that helps the Visual Builder to use the module interface without actual evaluation of the SUB code.

### AUTOFORM_REALIZE WIDGETS, PARAMETERS

WIDGETS is an array reference which contains evaluated data of the content of a .fm file, assuming its format is preserved. PARAMETERS is a hash reference with custom parameters passed to widgets during the creation. The widgets are distinguished by their names. Visual Builder ensures that no widgets have equal names.

`AUTOFORM_REALIZE` creates a tree of widgets and returns the root window which is usually named `Form1`. It automatically resolves parent-child relations, so the order in which WIDGETS are specified does not matter. Moreover, if a parent widget is passed as a parameter to a child widget, the parameter is deferred and passed only after the creation using the `::set` call.

During the parsing and the creation process, some internal notifications may be invoked. These notifications (events) are stored in the .fm file and usually provide class-specific loading instructions. See the *Events* entry for details.

### AUTOFORM_CREATE FILENAME, %PARAMETERS

Reads FILENAME in the .fm file format, checks its version, loads, and creates a widget tree. Upon successful load, the root widget is returned. The parsing and creation are performed by calling `AUTOFORM_REALIZE`. If loading fails, `die()` is called.

### Prima::VBLoad FILENAME, %PARAMETERS

A wrapper around `AUTOFORM_CREATE` is exported in the `Prima` namespace. FILENAME can be specified either as a file system path name or as a relative module name. In a way,

```
Prima::VBLoad( 'Module::form.fm' )
```

and

```
Prima::VBLoad(
        Prima::Utils::find_image( 'Module', 'form.fm'))
```

are identical. If the procedure finds that FILENAME is a relative module name it calls `Prima::Utils::find_image` automatically. To tell explicitly that FILENAME is a file system path name, FILENAME must be prefixed with the < symbol ( the syntax is influenced by `CORE::open` ).

%PARAMETERS is a hash with custom parameters passed to the widgets during the creation. The widgets are distinguished by their names. Visual Builder ensures that no widgets have equal names.

If the form file loaded successfully returns the form object reference. Otherwise `undef` is returned and the error string is stored in the `$@` variable.

### Events

The events stored in the .fm file are executed during the loading process. The module provides no functionality for supplying extra events during the load. This interface is useful only for developers of new widget classes for Visual Builder.

The events section is located in the `actions` section of the widget entry. There can be more than one event of each type, registered to different widgets. NAME parameter is a string with the name of the widget. INSTANCE is a hash created during load for every widget, and is provided to keep internal event-specific or class-specific data there. The `extras` section of the widget entry is present there as the place to store local data.

**Begin NAME, INSTANCE**

Called before the creation of a widget tree.

**FormCreate NAME, INSTANCE, ROOT_WIDGET**

Called after the creation of the form, which reference is contained in ROOT_WIDGET.

**Create NAME, INSTANCE, WIDGET.**

Called after the creation of a widget. The newly created widget is passed in WIDGET

**Child NAME, INSTANCE, WIDGET, CHILD_NAME**

Called before a child of WIDGET is created with CHILD_NAME as a name.

**ChildCreate NAME, INSTANCE, WIDGET, CHILD_WIDGET.**

Called after a child of WIDGET is created; the newly created widget is passed in CHILD_WIDGET.

**End NAME, INSTANCE, WIDGET**

Called after the creation of all widgets is finished.

## File format

The idea of the format of the .fm file is that it should be evaluated by the perl `eval()` call without special manipulations and kept as plain text. The file starts with a header which is a #-prefixed string, and contains a signature, the version of file format, and the version of the creator of the file:

```
# VBForm version file=1 builder=0.1
```

The header can also contain additional headers, also prefixed with #. These can be used to tell the loader that another perl module is needed to be loaded before the parsing; this is useful, for example, if a constant is declared in the module.

```
# [preload] Prima::ComboBox
```

The main part of a file is enclosed in a `sub{}` statement. After evaluation, this sub returns an array of paired scalars where each first item is a widget name and the second item is a hash of its parameters and other associated data:

```
sub
{
        return (
                'Form1' => {
                        class   => 'Prima::Window',
                        module  => 'Prima::Classes',
```

```
                                parent => 1,
                                code   => GO_SUB('init()'),
                                profile => {
                                        width => 144,
                                        name => 'Form1',
                                        origin => [ 490, 412],
                                        size => [ 144, 100],
                        }},
                );
        }
```

The hash has several predefined keys:

**actions HASH**

Contains a hash of events. The events are evaluated via the `GO_SUB` mechanism and executed during the creation of the widget tree. See the *Events* entry for details.

**code STRING**

Contains the code executed before the form is created. This key is present only on the root widget record.

**class STRING**

Contains the name of the class to be instantiated.

**extras HASH**

Contains class-specific parameters used by the events.

**module STRING**

Contains the name of the perl module that contains the class. The module will be `use`'d by the loader.

**parent BOOLEAN**

A boolean flag; is set to 1 for the root widget only.

**profile HASH**

Contains the profile hash passed as a set of parameters to the widget during its creation. If custom parameters are passed to `AUTOFORM_CREATE`, these are merged with `profile` ( the custom parameters take precedence ) before passing the result to the `new()` call.

## 7.3 cfgmaint

Configuration tool for Visual Builder

### Syntax

cfgmaint [ -rbxop ] command object [ parameters ]

### Description

Maintains configuration of the widget palette in the Visual Builder. The widget palette can be stored in the system-wide and local user config files. `cfgmaint` allows adding, renaming, moving, and deleting the classes and pages in the Visual Builder widget palette.

### Usage

`cfgmaint` is invoked with the `command` and `object` arguments where the `command` defines the action to be taken and `object` the object to be handled.

#### Options

**-r**

Write configuration to the system-wide config file

**-b**

Read configuration from both the system-wide and user config files

**-x**

Do not write backups

**-o**

Read-only mode

**-p**

Execute `use Prima;` code before start. This option might be necessary when adding a module that relies on the toolkit but does not invoke the code itself.

#### Objects

**m**

Selects a module. Valid for the add, list, and remove commands.

**p**

Selects a page. Valid for all commands.

**w**

Selects a widget. Valid for the list, remove, rename, and move commands.

### Commands

**a**

Adds a new object to the configuration. Can be either a page or a module.

**d**

Removes an object.

**l**

Prints the object name. In case the object is a widget, prints all registered widgets. If the string is specified as an additional parameter, it is treated as a page name, and only widgets from the page are printed.

**r**

Renames the object to a new name, which is passed as an additional parameter. Can be either a widget or a page.

**m**

If the `object` is a widget, relocates one or more widgets to a new page. If the `object` is a page, moves the page before the page specified as an additional parameter, or to the end if no additional page is specified.

## Example

Add a new module to the system-wide configuration:

```
cfgmaint -r a m CPAN/Prima/VB/New/MyCtrls.pm
```

List widgets that are present in both config files:

```
cfgmaint -b l w
```

Rename a page:

```
cfgmaint r p General Basic
```

## Files

*Prima/VB/Config.pm*, *~/.prima/vbconfig*

## 7.4 Prima::VB::CfgMaint

Configures the widget palette in the Visual Builder

### Description

The module is used by the Visual Builder and `prima-cfgmaint` programs, to configure the Visual Builder widget palette. The installed widgets are displayed in the main panel of the Visual Builder and can be maintained by the `prima-cfgmaint` program.

### Usage

The widget palette configuration is contained in two files - the system-wide `Prima::VB::Config` and the user `~/.prima/vbconfig`. The user config file takes precedence when loaded by the Visual Builder. The module can select either configuration by assigning the `$systemWide` boolean property.

The widgets are grouped in pages which are accessible by their names.

New widgets can be added to the palette by calling the `add_module` method which accepts a perl module file as its first parameter. The module must conform to the VB-loadable format.

### Format

This section describes the format of a module with VB-loadable widgets.

The module must define a package with the same name as the module. In the package, the `class` sub must be declared, that returns an array or paired scalars, where each first item in the pair corresponds to the widget class and the second to the hash which contains the class loading information, and must contain the following keys:

**class STRING**

Name of the package which represents the original widget class in the Visual Builder. This is usually a lightweight package that does not contain all the functionality of the original class but is capable of visually reflecting changes to the most useful class properties.

**icon PATH**

Sets the image file with the class icon. PATH provides an extended syntax for indicating a frame index if the image file is multiframed: the frame index is appended to the path name with the `:` character prefix, for example: `"NewWidget::icons.gif:2"`.

**module STRING**

Sets the module name that contains `class`.

**page STRING**

Sets the default palette page where the widget is to be stored. The current implementation of the Visual Builder provides four pages: `General,Additional,Sliders,Abstract`. If the page is not present, a new page is automatically created when the widget class is registered there.

**RTModule STRING**

Sets the module name that contains the original widget class.

The reader is urged to explore *Prima::VB::examples::Widgety* file which contains an example class `Prima::SampleWidget`, its VB-representation, and an example property `lineRoundStyle`.

## API

### Methods

**add_module FILE**

Reads the FILE module and loads all VB-loadable widgets from it.

**classes**

Returns declaration of all registered classes as a string ( see the *Format* entry ).

**open_cfg**

Loads class and pages information from either the system-wide or the user configuration file. If succeeds, the information is stored in the `@pages` and `%classes` variables ( the old information is lost ) and returns 1. If fails, returns 0 and a string with the error explanation; `@pages` and `%classes` content is undefined.

**pages**

Returns an array of page names

**read_cfg**

Reads information from both the system-wide and user configuration files and merges the information. If succeeds, returns 1. If fails, returns 0 and a string with the error explanation.

**reset_cfg**

Erases all information about pages and classes.

**write_cfg**

Writes either the system-wide or the user configuration file. If the `$backup` flag is set to 1, the old file is renamed to a `.bak` file. If succeeds, returns 1. If fails, returns 0 and a string with the error explanation.

## Files

*Prima::VB::Config.pm*, `~/.prima/vbconfig`.

# 8 PostScript printer interface

## 8.1 Prima::PS::PostScript

PostScript interface to Prima::Drawable

### Synopsis

Recommended usage:

```
use Prima::PS::Printer;
my $x = Prima::PS::File-> new( file => 'out.ps');
```

or

```
my $x = Prima::PS::FileHandle-> new( handle => \&STDOUT);
```

Low-level:

```
use Prima::PS::PostScript;
my $x = Prima::PS::PostScript-> create( onSpool => sub {
        open F, ">> ./test.ps";
        print F $_[1];
        close F;
});
```

Printing:

```
die "error:$@" unless $x-> begin_doc;
$x-> font-> size( 30);
$x-> text_out( "hello!", 100, 100);
$x-> end_doc;
```

### Description

Implements the Prima library interface to PostScript level 2 document language. The module is designed to be compliant with the Prima::Drawable interface. All properties' behavior is as same as Prima::Drawable's except those described below.

#### Inherited properties

**resolution**

Can be set while the object is in the normal stage - cannot be changed if the document is opened. Applies to implementation of the fillPattern property and general pixel-to-point and vice versa calculations

**alpha**

`alpha` is not implemented

### Specific properties

**copies**

The number of copies that the PS interpreter should print

**grayscale**

could be 0 or 1

**pageSize**

The physical page dimension, in points

**pageMargins**

Non-printable page area, an array of 4 integers: left, bottom, right, and top margins in points.

**reversed**

if 1, a 90 degrees rotated document layout is assumed

### Internal methods

**emit**

Can be called for direct PostScript code injection. Example:

```
$x-> emit('0.314159 setgray');
$x-> bar( 10, 10, 20, 20);
```

**pixel2point and point2pixel**

Helpers for translation from pixels to points and vice versa.

**fill & stroke**

Wrappers for PS outline that are expected to be either filled or stroked. Apply colors, line, and fill styles if necessary.

**spool**

Prima::PS::PostScript is not responsible for the output of the generated document, it only calls the `::spool` method when the document is closed through an `::end_doc` call. By default discards the data. The `Prima::PS::Printer` class handles the spooling logic.

**fonts**

Calls Prima::Application::fonts and returns its output filtered so that only the fonts that support the `iso10646-1` encoding are present. This effectively allows only unicode output.

## 8.2 Prima::PS::PDF

PDF interface to Prima::Drawable

### Synopsis

Recommended usage:

```
use Prima::PS::Printer;
my $x = Prima::PS::PDF::File-> new( file => 'out.pdf');
```

or

```
my $x = Prima::PS::PDF::FileHandle-> new( handle => \&STDOUT);
```

Low-level:

```
use Prima::PS::PDF;
my $x = Prima::PS::PDF-> create( onSpool => sub {
        open F, ">> ./test.pdf";
        binmode F;
        print F $_[1];
        close F;
});
```

Printing:

```
die "error:$@" unless $x-> begin_doc;
$x-> font-> size( 30);
$x-> text_out( "hello!", 100, 100);
$x-> end_doc;
```

### Description

Implements the Prima library interface to PDF v1.4. The module is designed to be compliant with the Prima::Drawable interface. All properties' behavior is as same as Prima::Drawable's except those described below.

#### Inherited properties

**::resolution**

Can be set while the object is in the normal stage - cannot be changed if the document is opened. Applies to implementation of the fillPattern property and general pixel-to-point and vice versa calculations

#### Specific properties

**grayscale**

could be 0 or 1

**pageSize**

The physical page dimension, in points

**pageMargins**

Non-printable page area, an array of 4 integers: left, bottom, right, and top margins in points.

**reversed**

if 1, a 90 degrees rotated document layout is assumed

### Internal methods

#### pixel2point and point2pixel

Helpers for translation from pixel to points and vice versa.

#### spool

Prima::PS::PDF is not responsible for the output of the generated document, it only calls the `::spool` method when the document is closed through an `::end_doc` call. By default discards the data. The `Prima::PS::Printer` class handles the spooling logic.

#### fonts

Calls Prima::Application::fonts and returns its output filtered so that only the fonts that support the `iso10646-1` encoding are present. This effectively allows only unicode output.

## 8.3 Prima::PS::Printer

PostScript and PDF interfaces to Prima::Printer

### Synopsis

```
use Prima::PS::Printer;

my $x;
if ( $preview) {
        $x = Prima::PS::Pipe-> new( command => 'gv $');
} elsif ( $print_in_file) {
        $x = Prima::PS::File-> new( file => 'out.ps');
} elsif ( $print_on_device) {
        $x = Prima::PS::LPR-> new( args => '-d colorprinter');
} elsif ( $print_pdf_file ) {
        $x = Prima::PS::PDF::File-> new( file => 'out.pdf');
} else {
        $x = Prima::PS::FileHandle-> new( handle => \*STDOUT );
}
$x-> begin_doc;
$x-> font-> size( 300);
$x-> text_out( "hello!", 100, 100);
$x-> end_doc;
```

### Description

Realizes the Prima printer interface to PostScript level 2 document language through the Prima::PS::PostScript module, and to PDF v1.4 through the Prima::PS::PDF module. Allows different user profiles to be created and managed with the standard setup dialog. The module is designed to be compliant with the Prima::Printer interface.

Also contains convenience classes (File, LPR, Pipe) for non-GUI use.

### Printer options

Below are the settigns supported by the `options` method:

**Color STRING**

One of : `Color, Monochrome`

**Resolution INTEGER**

Dots per inch.

**PageSize STRING**

One of: A*integer*, B*integer*, `Executive, Folio, Ledger, Legal, Letter, Tabloid, US Common #10 Envelope`.

**Copies INTEGER**

(not applicable to PDF)

**Scaling FLOAT**

1 is 100%, 1.5 is 150%, etc.

**Orientation**

One of : `Portrait`, `Landscape`.

**MediaType STRING**

An arbitrary string that represents special attributes of the medium other than its size, color, and weight. This parameter can be used to identify special media such as envelopes, letterheads, or preprinted forms.

(not applicable to PDF)

**MediaColor STRING**

A string identifying the color of the medium.

(not applicable to PDF)

**MediaWeight FLOAT**

The weight of the medium in grams per square meter. "Basis weight" or or null "ream weight" in pounds can be converted to grams per square meter by multiplying by 3.76; for example, 10-pound paper is approximately 37.6 grams per square meter.

(not applicable to PDF)

**MediaClass STRING**

(Level 3) An arbitrary string representing attributes of the medium that may require special action by the output device, such as the selection of a color rendering dictionary. Devices should use the value of this parameter to trigger such media-related actions, reserving the MediaType parameter (above) for generic attributes requiring no device-specific action. The MediaClass entry in the output device dictionary defines the allowable values for this parameter on a given device; attempting to set it to an unsupported value will cause a configuration error.

(not applicable to PDF)

**InsertSheet BOOLEAN**

(Level 3) A flag specifying whether to insert a sheet of some special medium directly into the output document. Media coming from a source for which this attribute is Yes are sent directly to the output bin without passing through the device's usual imaging mechanism (such as the fuser assembly on a laser printer). Consequently, nothing painted on the current page is actually imaged on the inserted medium.

(not applicable to PDF)

**LeadingEdge BOOLEAN**

(Level 3) A value specifying the edge of the input medium that will enter the printing engine or imager first and across which data will be imaged. Values reflect positions relative to a canonical page in portrait orientation (width smaller than height). When duplex printing is enabled, the canonical page orientation refers only to the front (recto) side of the medium.

(not applicable to PDF)

**ManualFeed BOOLEAN**

A flag indicating whether the input medium is to be fed manually by a human operator (Yes) or automatically (No). A Yes value asserts that the human operator will manually feed media conforming to the specified attributes ( MediaColor, MediaWeight, MediaType, MediaClass, and InsertSheet). Thus, those attributes are not used to select from available media sources in the normal way, although their values may be presented to the human operator as an aid in selecting the correct medium. On devices that offer more than one manual feeding mechanism, the attributes may select among them.

(not applicable to PDF)

**TraySwitch BOOLEAN**

(Level 3) A flag specifying whether the output device supports automatic switching of media sources. When the originally selected source runs out of medium, some devices with multiple media sources can switch automatically, without human intervention, to an alternate source with the same attributes (such as PageSize and MediaColor) as the original.

(not applicable to PDF)

**MediaPosition STRING**

(Level 3) The position number of the media source to be used. This parameter does not override the normal media selection process described in the text, but if specified it will be honored - provided it can satisfy the input media request in a manner consistent with normal media selection - even if the media source it specifies is not the best available match for the requested attributes.

(not applicable to PDF)

**DeferredMediaSelection BOOLEAN**

(Level 3) A flag determining when to perform media selection. If Yes, media will be selected by an independent printing subsystem associated with the output device itself.

(not applicable to PDF)

**MatchAll BOOLEAN**

A flag specifying whether input media request should match to all non-null values - MediaColor, MediaWeight etc.

(not applicable to PDF)

# 9 Widget helpers

## 9.1 Prima::Widget::BidiInput

Heuristics for i18n input

### Description

Provides common functionality for the bidirectional input to be used in editable widgets

### Methods

**handle_bidi_input %OPTIONS**

Given `action` and `text` in `%OPTIONS`, returns new text and a suggested cursor position.

The following options are understood:

**action**

One of: backspace, delete, cut, insert, overtype

**at INTEGER**

Current cursor position, calculated in clusters

**glyphs Prima::Drawable::Glyphs object**

Shaped text

**n_clusters INTEGER**

The number of clusters in the text

**rtl BOOLEAN**

Set to 1 if the default input direction is RTL (right-to-left)

**text STRING**

The text to edit

## 9.2 Prima::Widget::Fader

Fading- in/out functions

### Description

The role implements fading effects in widgets

### Synopsis

```
use base qw(Prima::Widget Prima::Widget::Fader);
{
my %RNT = (
        %{Prima::Widget-> notification_types()},
        %{Prima::Widget::Fader-> notification_types()},
);
sub notification_types { return \%RNT; }
}

sub on_mouseenter { shift-> fader_in_mouse_enter }
sub on_mouseleave { shift-> fader_out_mouse_leave }

sub on_paint
{
        my ( $self, $canvas ) = @_;
        $canvas->backColor( $self-> fader_prelight_color( $self-> hiliteBackColor ));
        $canvas->clear;
}
```

### API

The API is currently under design so the parts that are documented are those that expected to be staying intact.

#### Methods

**fader_in_mouse_enter**

Initiates a fade-in transition, calls repaint on each step.

**fader_out_mouse_leave**

Initiates a fade-out transition, calls repaint on each step.

**fader_current_value**

Returns the current fader value in the range from 0 to 1. Returns `undef` if there is no current fading transition in effect

**fader_prelight_color $COLOR [, $MULTIPLIER ]**

Given a base `$COLOR`, increases (or decreases) its brightness according to `fader_current_value` and an eventual `$MULTIPLIER` that is expected to be in the range from 0 to 1.

**Events**

**FadeIn $ENDS_OK**

Called when `fader_in_mouse_enter` finishes the fading, the `$ENDS_OK` flag is set to 0 if the process was overridden by another fader call, 1 otherwise.

**FadeOut $ENDS_OK**

Called when `fader_out_mouse_leave` finishes the fading, the `$ENDS_OK` flag is set to 0 if the process was overridden by another fader call, 1 otherwise.

**FadeRepaint**

By default repaints the whole widget, but can be overloaded if only some widget parts need to reflect the fader effect.

## 9.3 Prima::Widget::GroupScroller

Optional automatic scroll bars

### Description

The class is used for widgets that contain optional scroll bars and provides means for their management. The class is the descendant of the *Prima::IntIndents* section and adjusts its the *indents* entry property when scrollbars are shown, hidden, or the *borderWidth* entry is changed.

The class does not provide range selection for the scrollbars; the descendant classes must implement that.

The descendant classes must follow the following guidelines:

- A class may provide `borderWidth`, `hScroll`, `vScroll`, `autoHScroll`, and `autoVScroll` property keys in profile_default() .
- A class' init() method must call the `setup_indents` method

  If a class overrides the `autoHScroll` and `autoVScroll` properties, these must be set to 0 before the initialization.
- If a class needs to overload one of the `borderWidth`, `hScroll`, `vScroll`, `autoHScroll`, and `autoVScroll` properties, it is mandatory to call the inherited properties.
- A class must implement the scroll bar notification callbacks: `HScroll_Change` and `VScroll_Change`.
- A class must not use the reserved variable names, which are:

```
{borderWidth}  - internal borderWidth storage
{hScroll}      - internal hScroll value storage
{vScroll}      - internal vScroll value storage
{hScrollBar}   - pointer to the horizontal scroll bar
{vScrollBar}   - pointer to the vertical scroll bar
{bone}         - rectangular widget between the scrollbars
{autoHScroll}  - internal autoHScroll value storage
{autoVScroll}  - internal autoVScroll value storage
```

  The reserved method names:

```
set_h_scroll
set_v_scroll
insert_bone
setup_indents
reset_indents
borderWidth
autoHScroll
autoVScroll
hScroll
vScroll
```

  The reserved widget names:

```
HScroll
VScroll
Bone
```

## Properties

**autoHScroll BOOLEAN**

Selects if the horizontal scrollbar is to be shown and hidden dynamically, depending on the widget layout.

**autoVScroll BOOLEAN**

Selects if the vertical scrollbar is to be shown and hidden dynamically, depending on the widget layout.

**borderWidth INTEGER**

Width of the border around the widget.

Depends on the `skin` property.

**hScroll BOOLEAN**

Selects if the horizontal scrollbar is visible. If it is, {`hScrollBar`} points to it.

**vScroll BOOLEAN**

Selects if the vertical scrollbar is visible. If it is, {`vScrollBar`} points to it.

**scrollBarClass STRING = Prima::ScrollBar**

A create-only property that allows to change the scrollbar class

**hScrollBarProfile, vScrollBarProfile HASH**

Create-only properties that allows to adjust the scrollbar parameters when the scrollbars are created

## Methods

**setup_indents**

The method is never called directly; it should be called whenever the widget layout is changed so that its indents are affected. The method is a request to recalculate indents, depending on the new widget layout.

The method is not reentrant; to receive this callback and update the widget layout that in turn can result in more `setup_indents` calls, overload `reset_indents` .

**reset_indents**

Called after `setup_indents` updates the internal widget layout, to give a chance to follow up the layout changes. Does not do anything by default.

## 9.4 Prima::Widget::Header

Multi-column header widget

### Description

The widget class provides functionality of several button-like caption tabs, that can be moved and resized by the user. The class was implemented to serve as a table header for list and grid widgets.

### API

#### Events

**Click INDEX**

Called when the user clicks on the tab positioned at INDEX.

**DrawItem CANVAS, INDEX, X1, Y1, X2, Y2, TEXT_BASELINE**

A callback to draw the tabs. CANVAS is the output object; INDEX is the index of a tab. X1,Y2,X2,Y2 are the coordinates of the boundaries of the tab rectangle; TEXT_BASELINE is a pre-calculated vertical position for eventual centered text output.

**MeasureItem INDEX, RESULT**

Stores in scalar referenced by RESULT the width or height ( depending on the the *vertical* entry property value ) of the INDEXth tab, in pixels.

**MoveItem OLD_INDEX, NEW_INDEX**

Called when the user moves a tab from its old location, specified by OLD_INDEX, to the NEW_INDEX position. By the time of the call, all internal structures are updated.

**SizeItem INDEX, OLD_EXTENT, NEW_EXTENT**

Called when the user resizes a tab in the INDEXth position. OLD_EXTENT and NEW_EXTENT are either the width or height of the tab, depending on the the *vertical* entry property value.

**SizeItems**

Called when more than one tab changes its extent. This might happen as a result of both user and programmatic actions.

#### Properties

**clickable BOOLEAN**

Selects if the user is allowed to click the tabs.

Default value: 1

**dragable BOOLEAN**

Selects if the user is allowed to move the tabs.

Default value: 1

**items ARRAY**

An array of scalars representing the internal data of the tabs. By default, the scalars are treated as text strings.

**minTabWidth INTEGER**

A minimal extent in pixels a tab must occupy.

Default value: 2

**offset INTEGER**

An offset on the major axis ( depends on the the *vertical* entry property value ) that the widget is drawn with. Used for the conjunction with list widgets ( see the *Prima::DetailedList* section ), when the list is horizontally or vertically scrolled.

Default value: 0

**pressed INTEGER**

Contains the index of the currently pressed tab. A -1 value is selected when no tabs are pressed.

Default value: -1

**scalable BOOLEAN**

Selects if the user is allowed to resize the tabs.

Default value: 1

**vertical BOOLEAN**

If 1, the tabs are aligned vertically; the the *offset* entry, the *widths* entry property, and extent parameters of the callback notification assume the heights of the tabs.

If 0, the tabs are aligned horizontally, and the extent properties and parameters assume tab widths.

**widths ARRAY**

Array of integer values, corresponding to the extents of the tabs. The extents are widths ( `vertical` is 0 ) or heights ( `vertical` is 1 ).

### Methods

**tab2offset INDEX**

Returns the offset of the INDEXth tab ( without regard to the the *offset* entry property value ).

**tab2rect INDEX**

Returns four integers representing the rectangle area occupied by the INDEXth tab ( without regard to the the *offset* entry property value ).

## 9.5 Prima::Widget::IntIndents

Indenting support

### Description

Provides common functionality for the widgets that delegate part of their surface to border elements. For example, scroll bars and borders in a list box are such elements.

### Properties

**indents ARRAY**

Contains four integers specifying the breadth of decoration elements for each side. The first integer is the width of the left element, the second is the height of the lower element, the third is the width of the right element, and the fourth is the height of the upper element.

The property can accept and return the array either as four scalars, or as an anonymous array of four scalars.

#### Methods

**get_active_area [ TYPE = 0, WIDTH, HEIGHT ]**

Calculates and returns the extension of the area without the border elements, or the *active area*. The extension is related to the current size of a widget, however, can be overridden by specifying WIDTH and HEIGHT. TYPE is an integer, indicating the requested type of calculation:

**TYPE = 0**

Returns four integers, defining the area in the inclusive-exclusive coordinates.

**TYPE = 1**

Returns four integers, defining the area in the inclusive-inclusive coordinates.

**TYPE = 2**

Returns two integers, the size of the area.

## 9.6 Prima::Widget::Link

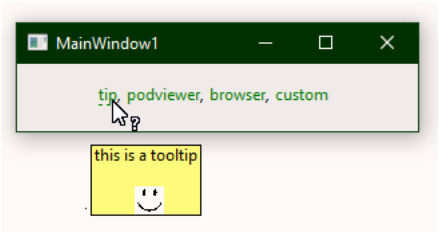
Routines for interactive links

### Description

The class can be used in widgets that need to feature *links*, i e highlighted rectangles, usually with a line of text. When the user moves the mouse or clicks on a link, depending on the link type, various actions can be executed. A "tooltip" link can display a hint with (rich) text, and a "hyperlink" link can open a browser or a pod viewer. The programmer can also customize these actions.

### Synopsis

```
use Prima qw(Label Application);

my $main_window = Prima::MainWindow->new( size => [400, 100] );
$main_window->insert( Label =>
  centered => 1,
  text   => \ "the I<tip:$0ttt|tip> entry, the I<pod:Prima|podviewer> entry, the I<http:google.
  onLink => sub { print "$_[2]\n" },
);
run Prima;

=pod

=head1 ttt

this is a tooltip

=for podview <img src="data:base64">
R0lGODdhFgAVAIAAAAAAAP///ywAAAAAFgAVAIAAAAD///8CLIyPqcutsKALQKI6qT11R69lWDJm
5omm6jqBjucycEx+bVOSNNf1+/NjCREFADs=
```




### Link types

Link types can be set with the *url* syntax. Four recognized link types behave differently

#### Tooltips

These are not links in the strict sense, as clicking on them doesn't cause any action, however when the user hovers the mouse over a tooltip, the module loads the pod content from the URL and displays it as a hint.

The idea behind this feature is to collect all tooltip cards in a pod section and reference them in the text like in the example code in the *Synopsis* entry above.

Syntax: `L<tip://FILEPATH_OR_MODULE/SECTION>` or `L<tip://FILEPATH_OR_MODULE>` where `FILEPATH_OR_MODULE` can refer either to a file (path with slashes/backslashes) or a perl module (with `::`s ).

The tooltip text, when selected, is underscored by a dashed line, vs all other link types that use a solid line.

### Pod sections

These links display a pod section preview like the tooltip but also open a pod viewer with the referred section when clicked on.

Syntax: `L<pod://FILEPATH_OR_MODULE/SECTION>` or `L<pod://FILEPATH_OR_MODULE>` where `FILEPATH_OR_MODULE` can refer either to a file (path with slashes/backslashes) or a perl module (with `::`s ).

### Hyperlinks

Links with schemes `ftp://`, `http://`, and `https://` open a browser when clicked on.

### Custom links

All other URLs, not matched by either scheme above, are expected to be handled programmatically. The preview, if any, should be handled by the `LinkPreview` event, and the mouse click by the `Link` event.

See the *Events* entry below.

## Usage

Since `Prima::Widget::Link` is not a widget by itself but a collection of routines in a class, an object of such class should be instantiated programmatically and attached to an *owner* widget that needs to display links.

The owner widget needs to call the mouse and paint methods from inside its `on_mousedown` etc relevant events. The owner widget class might also want to overload link events, see below how.

## Markup

the *Prima::Drawable::Markup* section understands the `L<..|..>` command, which, unlike perlpod, is formatted with its arguments reversed, to stay consistent with the other markup commands (i e it is `L<http://google.com|search>`, not `L<search|http://google.com>` .

The simple way to incorporate rich text in both the widget and link handler is to use `Prima::Drawable::Markup` to handle the markup parsing and use the resulting object from the same class both for widget drawing and for the link reactions. One just needs to add `markup =` $markup_object > to `Prima::Widget::Link`-new() >.

## API

### Properties

#### rectangles

Contains an array of rectangles in arbitrary coordinates that could be used to map screen coordinates to a URL. Filled automatically.

#### references

An array of URLs

### Methods

**add_positions_from_blocks LINK_ID, BLOCKS, %DEFAULTS**

Used when the link object is not bound to any markup object but recalculation of the visual rectangle that the link occupies is needed due to change in formatting, f ex after a change in widget size, font size, etc. `%DEFAULTS` is sent internally to `tb::block_walk` which may need eventual default parameters.

Scans BLOCKS and add monotonically increasing LINK_ID to new link rectangles. Return new LINK_ID.

**clear_positions**

Clears the content of `rectangles`

**id2rectangles ID**

Returns rectangles mapped to a link ID. There can be more than 1 rectangle bound to a single link ID since link text could be f ex wrapped.

**open_podview URL**

Opens a pod viewer with the URL

**open_browser URL**

Opens a web browser with the URL

**reset_positions_markup BLOCKS, %DEFAULTS**

Used when the link object is bound to a markup object and recalculation of the visual rectangle that the link occupies is needed due to change in formatting, f ex after a change in widget size, font size, etc. `%DEFAULTS` is sent internally to `tb::block_walk` which may need eventual default parameters.

### Events

All events are sent to the owner, not to the link object itself, however, the `SELF` parameter which contains the link object is always the first parameter

**Link SELF, URL, BUTTON, MOD**

Sent to the owner, if any, from within the `on_mousedown` event to indicate that the link was pressed on.

**LinkPreview SELF, URL_REF**

Sent to the owner, if any, from within the `on_mousemove` event. The owner may want to fill URL_REF with (rich) text that will be displayed as a link preview

**LinkAdjustRect SELF, RECT_REF**

Since the owner may implement a scrollable view or any other view that has a coordinate system that is not necessarily consistent with the rectangles stored in the link object, this event will be called when a link rectangle needs to be mapped to the owner coordinates.

## 9.7 Prima::Widget::ListBoxUtils

Common paint routine for listboxes

### Description

Used internally by list-like widgets

## 9.8 Prima::Widget::MouseScroller

Auto-repeating mouse events

### Description

Implements routines for emulation of auto repeating mouse events. A code inside `MouseMove` callback can be implemented by the following scheme:

```
if ( mouse_pointer_inside_the_scrollable_area) {
        $self-> scroll_timer_stop;
} else {
        $self-> scroll_timer_start unless $self-> scroll_timer_active;
        return unless $self-> scroll_timer_semaphore;
        $self-> scroll_timer_semaphore( 0);
}
```

The class uses a semaphore {`mouseTransaction`}, which should be set to non-zero if a widget is in mouse capture state, and set to zero or `undef` otherwise.

The class starts an internal timer, which sets a semaphore and calls `MouseMove` notification when triggered. The timer is assigned the timeouts, returned by `Prima::Application::get_scroll_rate` ( see the **get_scroll_rate** entry in the *Prima::Application* section ).

### Methods

**scroll_timer_active**

Returns a boolean value indicating if the internal timer is started.

**scroll_timer_semaphore** [ **VALUE** ]

A semaphore, set to 1 when the internal timer was triggered. It is advisable to check the semaphore state to discern a timer-generated event from the real mouse movement. If VALUE is specified, it is assigned to the semaphore.

**scroll_timer_start**

Starts the internal timer.

**scroll_timer_stop**

Stops the internal timer.

## 9.9 Prima::Widget::Panel

Simple panel widget

Provides a simple panel widget capable of displaying a single line of centered text on a custom background. Probably this functionality is better to be merged with `Prima::Label`.

### Properties

**borderWidth INTEGER**

Width of 3d-shade border around the widget.

Default value: 1

**image OBJECT**

Selects the image to be drawn as a tiled background. If `undef` the background is drawn with the background color.

**imageFile PATH**

Sets the image FILE to be loaded and displayed. Is rarely used since does not return the success flag.

**raise BOOLEAN**

The style of the 3d-shade border around the widget. If 1, the widget is 'risen'; if 0 it is 'sunken'.

Default value: 1

**zoom INTEGER**

Selects the zoom level for the image display. The acceptable value range is between 1 and 10.

Default value: 1

## 9.10 Prima::Widget::RubberBand

Dynamic rubberbands

### Description

The motivation for this module was that I was tired of seeing corrupted screens on Windows 7 when dragging rubberbands in Prima code. Even though MS somewhere warned of not doing any specific hacks to circumvent the bug, I decided to give it a go anyway.

This module thus is a `Prima::Widget/rect_focus` with a safeguard. The only thing it can do is to draw a static rubberband - but also remember the last coordinates drawn, so cleaning and animation come for free.

The idea is that a rubberband object is meant to be a short-lived one: as soon as it gets instantiated it draws itself on the screen. When it is destroyed, the rubberband is erased too.

### Synopsis

```
use strict;
use Prima qw(Application Widget::RubberBand);

sub xordraw
{
        my ($self, @new_rect) = @_;
        $::application-> rubberband( @new_rect ?
                ( rect => \@new_rect ) :
                ( destroy => 1 )
        );
}

Prima::MainWindow-> new(
        onMouseDown => sub {
                my ( $self, $btn, $mod, $x, $y) = @_;
                $self-> {anchor} = [$self-> client_to_screen( $x, $y)];
                xordraw( $self, @{$self-> {anchor}}, $self-> client_to_screen( $x, $y));
                $self-> capture(1);
        },
        onMouseMove => sub {
                my ( $self, $mod, $x, $y) = @_;
                xordraw( $self, @{$self-> {anchor}}, $self-> client_to_screen( $x, $y)) i
        },
        onMouseUp => sub {
                my ( $self, $btn, $mod, $x, $y) = @_;
                xordraw if delete $self-> {anchor};
                $self-> capture(0);
        },
);

run Prima;
```

### API

#### new %properties

Creates a new RubberBand instance. See the description of its properties below.

### Properties

**breadth INTEGER = 1**

Defines the rubberband breadth in pixels.

**canvas = $::application**

Sets the painting surface, and also the widget (it must be a widget) used for drawing.

**clipRect X1, Y1, X2, Y2**

Defines the clipping rectangle in inclusive-inclusive coordinates. If set to [-1,-1,-1,-1], means no clipping is needed.

**rect X1, Y1, X2, Y2**

Defines the band geometry in inclusive-inclusive coordinates. The band is drawn so that its body is always inside these coordinates, no matter what the breadth is.

### Methods

**hide**

Hides the band

**has_clip_rect**

Checks whether clipRect contains an actual clipping rectangle or it is empty.

**set %profile**

Applies all properties

**left, right, top, bottom, width, height, origin, size**

The same shortcuts as in `Prima::Widget`, but read-only.

**show**

Shows the band

## Prima::Widget interface

The module adds a single method to the `Prima::Widget` namespace, `rubberband` (see example of use in the synopsis).

**rubberband(%profile)**

Instantiates a `Prima::RubberBand` object with `%profile`, also sets `canvas` to `$self` ( unless `canvas` is set explicitly ).

**rubberband()**

Returns the existing `Prima::RubberBand` object

**rubberband(destroy => 1)**

Destroys the existing `Prima::Widget::RubberBand` object

## 9.11 Prima::Widget::ScrollWidget

Scrollable generic document widget

### Description

`Prima::Widget::ScrollWidget` is a simple class that declares two pairs of properties, *delta* and *limit* for vertical and horizontal axes, which define the extensions of a virtual document. *limit* is the document dimension, and *delta* is the current offset.

`Prima::Widget::ScrollWidget` is a descendant of `Prima::Widget::GroupScroller`, and as well as its ascendant, provides the same user navigation by two scrollbars. The scrollbars' `partial` and `whole` properties are automatically updated when the document or widget extensions change.

`Prima::Widget::ScrollGroup` provides the capability of hosting other widgets inside, and also scrolling them. Useful for widget group panels that cannot fit in a window

### Prima::Widget::ScrollWidget

#### Properties

**deltas X, Y**

Selects the horizontal and vertical document offsets.

**deltaX INTEGER**

Selects the horizontal document offset.

**deltaY INTEGER**

Selects the vertical document offset.

**limits X, Y**

Selects the horizontal and vertical document extensions.

**limitX INTEGER**

Selects the horizontal document extension.

**limitY INTEGER**

Selects the vertical document extension.

#### Events

**Scroll DX, DY**

Called whenever the client area is to be scrolled. The default action calls `Widget::scroll` .

### Prima::Widget::ScrollGroup

#### Properties

**client**

Returns the parent widget to insert other widgets. The client size is fixed and is panned through the slave widget when scrolling. The client is unaffected by the eventual automated pack/place/growMode size alteration the parent or slave might be subjected to.

**clientClass**

A `clientClass` widget is inserted in the `slave` widget.

**slave**

Returns the slave widget. The slave widget covers the scrollable area and is otherwise just a normal `Prima::Widget` object that can be resized, moved, etc.

**slaveClass**

A `slaveClass` widget is inserted directly in the scroll group widget.

## 9.12 Prima::Widget::StartupWindow

A simplistic startup banner window

### Description

The module, when imported by the `use` call, creates a temporary window which appears with the 'loading...' text while the modules required by a program are loading. The window parameters can be modified by passing custom parameters after the `use Prima::Widget::StartupWindow` statement, that are passed to the `Prima::Window` class as creation parameters. The window is discarded by explicit unimporting of the module ( see the *Synopsis* entry ).

### Synopsis

```
use Prima;
use Prima::Application;
use Prima::Widget::StartupWindow; # the window is created here

use Prima::Buttons;
.... # lots of 'use' of other modules

no Prima::Widget::StartupWindow; # the window is discarded here
```

## 9.13 Prima::Widget::UndoActions

Undo and redo the content of editable widgets

### Description

Generic helpers that implement stored actions for undo/redo.

### Synopsis

```
package MyUndoableWidget;
use base qw(Prima::Widget Prima::Widget::UndoActions);

sub on_mousedown
{
        if ( $button == mb::Left ) {
                $self->begin_undo_group;
                $self->push_undo_action(text => $self->text);
                $self->text($self->text . '.');
                $self->end_undo_group;
        } else {
                $self->undo; # will call $self->text( old text )
        }
}
```

### Properties

**undoLimit INTEGER**

Sets the limit on the number of atomic undo operations. If 0, undo is disabled.

#### Methods

**begin_undo_group**

Opens a bracket for a group of actions that can be undone as a single operation. The bracket is closed by calling `end_undo_group`.

**can_undo, can_redo**

Return a boolean flag that reflects if the undo or redo actions could be done. Useful for graying a menu, f ex.

**end_undo_group**

Closes the bracket for a group of actions, that was previously opened by `begin_undo_group`.

**init_undo**

Should be called once, inside init()

**has_undo_action ACTION**

Checks whether there are any undo-able ACTIONs in the undo list.

**push_grouped_undo_action ACTION, @PARAMS**

Stores a single undo action where ACTION is a method to be called inside undo/redo, if any. Each action is added to the last undo group and will be removed/replayed together with the other actions in the group.

**push_undo_action ACTION, @PARAMS**

Stores a single undo action where ACTION is the method to be called inside undo/redo, if any. Each action is a single undo/redo operation.

**redo**

Re-applies changes, previously rolled back by `undo`.

**undo**

Rolls back changes into an internal array. The array size cannot extend the `undoLimit` value. In case `undoLimit` is 0 no undo actions can be made.

# 10 C interface to the toolkit

## 10.1 Prima::internals

Prima internal architecture

### Description

This document describes the internal structures of the Prima toolkit, its loading considerations, object and class representation, and C coding style.

### Bootstrap

#### Initializing

From the point of view of a perl script, Prima is no more but an average module that uses DynaLoader. As the `use Prima` code gets executed, the bootstrap procedure boot_Prima() is called. This procedure initializes all internal structures and built-in Prima classes. It also initializes all system-dependent structures, calling window_subsystem_init(). After that point, the Prima module is ready to use. All wrapping code for built-in functionality that can be seen from perl is in two modules - Prima::Const and Prima::Classes.

#### Constants

Prima defines a lot of constants for different purposes ( e.g. colors, font styles,0 etc). Prima does not follow perl naming conventions here, for the sake of simplicity. It is ( arguably ) easier to write cl::White rather than Prima::cl::White. As perl constants are functions to be called once ( that means that a constant's value is not defined until it is used first ), Prima registers these functions during the boot_Prima stage. As soon as perl code tries to get a constant's value, an AUTOLOAD function is called, which is banded inside Prima::Const. Constants are widely used both in C and perl code and are defined in *apricot.h* in such a way so that perl constant definitions come along with the C ones. As an example file event constants set is described here.

```
apricot.h:
  #define FE(const_name) CONSTANT(fe,const_name)
  START_TABLE(fe,UV)
  #define feRead      1
  FE(Read)
  #define feWrite     2
  FE(Write)
  #define feException 4
  FE(Exception)
  END_TABLE(fe,UV)
  #undef FE
```

```
Const.pm:
  package fe; *AUTOLOAD = \&Prima::Const::AUTOLOAD;
```

This code creates a structure filled with UVs ( unsigned integers ) and declares a register_fe_constants() function, which should be called at the boot_Prima stage. This way feRead becomes the C analog to the fe::Read in perl.

## Classes and methods

### Virtual method tables

Prima implementation of classes uses virtual method tables, or VMTs, to make the classes inheritable and their methods overridable. The VMTs are usual C structs, that contain pointers to functions. A set of these functions represents a class. This chapter is not about OO programming, you have to find a good book on it if you are not familiar with the OO concepts, but in short, because Prima is written in C, not in C++, it uses its own classes and objects implementation, so all object syntax is devised from scratch.

The built-in classes already contain all the information needed for method overloading, but when a new class is derived from an existing one, a new VMT has to be created as well. The actual sub-classing is performed inside build_dynamic_vmt() and build_static_vmt(). gimme_the_vmt() function creates a new VMT instance on the fly and caches the result for every new class that is derived from a Prima built-in class.

### C to Perl and Perl to C calling routines

The majority of Prima methods are written in C using XS perl routines, which represent a natural ( from a perl programmer's view ) way of C-to-Perl communication. *perlguts* manpage describes these functions and macros.

Note: Do not mix XS calls with the XS language ( *perlxs* manpage) - the latter is a meta-language for simplification of coding tasks and is not used in the Prima implementation.

It was decided not to code every function with XS calls, but instead use special wrapper functions ( also called "thunks") for every function that is called from within perl. Thunks are generated automatically by the gencls tool ( the *prima-gencls* section manpage ), and the typical Prima method consists of three functions, two of which are thunks.

The first function, say Class_init(char*), would initialize a class ( for example). It is written fully in C, so to be called from perl code a registration step must be taken for the second function, Class_init_FROMPERL(), that would look like this:

```
newXS( "Prima::Class::init", Class_init_FROMPERL, "Prima::Class");
```

Class_init_FROMPERL() is the first thunk, that translates the parameters passed from perl to C and the result back from the C function to perl. This step is almost fully automatized, so one never bothers about writing the XS code, the gencls utility creates the thunks code automatically.

Many C methods are called from within Prima C code using VMTs, but it is possible to override these methods from the perl code too. The actions for such a situation when a function is called from C but is an overridden method therefore must be taken. On that occasion, the third function Class_init_REDEFINED() is declared. Its task is a reverse from Class_init_FROMPERL() - it conveys all C parameters to perl and returns values from a perl function back to C. This thunk is also generated automatically by the gencls tool.

As one can notice, only basic data types can be converted between C and perl, and at some point, automated routines do not help. In such a situation data conversion code is written manually and is included in core C files. In the class declaration files these methods are prepended with the 'public' or 'weird' modifiers, while methods with no special data handling need to use the 'method' or 'static' modifiers.

Note: functions that are not allowed to be seen from perl should have the 'c_only' modifier, and do not need the thunk wrapping. These functions can nevertheless be overridden from C.

### Built-in classes

Prima defines the following built-in classes: (in hierarchy order)

```
Object
    Component
            AbstractMenu
                    AccelTable
                    Menu
                    Popup
            Clipboard
            Drawable
                    DeviceBitmap
                    Printer
                    Image
                            Icon
            File
            Region
            Timer
            Widget
                    Application
                    Window
```

These classes can be seen from perl with Prima:: prefix. Along with these, the Utils class is defined in a special way. Its only difference is that it cannot be used as a prototype for an object, and used merely as a package that binds functions. Classes that are not intended to be an object prototype are marked with the 'package' prefix while the other classes are marked with the 'object' prefix (see prima-gencls manpage).

## Objects

This chapter deals only with Prima::Object descendants, pure perl objects are not of interest here, so the 'object' term is thereafter referenced to be the Prima::Object descendant object. Prima employs blessed hashes as its objects.

### Creation

All built-in object classes and their descendants can be used for creating objects with perl semantics. Perl objects are created by calling bless(), but this is not enough to create Prima objects. Every Prima::Object descendant class therefore is equipped with the create() method, which allocates the object instance and calls bless() itself. Parameters that come to the create() call are formed into a hash and passed to the init() method, which is also present on every object. Note that although the perl-coded init() returns the hash, it is not seen in C code. This is a special consideration for the methods that have 'HV * profile' as a last parameter in their class declaration. The corresponding thunk copies the hash content back to the perl stack, using the parse_hv() and push_hv() functions.

Objects can be created from perl by using the following code example:

```
$obj = Prima::SampleObject-> create(
    name  => "Sample",
    index => 10,
);
```

and from C:

```
Handle obj;
HV * profile = newHV();
pset_c( name, "Sample");
pset_i( index, 10);
obj = Object_create("SampleObject", profile);
sv_free(( SV*) profile);
```

or even

```
create_object("SampleObject", "si",
    "name", "Sample",
    "index", 10
);
```

Convenience pset_XX macros assign a value of XX type to the hash key given as a first parameter, to a hash variable named profile. `pset_i` works with integers, `pset_c` - with strings, etc.

### Destruction

As well as the create() method, every object class has the destroy() method. An object can be destroyed either from perl

```
$obj-> destroy
```

or from C

```
Object_destroy( obj);
```

An object can be automatically destroyed when its reference count reaches 0. Note that the auto destruction would never happen if the object's reference count is not lowered after its creation. The code

```
--SvREFCNT( SvRV( PAnyObject(object)-> mate));
```

is required if the object is to be returned to perl. If that code is not called, the object still could be destroyed explicitly, but its reference would still live, resulting in a memory leak problem.

For the user code it is sufficient to overload done() and/or cleanup() methods, or just the onDestroy notification. It is highly recommended to avoid overloading the destroy() method since it can be called in a re-entrant fashion. When overloading done(), be prepared that it may be called inside init(), and deal with the semi-initialized object.

### Data instance

All Prima objects are blessed hashes, and the hash key __CMATE__ holds a C pointer to a memory that is occupied by a C data instance, or a "mate". It keeps all object variables and a pointer to the VMT. Every object has its own copy of the data instance, but the VMTs can be shared. To reach the C data instance gimme_the_mate() function is used. As the first argument, it accepts a scalar (SV*), which is expected to be a reference to a hash, and returns the C data instance if the scalar is a Prima object.

### Object life stages

There are several steps, or "stages", in every object's life cycle. Every stage is mirrored into PObject(self)-> stage integer variable, which can be one of the following csXXX constants:

**csConstructing**

The object is this initial stage until create() is finished. Right after init() is completed, the setup() method is called.

**csNormal**

After create() is finished and before destroy() starts. If an object is in either csNormal or csConstructing stage, the result of the Object_alive() method is non-zero.

**csDestroying**

The destroy() started. This stage runs the cleanup() and the done() methods.

**csFrozen**

cleanup() started.

**csFinalizing**

done() started

**csDead**

destroy() finished

## Coding techniques

### Accessing object data

C coding in Prima has no specific conventions, except when coding an object method. The object syntax for accessing the object's data instance is though fairly straightforward. For example, accessing the component field called 'name' can be done in several ways:

```
((PComponent) self)-> name; // classic C
PComponent(self)-> name;    // using PComponent() macro from apricot.h
var-> name;                 // using local var() macro
```

Object methods could be called in several ways:

```
(((PComponent) self)-> self)-> get_name( self); // classic C
CComponent(self)-> get_name( self);             // using CComponent() macro from apricot.h
my-> get_name( self);                           // using local my() macro
```

The calling of methods via the object's VMTs is preferred, compared to the direct call of Component_get_name(), primarily because get_name() is a method that can be overridden in the user code.

### Calling perl code

The call_perl_indirect() function accepts an object, a name of the method to be called, an argument format string, and an argument list. It has several wrappers for the ease of use, which are:

```
call_perl( Handle self, char * method, char * format, ...)
sv_call_perl( SV * object, char * method, char * format, ...)
cv_call_perl( SV * object, SV * code_reference, char * format, ...)
```

each character of the format string represents the type of the corresponding argument, using the following characters to encode types:

```
'i' - integer
's' - char *
'n' - float
'H' - Handle
'S' - SV *
'P' - Point
'R' - Rect
```

The format string can be prepended with the '<' character, in which case an SV * scalar ( always scalar, even if the code returns nothing or an rray ) value is returned. The caller is responsible for freeing the returned value.

### Exceptions

As described in the *perlguts* manpage, the G_EVAL flag is used in perl_call_sv() and perl_call_method() to indicate that an eventual exception should never be propagated automatically. The caller checks if the exception was taken place by evaluating the

```
SvTRUE( GvSV( errgv))
```

statement. It is guaranteed to be false if there was raised no exception. In some situations though, namely, when no perl_call_* functions are called or an error value is already assigned before calling code, there is a wrapping technique that keeps the eventual previous error message. Such code may look like this:

```
dG_EVAL_ARGS;                       // define arguments
....
OPEN_G_EVAL;                        // open brackets
// call code
perl_call_method( ... | G_EVAL);    // G_EVAL is necessary
if ( SvTRUE( GvSV( errgv)) {
    CLOSE_G_EVAL;                   // close brackets
    croak( SvPV_nolen( GvSV( errgv)));// propagate exception
    // no code is executed after croak
}
CLOSE_G_EVAL;                       // close brackets
...
```

This technique provides a workaround to a "false alarm" situation if SvTRUE( GvSV( errgv)) is already true before perl_call_method().

### Object protection

After the object destroy stage is completed, it is possible that the object's data instance is gone, and even a simple stage check might cause a segmentation fault. To avoid this, bracketing functions called `protect_object()` and `unprotect_object()` are used. protect_object() increments the reference count to the object instance, thus delaying its freeing until decrementing unprotect_object() is called.

All C code that references an object must check its stage after every routine that may potentially switch to perl code because the object might be destroyed inside the call. A typical code example would be like this:

```
int handle_object(Handle object) {
     int stage;
     protect_object( object);

     // call some perl code
     perl_call_method( object, "test", ...);

     stage = PObject(object)-> stage;
     unprotect_object( object);
     if ( stage != csNormal) return 0;

     // proceed with the object
     ...
     return 1;
}
```

Usually C code doesn't need to check the object stage before the call to perl is made because the gimme_the_mate() function returns NULL when the object's stage is csDead, and the majority of Prima C code is prepended with this call, thus rejecting invalid references on the early stage. If it is desired to get the C mate for objects that are in the csDead stage, use the `gimme_the_real_mate()` function instead.

### init

Object's method init() is responsible for setting all its initial properties to the given values, however, all code that is executed inside the init() must be aware that the object's stage is csConstructing. init() consists of two parts: calling of ancestor's init() and setting the properties. Examples are many in both C and perl code, but in short, it looks like this:

```
void
Class_init( Handle self, HV * profile)
{
   inherited init( self, profile);
   my-> set_index( pget_i( index));
   my-> set_name( pget_c( name));
}
```

pget_XX macros call croak() if the profile key is not present in the profile, but the mechanism guarantees that all keys that are listed in profile_default() are conveyed to the init(). For explicit checking of key presence the pexists() macro is used, and the pdelete() macro is used for the key deletion, although is it not recommended to use pdelete() inside init().

### Object creation and returning

As described in the previous sections, there are some precautions to be taken into account when an object is created inside C code. A piece of real code from DeviceBitmap.c would serve as an example:

```
static
Handle xdup( Handle self, char * className)
{
   Handle h;
   Point s;
   PDrawable i;

   // allocate a parameters hash
   HV * profile = newHV();
```

```
    // set all necessary arguments
    pset_H( owner,        var-> owner);
    pset_i( width,        var-> w);
    pset_i( height,       var-> h);
    pset_i( type,         (var-> type == dbtBitmap) ? imBW : imRGB);

    // create object
    h = Object_create( className, profile);

    // free profile, do not need it anymore
    sv_free(( SV *) profile);

    i = ( PDrawable) h;
    s = i-> self-> get_size( h);
    i-> self-> begin_paint( h);
    i-> self-> put_image_indirect( h, self, 0, 0, 0, 0, s.x, s.y, s.x, s.y, ropCopyPut);
    i-> self-> end_paint( h);

    // decrement reference count
    --SvREFCNT( SvRV( i-> mate));
    return h;
}
```

Note that all code that would use this xdup() has to increase and decrease the object's reference count if some perl functions are to be executed before returning the object to perl, otherwise it might get destroyed in the middle of the execution:

```
Handle x = xdup( self, "Prima::Image");
++SvREFCNT( SvRV( PAnyObject(x)-> mate)); // Code without these
CImage( x)-> type( x, imbpp1);
--SvREFCNT( SvRV( PAnyObject(x)-> mate)); // brackets is unsafe
return x;
```

### Attaching objects

The newly created object returned from C would be destroyed due perl's garbage cleaning mechanism right away, unless the object value is assigned to a scalar, for example.

Thus

```
$c = Prima::Object-> create();
```

and Prima::Object-> create;

have different results. But for some classes, namely Widget and its descendants, and also for Timer, AbstractMenu, Printer, and Clipboard the code above would have the same result - the objects would not be killed. That is because these objects call the Component_attach() method during the init-stage, automatically increasing their reference count. The Component_attach() and its reverse Component_detach() keep a list of objects that are attributed to each other. An object can be attached to more than object at a time, but cannot be attached more than once to another object.

### Notifications

All descendats of the Prima::Component class are equipped with a mechanism that allows user callback routines to be called when corresponding events occur. A very similar mechanism is used typically everywhere in the event-driven programming. Component_notify() is used to call the user notifications; its format string has the same format as accepted by perl_call_indirect(). The only difference is that it always has to be prepended with '<s', - this way the call success flag can be returned, and the first parameter must be the name of the notification.

```
Component_notify( self, "<sH", "Paint", self);
Component_notify( self, "<sPii", "MouseDown", self, point, int, int);
```

The notification mechanism keeps another reference list, similar to the attach-detach mechanism so that notifications can be attributed to different objects. Objects entering the list don't get their reference counter changed.

### Multi-property setting

Prima::Object method set() is designed to assign several properties at once. Sometimes it is more convenient to write

```
$c-> set( index => 10, name  => "Sample" );
```

than to invoke several methods one by one. The set() method executes these calls itself, but for the performance reasons it is possible to overload this method and code special conditions for the mult-assignment. As an example, here's Prima::Image type conversion code using this technique:

```
void
Image_set( Handle self, HV * profile)
{
   ...
   if ( pexist( type))
   {
      int newType = pget_i( type);
      if ( !itype_supported( newType))
         warn("Invalid image type requested (%08x) in Image::set_type",
            newType);
      else
         if ( !opt_InPaint)
            my-> reset( self, newType, pexist( palette) ?
               pget_sv( palette) : my->get_palette( self));
      pdelete( palette);
      pdelete( type);
   }
   ...
   inherited set ( self, profile);
}
```

Here, if the type conversion is performed along with the palette change, some efficiency is gained by supplying both the 'type' and 'palette' parameters at once. Moreover, because the ordering of the fields is not determined by default ( although that is done by supplying the '__ORDER__' hash key to set() }, it can easily be discovered that

```
$image-> type( $a);
$image-> palette( $b);
```

and

```
$image-> palette( $b);
$image-> type( $a);
```

produce different results. Therefore the only solution here is to code Class_set() explicitly.

If it is desired to specify the exact order of how atomic properties have to be called, __ORDER__ anonymous array has to be added to the set() parameters.

```
$image-> set(
   owner => $xxx,
   type  => 24,
   __ORDER__ => [qw( type owner)],
);
```

## API reference

### Variables

**primaObjects, PHash**

Hash with all prima objects, where keys are their data instances

**application, Handle**

Pointer to the application. There can be only one Application instance at a time, or none at all.

### Macros and functions

**dG_EVAL_ARGS**

Defines a variable for $@ value storage

**OPEN_G_EVAL, CLOSE_G_EVAL**

Brackets for exception catching

**build_static_vmt**

```
Bool(void * vmt)
```

Caches pre-built VMT for further use

**build_dynamic_vmt**

```
Bool( void * vmt, char * ancestorName, int ancestorVmtSize)
```

Creates a subclass from vmt and caches the result under ancestorName key

**gimme_the_vmt**

```
PVMT( const char *className);
```

Returns the VMT pointer associated with class by name.

**gimme_the_mate**

```
Handle( SV * perlObject)
```

Returns a C pointer to an object, if perlObject is a reference to a Prima object. returns NULL_HANDLE if the object stage is csDead

**gimme_the_real_mate**

```
Handle( SV * perlObject)
```

Returns a C pointer to an object, if perlObject is a reference to a Prima object. Same as `gimme_the_mate`, but does not check for the object stage.

**alloc1**

```
alloc1(type)
```

To be used instead (type*)(malloc(sizeof(type))

**allocn**

```
allocn(type,n)
```

To be used instead (type*)(malloc((n)*sizeof(type))

**alloc1z**

Same as `alloc1` but fills the allocated memory with zeros

**allocnz**

Same as `allocn` but fills the allocated memory with zeros

**prima_mallocz**

Same as malloc() but fills the allocated memory with zeros

**prima_hash_create**

```
PHash(void)
```

Creates an empty hash

**prima_hash_destroy**

```
void(PHash self, Bool killAll);
```

Destroys a hash. If killAll is true, assumes that every value in the hash is a dynamic memory pointer and calls free() on each.

**prima_hash_fetch**

```
void*( PHash self, const void *key, int keyLen);
```

Returns the pointer to a value, if found, NULL otherwise

**prima_hash_delete**

```
void*( PHash self, const void *key, int keyLen, Bool kill);
```

Deletes the hash key and returns the associated value. If the kill argument is true, calls free() on the value and returns NULL.

**prima_hash_store**

```
void( PHash self, const void *key, int keyLen, void *val);
```

Stores a new value into hash. If the key is already present, the old value is overwritten.

**prima_hash_count**

```
int(PHash self)
```

Returns the number of keys in the hash

**prima_hash_first_that**

```
void * ( PHash self, void *action, void *params, int *pKeyLen, void **pKey);
```

Enumerates all hash entries, calling action procedure on each. If the action procedure returns true, enumeration stops and the last processed value is returned. Otherwise NULL is returned. action has to be a function declared as

```
Bool action_callback( void * value, int keyLen, void * key, void * params);
```

The params argument is a pointer to arbitrary user data

**kind_of**

```
Bool( Handle object, void *cls);
```

Returns true, if the object is an exemplar of class cls or its descendant

**PERL_CALL_METHOD, PERL_CALL_PV**

To be used instead of perl_call_method and perl_call_pv, (see the *perlguts* manpage). These functions should be used to code a workaround of the perl bug which emerges when the G_EVAL flag is combined with G_SCALAR.

**eval**

```
SV *( char *string)
```

Simplified perl_eval_pv() call.

**sv_query_method**

```
CV * ( SV * object, char *methodName, Bool cacheIt);
```

Returns a perl pointer to a method searched by the perl object and the name. If cacheIt is true, caches the result of the hierarchy traversal for speedup.

**query_method**

```
CV * ( Handle object, char *methodName, Bool cacheIt);
```

Returns a perl pointer to a method searched by the C object and the name. If cacheIt is true, caches the hierarchy traverse result for a speedup.

**call_perl_indirect**

```
SV * ( Handle self, char *subName, const char *format, Bool cdecl,
       Bool coderef, va_list params);
```

The main core function for calling Prima methods. Is used by the following three functions, but is never called directly. The format is described in the `Calling perl code` section.

**call_perl**

```
SV * ( Handle self, char *subName, const char *format, ...);
```

Calls the subName method on a C object

**sv_call_perl**

```
SV * ( SV * mate, char *subName, const char *format, ...);
```

Calls the subName method on a perl object

**cv_call_perl**

```
SV * ( SV * mate, Sv * coderef, const char *format, ...);
```

Calls arbitrary perl code with a SV mate as the first parameter. Used in notifications mechanism.

**Object_create**

```
Handle( char * className, HV * profile);
```

Creates an exemplar of className class with parameters in the profile hash. Never returns NULL_HANDLE, throws an exception instead.

**create_object**

```
void*( const char *objClass, const char *format, ...);
```

A convenience wrapper to Object_create. Uses the format specification that is described in the section `Calling perl code` above.

**create_instance**

```
Handle( const char * className)
```

Convenience call to `Object_create` with parameters in hash 'profile'.

**Object_destroy**

```
void( Handle self);
```

Destroys an object. One of the few Prima functions that can be called in the re-entrant fashion.

**Object_alive**

```
void( Handle self);
```

Returns non-zero if the object is alive, 0 otherwise. In particular, returns 1 if the object's stage is csNormal and 2 if it is csConstructing. Has virtually no use in C, only used in perl code.

**protect_object**

```
void( Handle obj);
```

Protects the object data from deletion after Object_destroy() is called. Can be called several times on an object. Increments Object.protectCount .

**unprotect_object**

```
void( Handle obj);
```

Frees the objectdatapointer after Object.protectCount hits zero. Can be called several times on an object.

**parse_hv**

```
HV *( I32 ax, SV **sp, I32 items, SV **mark, int expected, const char *methodName);
```

Transfers arguments in perl stack to a newly created HV and returns it.

**push_hv**

```
void ( I32 ax, SV **sp, I32 items, SV **mark, int callerReturns, HV *hv);
```

Puts all hv contents back into perl stack.

**push_hv_for_REDEFINED**

```
SV **( SV **sp, HV *hv);
```

Puts hv content as arguments to perl code to be called

**pop_hv_for_REDEFINED**

```
int ( SV **sp, int count, HV *hv, int shouldBe);
```

Reads the result of the executed perl code and stores it into the HV hash.

**pexist**

```
Bool(char*key)
```

Returns true if a key is present in the hash 'profile'

**pdelete**

```
void(char*key)
```

Deletes a key in the hash 'profile'

**pget_sv, pget_i, pget_f, pget_c, pget_H, pget_B**

```
TYPE(char*key)
```

Returns a value of one of the types supported (SV*, int, float, char*, Handle or Bool) that is associated with a key in the hash 'profile'. Calls croak() if the key is not present.

**pset_sv, pset_i, pset_f, pset_c, pset_H**

```
void( char*key, TYPE value)
```

Assigns value to a key in hash 'profile' and increments reference count to a newly created scalar.

**pset_b**

```
void( char*key, void* data, int length)
```

Assigns binary data to a key in the hash 'profile' and increments the reference counter for the newly created scalar.

**pset_sv_noinc**

```
void(char* key, SV * sv)
```

Assigns a scalar value to a key in the hash 'profile' without incrementing the reference counter.

**duplicate_string**

```
char*( const char *)
```

Returns copy of a string

**list_create**

```
void ( PList self, int size, int delta);
```

Creates a list instance with a static List structure.

**plist_create**

```
PList( int size, int delta);
```

Created list instance and returns newly allocated List structure.

**list_destroy**

```
void( PList self);
```

Destroys the list data.

**plist_destroy**

```
void ( PList self);
```

Destroys the list data and frees the list instance.

**list_add**

```
int( PList self, Handle item);
```

Adds new item into a list, returns its index or -1 on error.

**list_insert_at**

```
int ( PList self, Handle item, int pos);
```

Inserts new item into a list at a given position; returns its position or -1 on error.

**list_at**

```
Handle ( PList self, int index);
```

Returns the item that is located at a given index or NULL_HANDLE if the index is out of range.

**list_delete**

```
void( PList self, Handle item);
```

Removes the item from the list.

**list_delete_at**

```
void( PList self, int index);
```

Removes the item located at a given index from a list.

**list_delete_all**

```
void ( PList self, Bool kill);
```

Removes all items from the list. If the kill argument is true, calls free() on every item before removing them from the list.

**list_first_that**

```
int( PList self, void * action, void * params);
```

Enumerates all list entries, calling action procedure on each. If the action returns true, the enumeration stops and the index is returned. Otherwise, -1 is returned. action has to be a function declared as

```
Bool action_callback( Handle item, void * params);
```

where params is a pointer to an arbitrary user data

**list_index_of**

```
int( PList self, Handle item);
```

Returns index of an item, or -1 if the item is not in the list.

## 10.2 Prima::codecs

How to write a codec for Prima image subsystem

### Description

How to write a codec for the Prima image subsystem

### Start simple

There are many graphical formats in the world, and yet more libraries, that depend on them. Writing a codec that supports a particular library is a tedious task, especially if one wants to support more than one format. Usually, you never want to get into internal parts, the functionality comes first, and who needs all those funky options that format provides? We want to load a file and to display its content. Everything else comes later - if ever. So, in a way to not scare you off, we start it simple.

#### Loading

Define a callback function like this:

```
static Bool
load( PImgCodec instance, PImgLoadFileInstance fi)
{
}
```

Just that function is not enough for the whole mechanism to work, but the bindings will come later. Let us imagine we work with an imaginary library libduff, and we want to load files of .duf format. *[ To discern imaginary code from real, imaginary will be prepended with _ - for example, _libduff_loadfile ].* So, we call the `_libduff_loadfile()` function, which loads black-and-white, 1-bits/pixel images, where 1 is white and 0 is black.

```
static Bool
load( PImgCodec instance, PImgLoadFileInstance fi)
{
   _LIBDUFF * _l = _libduff_load_file( fi-> fileName);
   if ( !_l) return false;

   // - create storage for our file
   CImage( fi-> object)-> create_empty( fi-> object,
     _l-> width, _l-> height, imBW);

   // Prima wants images aligned to a 4-byte boundary,
   // happily libduff has the same considerations
   memcpy( PImage( fi-> object)-> data, _l-> bits,
     PImage( fi-> object)-> dataSize);

   _libduff_close_file( _l);

   return true;
}
```

Prima keeps an open handle of the file; if libduff can use file handles, then we can use it too, which is more robust than just file names because the caller can also load images from a byte stream.

```
{
  _LIBDUFF * _l = _libduff_load_file_from_handle( fi-> f);
   ...
// In both cases, you don't need to close the handle -
// however you might, it is ok:

   _libduff_close_file( _l);
   fclose( fi-> f);
// You just assign it to NULL to indicate that you've closed it
   fi-> f = NULL;
   ...
}
```

Together with `load()` you will need to implement minimal `open_load()` and `close_load()` functions.

The simplest `open_load()` returns a non-null pointer as a success flag:

```
static void *
open_load( PImgCodec instance, PImgLoadFileInstance fi)
{
   ... open file handle ...
   return (void*)1;
}
```

Its result will be stored in `PImgLoadFileInstance-> instance` for future reference. If it was dynamically allocated, free it in `close_load()`. A dummy `close_load()` is doing nothing but must be present nevertheless:

```
static void
close_load( PImgCodec instance, PImgLoadFileInstance fi)
{
}
```

### Writing to `PImage-> data`

Prima formats its image data as 32-bit aligned scanlines in a contiguous memory block. If *libduff* allows reading from files by scanlines, we can use the `lineSize` field to properly address the data:

```
PImage i = ( PImage) fi-> object;
// note - since this notation is more convenient than
// PImage( fi-> object)-> , instead i-> will be used

Byte * dest = i-> data + ( _l-> height - 1) * i-> lineSize;
while ( _l-> height--) {
   _libduff_read_next_scanline( _l, dest);
   dest -= i-> lineSize;
}
```

Note that the image is filled in reverse - Prima images are built like a classical XY-coordinate grid, where Y ascends upwards.

Here ends the simple part. You can skip down to the the *Registering with the image subsystem* entry part if you want it fast.

## Single-frame loading

### Palette

Our *libduff* images can be black-and-white in two ways - where 0 is black and 1 is white and vice versa. While 0B/1W perfectly corresponds to the `imbpp1 | imGrayScale` Prima image type and no palette operations are needed ( Prima cares automatically about these), a 0W/1B is a black-and-white grayscale image that should be treated like the imbpp1 type with custome palette:

```
if ( l-> _reversed_BW) {
   i-> palette[0].r = i-> palette[0].g = i-> palette[0].b = 0xff;
   i-> palette[1].r = i-> palette[1].g = i-> palette[1].b = 0;
}
```

Note. The image always has a palette array with a size enough to store 256 colors, since it can't know beforehand the actual palette size. If the color palette for, say, a 4-bit image contains 15 out of the 16 colors possible, the code like

```
i-> palSize = 15;
```

does the trick.

### Data conversion

Prima defines image scanline size to be aligned to 32 bits, and the formula for the calculation of the scanline size is

```
lineSize = (( width * bits_per_pixel + 31) / 32) * 4;
```

Prima defines many converting routines between different data formats. Some of them can be applied to scanlines, and some to the whole image ( because sampling algorithms generally may need access to more than a single scanline). These are defined in *include/img_conv.h*, and probably the ones that you'll need would be `bc_format1_format2`, which works on scanlines, and also `ibc_repad` that does byte repadding.

For those who are especially lucky, some libraries do not check between machine byte format and file byte format. Prima unfortunately doesn't provide an easy method for determining this situation, but you'll have to convert your data in the appropriate way to have picture data displayed correctly. Note the `BYTEORDER` symbol that is ( usually ) defined in *sys/types.h*.

### Loading with no data

If a high-level code just needs information about the image dimensions and bit depth rather than its pixels, a codec should be able to provide that in an effective way. The implementation above would still work but will use more memory and time. The `PImgLoadFileInstance-> noImageData` flag indicates if image data is needed. On that condition, the codec needs to report only the dimensions of the image - but the type must be set anyway. Here is the full code:

```
static Bool
load( PImgCodec instance, PImgLoadFileInstance fi)
{
   _LIBDUFF * _l = _libduff_load_file( fi-> fileName);
   HV * profile = fi-> frameProperties;
   PImage i = ( PImage) fi-> frameProperties;
   if ( !_l) return false;

   CImage( fi-> object)-> create_empty( fi-> object, 1, 1,
      _l-> _reversed_BW ? imbpp1 : imBW);
```

```
   // copy palette, if any
   if ( _l-> _reversed_BW) {
      i-> palette[0].r = i-> palette[0].g = i-> palette[0].b = 0xff;
      i-> palette[1].r = i-> palette[1].g = i-> palette[1].b = 0;
   }

   if ( fi-> noImageData) {
      // report dimensions
      pset_i( width,  _l-> width);
      pset_i( height, _l-> height);
      return true;
   }

   // - create storage for our file
   CImage( fi-> object)-> create_empty( fi-> object,
        _l-> width, _l-> height,
        _l-> _reversed_BW ? imbpp1 : imBW);

   // Prima wants images aligned to a 4-byte boundary,
   // happily libduff has the same considerations
   memcpy( PImage( fi-> object)-> data, _l-> bits,
     PImage( fi-> object)-> dataSize);

   _libduff_close_file( _l);

   return true;
}
```

The newly introduced macro `pset_i` is a convenience operator, assigning integer (i) as a value to a hash key, given as a first parameter - it becomes a string literal upon the expansion. The hash used for storage is a perl scalar of type `HV*`. The following code

```
HV * profile = fi-> frameProperties;
pset_i( width, _l-> width);
```

is a syntax sugar for

```
hv_store(
    fi-> frameProperties,
    "width", strlen( "width"),
    newSViv( _l-> width),
    0);
```

hv_store(), which together with HV's and SV's and the other symbols are described in *perlguts*.

### Returning extra information

The most useful image attributes are dimensions, type, palette, and (pixel) data. However different formats can supply a fair amount of other image information, often irrelevant but sometimes useful. In the perl code, an image has access have a special hash reference `'extras'` on object, where all this information is stored. Codec can report also such data, storing it in `PImgLoadFileInstance->frameProperties`. Data should be stored in the native perl format, so if you're not familiar with perl scalar implementation, you might want to read it first (see *perlguts*), especially if you want to return arrays and hashes. But for simple types, one can return the following perl scalars:

**integers**

```
pset_i( integer, _l-E<gt> integer);
```

**floats**

```
pset_f( float, _l-E<gt> float);
```

**strings**

```
pset_c( string, _l-E<gt> charstar);
```

- note - no `malloc` call is required

**prima objects**

```
pset_H( Handle, _l-E<gt> primaHandle);
```

**SVs**

```
pset_sv_noinc( scalar, newSVsv(sv));
```

**hashes**

```
pset_sv_noinc( scalar, ( SV *) newHV());
```

hashes created through `newHV` can be filled in the same manner as described here

**arrays**

```
pset_sv_noinc( scalar, ( SV *) newAV());
```

arrays (AVs) are described in perlguts also, but the most useful function here is `av_push`. To push 4 values, for example, this code:

```
AV * av = newAV();
for ( i = 0;i < 4;i++) av_push( av, newSViv( i));
pset_sv_noinc( myarray, newRV_noinc(( SV *) av);
```

is a C equivalent to

```
->{extras}-> {myarray} = [0,1,2,3];
```

High-level code can specify if the extra information should be loaded. This behavior is determined by the flag `PImgLoadFileInstance-> loadExtras`. A codec may choose to not respect this flag, and thus the image extra information will not be returned. All data that can be possibly extracted from an image, should be listen in the <`char ** PImgCodecInfo-` loadOutput>> array:

```
static char * loadOutput[] = {
   "hotSpotX",
   "hotSpotY",
   NULL
};

static ImgCodecInfo codec_info = {
   ...
   loadOutput
};
```

```
static void *
init( PImgCodecInfo * info, void * param)
{
   *info = &codec_info;
   ...
}
```

The code above is taken from codec_X11.c, where the X11 bitmap can provide the location of the hotspot, as two integers, X and Y. The type of the data is not specified.

### Loading to icons

If high-level code wants an Icon object with 1-bit mask (*and*-mask) instead of an Image object, Prima can take care of producing the mask automatically. However, if codec can read the explicit transparency data, it might instead change the final mask in a more precise way. The mask pixels are stored on the Icon obejct in the `mask` field.

a) Let us imagine, that a 4-bit image always carries a transparent color index, in the 0-15 range. In this case, the following code will create the correct mask:

```
if ( kind_of( fi-> object, CIcon) &&
     ( _l-> transparent >= 0) &&
     ( _l-> transparent < PIcon( fi-> object)-> palSize)) {
   PRGBColor p = PIcon( fi-> object)-> palette;
   p += _l-> transparent;
   PIcon( fi-> object)-> maskColor = ARGB( p->r, p-> g, p-> b);
   PIcon( fi-> object)-> autoMasking = amMaskColor;
}
```

Of course,

```
pset_i( transparentColorIndex, _l-> transparent);
```

would be also helpful to report.

b) if an explicit bit mask is contained in the image, the code will be using the `amNone` constant instead:

```
if ( kind_of( fi-> object, CIcon) &&
     ( _l-> maskData >= 0)) {
   memcpy( PIcon( fi-> object)-> mask, _l-> maskData, _l-> maskSize);
   PIcon( fi-> object)-> autoMasking = amNone;
}
```

Note that the mask is also subject to LSB/MSB and 32-bit alignment issues. Treat it as a regular imbpp1 data format.

c) A format supports transparency information, but the image does not contain any. In this case no, action is required on the codec's part; the high-level code specifies if the transparency mask is created ( iconUnmask field ).

d) The full alpha transparency, if present, can be loaded into a 8-bit alpha mask. The icon mask storage should be upgraded to accomodate for the 8-bit mask pixel depth by calling either `mask` or `create_empty_icon` methods.

### open_load() and close_load()

`open_load()` and `close_load()` are used as brackets for load requests. If a codec assigns `false` to `PImgCodecInfo-> canLoadMultiple` that means that it can only load a single image object from an image file, even if the image format supports many images per file. It may report

the total amount of frames, but still be incapable of loading them. There is also a load sequence, called null-load, when no load() calls are made, just `open_load()` and `close_load()`. These requests are made in case the codec can provide some file information without loading frames at all. It can be any information, of whatever kind. It has to be stored in the hash `PImgLoadFileInstance-> fileProperties`, to be filled once on `open_load()`. The only exception is `PImgLoadFileInstance-> frameCount`, which can be updated during one of `load()` calls. Actually, the `frameCount` field could be filled during any load stage, except `close_load()`, so that the Prima code that drives the multiframe logic would be able to correctly track individual images.

Even if the codec can only load single image per file, it is still advised to fill this field, at least to tell whether a file is empty (`frameCount == 0`) or not ( `frameCount == 1`). More information about the `frameCount` field can be found below in the chapters dedicated to the multiframe requests.

### Load input

So far a codec is expected to respond for the `noImageData` hint only, and it is possible to allow a high-level code to alter the codec load behavior, passing specific parameters. `PImgLoadFileInstance-> profile` is a hash, that contains these parameters. The data that should be applied to all frames and/or the whole image file are set there when open_load() is called. These data, plus frame-specific keys passed to every load() call. However, Prima passes only those hash keys, which are returned by the `load_defaults()` function. This function returns a newly created ( by calling `newHV()`) hash, with the accepted keys and their default ( and always valid ) value pairs. The example below defines the `speed_vs_memory` field, which should accept integer values 0, 1, or 2.

```
static HV *
load_defaults( PImgCodec c)
{
   HV * profile = newHV();
   pset_i( speed_vs_memory, 1);
   return profile;
}
...
static Bool
load( PImgCodec instance, PImgLoadFileInstance fi)
{
     ...
     HV * profile = fi-> profile;
     if ( pexist( speed_vs_memory)) {
        int speed_vs_memory = pget_i( speed_vs_memory);
        if ( speed_vs_memory < 0 || speed_vs_memory > 2) {
             strcpy( fi-> errbuf, "speed_vs_memory should be 0, 1 or 2");
             return false;
        }
        _libduff_set_load_optimization( speed_vs_memory);
     }
}
```

The latter code chunk can be applied to open_load() as well.

### Returning an error

The image subsystem defines no severity gradation for codec errors. If an error occurs during loading, the codec returns a false value, which is `NULL` on `open_load()` and `false` on `load()`. It is advisable to explain the error, otherwise, the user gets just the generic "Load error" string. To

do so, the error message is to be copied to `PImgLoadFileInstance-> errbuf :: char[256]` . On an extremely severe error codec may call croak(), which jumps to the closest G_EVAL block. If there are no G_EVAL blocks then the program aborts. This condition could also happen if the codec calls some Prima code that issues croak(). This condition is untrappable, - at least without calling perl functions. Understanding that that behavior is not acceptable, it is still under design.

## Multiple-frame load

To indicate that the codec is ready to read multiframe images, it must set the `PImgCodecInfo-> canLoadMultiple` flag to true. This only means, that codec should respond to the `PImgLoadFileInstance-> frame` field, which value is an integer that should be in the range from `0` to `PImgLoadFileInstance-> frameCount - 1`. It is advised that the codec should change the frameCount from its original value `-1` to the actual one, to help Prima filter range requests before they go down to the codec. The only real problem that may happen to the codec which is unwilling to initialize `frameCount`, is as follows. If a loadAll request was made ( corresponding boolean `PImgLoadFileInstance-> loadAll` flag is set for codec's information) and frameCount is not initialized, then Prima starts loading all frames, incrementing frame index until it receives an error. Assuming the first error it gets is an EOF, it reports no error, so there's no way for a high-level code to tell whether there was a loading error or an end-of-file condition. Codec may initialize the `frameCount` field at any time during `open_load()` or `load()`, even while returning a false return value to the caller.

## Saving

The approach for handling saving requests is very similar to the handling of the loading requests. For the same reason and with the same restrictions functions `save_defaults()`, `open_save()`, `save()`, and `close_save()` are defined. Below is an example of typical saving code with highlighted differences from the loading code. As an example, we'll take existing *img/codec_X11.c*, which defines extra hotspot coordinates, X and Y.

```
static HV *
save_defaults( PImgCodec c)
{
   HV * profile = newHV();
   pset_i( hotSpotX, 0);
   pset_i( hotSpotY, 0);
   return profile;
}

static void *
open_save( PImgCodec instance, PImgSaveFileInstance fi)
{
   return (void*)1;
}

static Bool
save( PImgCodec instance, PImgSaveFileInstance fi)
{
   PImage i = ( PImage) fi-> object;
   Byte * l;
   ...

   fprintf( fi-> f, "#define %s_width %d\n", name, i-> w);
   fprintf( fi-> f, "#define %s_height %d\n", name, i-> h);
   if ( pexist( hotSpotX))
```

```
      fprintf( fi-> f, "#define %s_x_hot %d\n", name, (int)pget_i( hotSpotX));
   if ( pexist( hotSpotY))
      fprintf( fi-> f, "#define %s_y_hot %d\n", name, (int)pget_i( hotSpotY));
   fprintf( fi-> f, "static char %s_bits[] = {\n  ", name);
   ...
   // printing of data bytes is omitted
}

static void
close_save( PImgCodec instance, PImgSaveFileInstance fi)
{
}
```

A saving request takes into account the image types that the codec previously declared to support, and that are defined in the `PImgCodecInfo-> saveTypes` array. Prima converts an image to be saved into one of these formats, before the actual `save()` call takes place.

A codec may also set two of `PImgCodecInfo` flags, `canSave` and `canSaveMultiple`. Save requests will never be called if `canSave` is false, and correspondingly, the multiframe save requests would be never invoked for a codec with `canSaveMultiple` set to false. The scenario for a multiframe save request is the same as for a multiframe loading request. All the issues concerning palette, data converting, and saving extra information are actual, however, there's no corresponding flag like `loadExtras` - the codec is expected to save all information that it can extract from the `PImgSaveFileInstance-> objectExtras` hash.

## Registering with the image subsystem

Finally, the codec has to be registered. All of its callback functions are to be set into a `ImgCodecVMT` structure. The function slots that are unused should not be defined as dummies - those are already defined and gathered under the `CNullImgCodecVMT` struct. That's why all functions in the illustration code were defined as static. A codec has to provide some information that Prima uses to decide which codec should load a particular file type. If no explicit directions are given, Prima would only ask the codecs that match with the loaded file's extensions. The `init()` function should return a pointer to the filled struct, that describes the codec's capabilities:

```
// extensions to file - might be several, of course, thanks to dos...
static char * myext[] = { "duf", "duff", NULL };

// we can work only with 1-bit/pixel
static int    mybpp[] = {
    imbpp1 | imGrayScale, // 1st item is a default type
    imbpp1,
    0 };   // Zero means end-of-list. No type has zero value.

// main structure
static ImgCodecInfo codec_info = {
   "DUFF", // codec name
   "Numb & Number, Inc.", // vendor
   _LIBDUFF_VERS_MAJ, _LIBDUFF_VERS_MIN,    // version
   myext,    // extension
   "DUmb Format",     // file type
   "DUFF",     // file short type
   NULL,    // features
   "",     // module
   true,   // canLoad
   false,  // canLoadMultiple
```

```
   false,  // canSave
   false,  // canSaveMultiple
   mybpp,  // save types
   NULL,    // load output
};

static void *
init( PImgCodecInfo * info, void * param)
{
   *info = &codec_info;
   return (void*)1; // just non-null, to indicate success
}
```

The result of init() is stored in the `PImgCodec-> instance`, and the information in the `PImgCodec-> info` field. If dynamic memory was allocated for these structs, it can be freed on `done()` invocation which happens at the end of the program run. Finally, the function that is invoked from Prima, is the only one that is required to be exported, and is responsible for registering a codec:

```
void
apc_img_codec_duff( void )
{
   struct ImgCodecVMT vmt;
   memcpy( &vmt, &CNullImgCodecVMT, sizeof( CNullImgCodecVMT));
   vmt. init          = init;
   vmt. open_load     = open_load;
   vmt. load          = load;
   vmt. close_load    = close_load;
   apc_img_register( &vmt, NULL);
}
```

This procedure can register as many codecs as it wants to, but currently, Prima is designed so that one codec_XX.c file should be connected to one library only.

The name of the procedure is *apc_img_codec_* plus the library name, which is required for a compilation with Prima. The file with the codec should be called `codec_duff.c` (in our case) and put into the `img` directory in the Prima source tree. Following these rules, Prima will be assembled with `libduff.a` ( or `duff.lib`, or some other file, as the actual library name is system dependent) if the library is present.

## 10.3 prima-gencls

Class interface compiler for Prima core modules

### Synopsis

```
prima-gencls --h --inc --tml -O -I<name> --depend --sayparent filename.cls
```

### Description

Creates C files for Prima core module object definitions.

### Arguments

The prima-gencls program accepts the following arguments:

**--h**

Generates a .h file ( with the declarations to be included in one or more files )

**--inc**

Generates a .inc file ( with the declarations to be included in only one file )

**-O**

Turns on the optimizing algorithm for the .inc files. The algorithm is based on an assumption that some function bodies are identical, and so the duplicates can be detected and removed. When the `-O` flag is set, the body of such a function is replaced with a call to the function with an auto-generated name. That function is not included in the .inc file, but in a .tml file instead. All the duplicate declarations from a set of .tml files can be later removed by the the *tmlink* entry utility.

**--tml**

Generates a .tml file. Turns the `-O` flag on automatically.

**-Idirname**

Adds a directory to a search path, where the program searches for the .cls files. Can be specified several times.

**--depend**

Prints out dependencies for the given file.

**--sayparent**

Prints out the immediate parent of a class inside the given file.

### Syntax

The syntax of a .cls file can be described by the following scheme:

```
[ zero or more type declarations ]
[ zero or one class declaration ]
```

The prima-gencls program produces .h, .inc, and .tml files with the same basename as the .cls file if no object or package name is given, or with the name of the object or the package inside the .cls file otherwise.

### Basic scalar data types

Prima-gencls can generate the conversion code for several built-in scalar data types that transfers data between C and perl, using the XS ( see *perlguts* ) library interface.

These types are:

```
int
Bool
Handle
double
SV*
HV*
char *
string ( C declaration is char[256] )
```

There are also some derived built-in types, which are

```
long
short
char
Color
U8
```

that are mapped to the `int` type. The data undergoes no conversion to int in the transfer process, but it is stored to the perl scalar using the newSViv() function which may lose bits or a sign.

### Derived data types

The syntax for a new data type definition is as follows:

```
<scope> <prefix> <id> <definition>
```

A scope can be one of two pragmas, `global` or `local`. That requests the usage locality of the new data type, i e whether the type will be used only for one class or in more than one. Usage of the `local` scope somewhat resembles the C predicate `static`. The difference between the scopes is that a function using a complex local type in the parameter list, or as a result, will not be optimized out with the `-O` flag.

### Scalar types

New scalar types may only be aliased to the existing ones, primarily for coding convenience in C. A scalar type can be defined in two ways:

### Direct aliasing

Syntax:

```
<scope> $id => <basic_scalar_type>;
```

Example:

```
global $Handle => int;
```

The new type id will not be visible in the C files, but the type will be substituted over all .cls files that include this definition.

### C macro

Syntax:

```
<scope> id1 id2
```

Example:

```
global API_HANDLE UV
```

Such code creates a C macro definition in the .h header file in the form

```
#define id1 id2
```

C macros with parameters are not allowed. id1 and id2 are not required to be present in the .cls namespace, and no substitution during .cls file processing is made.

### Complex types

Complex data types can be arrays, structs, and hashes. Prima-gencls allows several combinations of complex data types that C language does not recognize. These will be described below.

The complex data types do not get imported into the perl code. A perl program must conform to the data type used when passing the corresponding parameters to such a function.

### Arrays

Syntax:

```
<scope> @id <basic_scalar_type>[dimension];
```

Example:

```
global @FillPattern U8[8];
```

Example of functions using arrays:

```
Array * func( Array a1, Array * a2);
```

Perl code:

```
@ret = func( @array1, @array2);
```

Note that the array references are not used, and the number of items in the array parameters must be exactly the same as the dimensions of the arrays.

Warning: the following declaration will not compile with the C compiler, as C cannot return arrays. However, this construct is not treated as an error by prima-gencls:

```
Array func();
```

### Structs

Syntax:

```
<scope> @id {
   <basic_scalar_type> <id>;
   ...
   <basic_scalar_type> <id>;
};
```

Example:

```
global @Struc {
   int    number;
   string id;
}
```

Example of the functions using structs:

```
Struc * func1( Struc a1, Struc * a2);
Struc   func2( Struc a1, Struc * a2);
```

Perl code:

```
@ret = func1( @struc1, @struc2);
@ret = func2( @struc1, @struc2);
```

Note that the array references are not used, and both the number and order of items in the array parameters must be exactly as the dimensions and the order of the structs. The struct field names are not used in the perl code as well.

**Hashes**

Syntax:

```
<scope> %id {
   <basic_scalar_type> <id> [with undef];
   ...
   <basic_scalar_type> <id> [with undef];
};
```

Example:

```
global %Hash {
   int    number;
   string id with undef;
}
```

Examples of the functions using hashes:

```
Hash * func1( Hash a1, Hash * a2);
Hash   func2( Hash a1, Hash * a2);
```

Perl code:

```
%ret = %{func1( \%hash1, \%hash2)};
%ret = %{func2( \%hash1, \%hash2)};
```

Note that only the hash references are used and returned. When a hash is passed from the perl code it might have some or all fields unset. The C structure is filled and passed to a C function, and the unset fields are assigned to a corresponding C_TYPE_UNDEF value, where TYPE is one of NUMERIC, STRING, and POINTER literals.

If the optional `with undef` declarator was used, the C structure is augmented with additional struct `undef` with boolean fields that explicitly reflect whether the perl value was passed or not. The C-to-perl conversion respects these boolean flags as well and does not populate hash fields with this bit set.

### Namespace section

Syntax:

```
<namespace> <ID> {
   <declaration>
   ...
   <declaration>
}
```

A .cls file can have zero or one namespace sections filled with function descriptions. Functions described here will be exported to the given ID in the initialization code. A namespace can be either `object` or `package`.

The package namespace syntax only allows functions without the prefix inside the `package` block:

```
package <Package ID> {
    <function description>
    ...
}
```

The object namespace syntax allows variables and properties as functions ( called methods in the object syntax ). The general object namespace syntax is

```
object <Class ID> [(Parent class ID)] {
   <variables>
   <methods>
   <properties>
}
```

Within the object namespace the inheritance syntax can be also used:

```
object <Class ID> (<Parent class ID>)  { ... }
```

### Functions

Syntax:

```
[<prefix>] <type> <function_name> (<parameter list>) [ => <alias>];
```

Examples:

```
package A {
   int   func1( int a, int b);
   Point func2( Struc * x);
}
```

```
object B
   method int   func1( int a, int b = 1) => c_func_2;
   import Point func2( Struc * x, ...);
   c_only void  func3( HV * profile);
}
```

The prefix is used with object functions ( methods ) only. More on the prefix in the the *Methods* entry section.

A function can return nothing ( void ), a scalar ( int, string, etc ), or a complex ( array, hash ) type. It can as well accept scalar and complex parameters, with the type conversion that corresponds to the rules described above in the the *Basic scalar data types* entry section.

If a function has parameters and/or a result of the type that cannot be converted automatically between C and perl, it gets declared but not exposed to the perl namespace. A warning is also issued. It is not possible to use the gencls syntax to declare a function with custom parameters or result types. For such purpose, the explicit C declaration of the code along with a call to `newXS` must be made.

Example: ellipsis (...) cannot be converted by prima-gencls, even though it is a legal C construction.

```
Point package_func2( Struc * x, ...);
```

The function syntax has several convenience additions:

**Default parameter values**

Example:

```
void func( int a = 15);
```

A function declared in such a way can be called both with 0 or 1 parameters. If it is called with 0 parameters, an integer value of 15 will be automatically used. The syntax allows default parameters for types int, pointer, and string, and their scalar aliases.

The default parameters can be as many as possible, but they have to be at the end of the function parameter list. The declaration `func( int a = 1, int b)` is incorrect.

**Aliasing**

In the generated C code, a C function has to be called after the parameters have been parsed. Prima-gencls expects a conforming function to be present in the C code, with the fixed name and parameter list. However, if the only task of such a function is to be a one-to-one wrapper to the identical function published under another name, aliasing can be performed to save both code and speed:

Example:

```
package Package {
   void func( int x) => internal;
}
```

A function declared in that way will not call the Package_func() C function, but the internal() function instead. The only request here is that the internal() function must have the same C parameters and result as the func() function.

**Inline hash**

If a function is declared with the last parameter of the `HV*` type then the parameter translation from perl to C is performed as if all the parameters passed are a hash. This hash is passed to the C function and its content is returned then back to perl as a hash again. The hash content can be modified inside the C function.

This declaration is used heavily in constructors, that are coded in perl typically like this:

```
sub init
{
   my %ret = shift-> SUPER::init( @_);
   ...
   return %ret;
}
```

and the corresponding C code is

```
void Obj_init ( HV * profile) {
    inherited init( profile);
    ... [ modify profile content ] ...
}
```

### Methods

Methods are the functions called in the context of an object. Virtually all class methods need to have access to the object they are bound to. Prima objects are visible in C as the `Handle` data type. The Handle is a pointer to the object instance which in turn contains a pointer to the object virtual methods table ( VMT ). To facilitate the OO-like syntax, this Handle parameter is rarely mentioned in the method declarations:

```
method void a( int x)
```

however, the signature of the corresponding C function contains the Handle parameter

```
void Object_a( Handle self, int x)
```

Methods are accessible in C code by the direct name dereferencing of the `Handle self` like this:

```
((( PMyObject) self)-> self)-> my_method( self, ...);
```

A method can have one of the following six prefixes that produce different C code wrappers:

**method**

This is the most basic method type. Methods of this type are expected to be coded in C, the object handle is implicit and is not included in the .cls function declaration:

```
method void a()
```

results in

```
void Object_a( Handle self)
```

C declaration. The published method automatically converts its parameters and the result between C and perl.

**public**

When the methods need to have parameters and/or a result that cannot be automatically converted between C and perl, or the function declaration does not fit into the C syntax, the `public` prefix is used. The methods declared as `public` are expected to communicate with perl by through the XS ( see *perlxs*) interface. It is also expected that the programmer declares the REDEFINED and FROMPERL functions in C code( see the *Prima::internals* section for details). Examples are many throughout the Prima source and will not be shown here. `public` methods usually have the void result and no parameters, but that does not matter since prima-gencls provides no data conversion for such methods anyway.

**import**

For the methods that are best implemented in perl instead of C, prima-gencls can produce the C-to-perl wrappers using the `import` prefix. An `import` function does not need a C counterpart, only the auto-generated code.

**static**

If the method has to be able to work both with and without an object instance, it needs to be prepended with the `static` prefix. `static` methods are similar to `method` except that the `Handle self` first parameter is not implicitly declared. If the `static` method is called without an object ( but with a class ), like for example

```
Class::Object-> static_method();
```

its first parameter is not an object but the "Class::Object" string. If the method never uses that first parameter it is enough to declare it as

```
static a( char * className = "");
```

but if it does, a

```
static a( SV * class_or_object = NULL);
```

declaration is needed. In the latter case C code itself has to determine what exactly has been passed, if ever. Note the default parameter here: a `static` method is usually legible to call as

```
Class::Object::static_method();
```

where no parameters are passed to it. Without the default parameter, such a call generates an 'insufficient parameters passed' runtime error.

**weird**

We couldn't find a better name for it. The `weird` prefix describes a method that combines properties from both the `static` and `public` methods. Prima-gencls generates no conversion code for the `weird` methods and expects no `Handle self` as the first parameter. As an example, the Prima::Image::load function can be called using a wide spectrum of calling semantics ( see the *Prima::image-load* section for details).

**c_only**

The `c_only` methods are present in the VMT but are not accessible from perl. They can be overloaded, but from C only. Moreover, it is allowed to register a perl function with the same name as the existing `c_only` method, and these entities will be completely independent.

Note: methods that have a result and/or parameters declared as data types that cannot be converted automatically, change their prefix to `c_only` during the .cls processing. Probably this is the wrong behavior, and such a condition has to signal an error.

### Properties

The Prima toolkit introduces an entity named property, that is used to replace method pairs whose function is to acquire and assign some internal object variable, for example, an object name, color, etc. Instead of having a pair of methods like Object::set_color and Object::get_color, a property Object::color can be used instead. A property is a method with special considerations, in particular, when it is called without parameters, the get-mode is implied. On the contrary, if it is called with one parameter, such a call is treated as being done in the set-mode. Note that in both 'set' and 'get' invocations the first parameter `Handle self` is implicit and is always present.

Properties can operate with different, but always fixed number of parameters. For example,

```
property char * name
```

has the C counterpart

```
char * Object_name( Handle self, Bool set, char * name)
```

Depending on the calling mode, the `Bool set` argument is either `true` or `false`. In the set-mode, the C code result is discarded, while in the get-mode the property parameter value is undefined.

The syntax for a multi-parameter property is

```
property long pixel( int x, int y);
```

with the C code

```
long Object_pixel( Handle self, Bool set, int x, int y, long pixel)
```

Note that in the multi-parameter case, the parameters declared after the property name are always initialized, in both the set- and get- modes.

### Instance variables

The prima-gencls syntax allows variable declarations for the variables that are allocated for every object instance. Although data type validation is not performed for variables, and their declarations just get copied as is, complex C declarations involving array, struct, and function pointers are not supported. As a workaround, pointers to typedef'd entities are used. Example:

```
object SampleObject {
   int x;
   List list;
   struct { int x } s; # illegal declaration
}
```

The variables are accessible in C code by direct name dereferencing the `Handle self`:

```
 (( PMyObject) self)-> x;
```

# 11 Miscellaneous

## 11.1 Prima::faq

Frequently asked questions about Prima

### Description

The FAQ covers various topics around Prima, such as distribution, compilation, installation, and programming.

### COMMON

#### What is Prima?

Prima is a general-purpose extensible graphical user interface toolkit with a rich set of standard widgets and an emphasis on 2D image processing tasks. A Perl program using Prima looks and behaves identically in the X11 and Windows environments.

#### Yeah, right. So what is Prima again?

A Yet Another Perl GUI.

#### Why bother with the Yet Another thing, while there is Perl-Tk and plenty of others?

Prima was started on OS/2, where Tk didn't run. We have had two options - either port Tk, or write something on our own, probably better than the existing tools. We believe that we've succeeded.

However, Prima's support for OS/2 was removed because no one needed that in 2012.

#### Why Perl?

Why not? Perl is great. The high-level GUI logic fits badly into C, C++, or the like, so a scripting language is probably the way to go here.

#### But I want to use Prima in another language.

Unless your language has runtime binding with perl, you cannot.

#### Who wrote Prima?

Dmitry Karasik implemented the majority of the toolkit, after the original idea by Anton Berezin. Many contributors helped the development of the toolkit since then.

### What is the copyright?

The copyright is a modified BSD license, where only two first paragraphs remain out of the original four. The text of copyright is present in almost all files of the toolkit.

### I'd like to contribute

You can do this in several ways. The project would probably best benefit from the advocacy because not many people use it. Of course, you can send in new widgets, patches, suggestions, or even donations. Also, documentation is the topic that needs particular attention, since my native language is not English, so if there are volunteers for polishing the Prima docs, you are very welcome.

## INSTALLATION

### Where can I download Prima?

the *http:www.prima.eu.org* entry contains links to source and binary download resources, and some other useful info.

### What is better, source or binary?

Depends on where you are and what are your goals. On unix, the best is to use the source. On win32 the binaries based on Strawberry Perl distribution are preferred. If you happen to use cygwin you are probably still better off using the source.

### How to install the binary distribution?

First, check if you downloaded the Prima binary for the correct version of Perl, that should be enough.

To install, unpack the archive and type 'perl ms_install.pl'. The files will be copied into the perl tree.

### How to compile Prima from the source?

Type the following:

```
perl Makefile.PL
make
make install
```

If the 'perl Makefile.PL' fails with errors, you can check makefile.log to see if anything is wrong. A typical situation here is that Makefile.PL might report that it cannot find the Perl library, for example, where the real problem is that it invokes the compiler in the wrong way.

Note, that to get Prima working from sources, your system must contain graphic libraries, such as libgif or ligjpeg, for Prima to load graphic files.

### What about the graphic libraries?

To load and save images, Prima uses graphic libraries. Such as, to load GIF files, the libgif library is used, etc. Makefile.PL finds available libraries and links Prima against these. It is possible to compile Prima without any, but this is not useful.

On every supported platform Prima can make use of the following graphic libraries:

```
libXpm   - Xpm pixmaps
libjpeg  - JPEG images
libgif   - GIF images
libpng   - PNG images
libtiff  - tiff images
libwebp,libwebpdemux,libwebpmux  - WebP images
libheif  - Heif images
```

Strawberry perl and Cygwin come with most of them, so on these installations Prima just compiles without any trouble. For other perl builds, use one of the `Prima::codecs::` modules that contain the needed include and lib files. If you are installing Prima with CPAN, that gets done automatically.

### img/codec_XXX.c compile error

`img/codec_XXX.c` files are C sources for support of the graphic libraries. In case a particular codec does not compile, the ultimate fix is to remove the file and re-run Makefile.PL . This way the problem can be avoided easily, although at the cost of a lack of support for that graphic format.

### How do I check what graphic libraries are supported?

```
perl -MPrima::noX11 -MPrima -e 'print map { $_->{name}.qq(\n) } @{Prima::Image->codecs};'
```

### I have a graphic library installed, but Makefile.PL doesn't find it

The library is probably located in such a location Makefile.PL must be told about by adding LIBPATH+=/mypath/lib, and possibly INCPATH+=/mypath/include in the command line. Check makefile.log created by Makefile.PL for the actual errors reported when it tries to use the library.

### Compile errors

There are various reasons why a compilation may fail. The best would be to copy the output together with outputs of env and perl -V and send these to the author, or better, open a GitHub issue here the *https:github.comdkPrimaissues* entry.

### Prima doesn't run

Again, there are reasons for Prima to fail.

First, check whether all main files are installed correctly. *Prima.pm* must be in your perl directory, and the Prima library file ( *Prima.a* or *Prima.so* for unix, *Prima.dll* for win32 ) is copied in the correct location in the perl tree.

Second, try to run 'perl -MPrima -e 1' . If Prima.pm is not found, the error message would be something like

```
Can't locate Prima.pm in @INC
```

If the Prima library or one of the libraries it depends on cannot be found, perl Dynaloader would complain. On win32 this usually happens when some dll files Prima needs are not found. If this is the case, try to copy these files into your PATH, for example in C:/Windows .

### Cannot install Prima on ActiveState

ActiveState doesn't seem to support anymore compilation of locally built libraries and doesn't provide precompiled Prima distributions either. Consider using Strawberry or msys2 builds instead.

### Prima error: Can't open display

This error happens when you are running under the X11 environment and no connection to the X11 display can be established. Check your DISPLAY environment variable, or use the --display command line parameter. If you do not want Prima to connect to the display, for example, to use it inside of a CGI script, either use the --no-x11 parameter or include the `use Prima::noX11` statement in your program.

### X11: my fonts are bad!

Check whether you have Xft and fontconfig installed. Prima benefits greatly from having been compiled with Xft/fontconfig. Read more in the *Prima::X11* section .

### Where are the docs installed?

Prima documentation comes in .pm and .pod files. These, when installed, are copied under the perl tree, and the man tree in unix. So, 'perldoc Prima' should be sufficient to invoke the main page of the Prima documentation. Other pages can be invoked as 'perldoc Prima::Buttons', say, or, for the graphical pod reader, 'podview Prima::Buttons'. podview is the Prima doc viewer, which is also capable of displaying any POD page.

There is also the pdf file on the Prima website the *http:prima.eu.org* entry that contains the same set of documentation but composed as a single book. Its sources are in the *utils/makedoc* directory, are somewhat rudimentary, and require an installation of latex and dvips to produce one of the tex, dvi, ps, or pdf targets.

### Screen grabbing doesn't work on MacOSX.

It does if you 1) compile Prima with cocoa and 2) allow the application (XQuartz and probably terminal) to access the screen. To do the latter, Choose the Apple menu, System Preferences, click Security & Privacy, then click Privacy. Click on an icon on the left lower corner to allow changes. Then, in the screen recording tab, add XQuartz to the list of allowed applications. Note that it might not work if you run your application from a (remote) ssh session - I couldn't find how to enable screen grabbing for sshd.

### I've found a bug!

the *https:github.comdkPrimaissues* entry is the place.

## PROGRAMMING

### How can I use the .fm files from the Visual Builder inside my program?

podview the *Prima::VB::VBLoader* section

### I want to use Prima inside CGI for loading and converting images only, without an X11 display.

```
use Prima::noX11; # this prevents Prima from connecting to the X11 display
use Prima;
my $i = Prima::Image-> load( ... )
```

Note that drawing on images will be somewhat limited.

### How would I change several properties with a single call?

```
$widget-> set(
   property1 => $value1,
   property2 => $value2,
   ...
);
```

### I want Prima::Edit to have feature XXX

If the feature is not managed by none of the `Prima::Edit` properties, you need to overload `::on_paint`. It is not as hard as you might think.

If the feature is generic enough, you can send a GitHub pull request.

### Tk ( Wx, Qt, whatever ) has a feature Prima doesn't.

Well, I'd probably love to see the feature in Prima as well, but I don't have time to write it myself. Send in a patch, and I promise I'll check it out.

### I wrote a program and it looks ugly with another font size

This would most certainly happen when you rely on your screen properties. There are several ways to avoid this problem.

First, if one programs a window where there are many widgets independent of each other size, one actually can supply coordinates for these widgets as they are positioned on a screen. Don't forget to set the `designScale` property of the parent window, which contains the dimensions of the font used to design the window. One can get these by executing

```
perl -MPrima -MPrima::Application -le '$_=$::application->font; print $_->width, q( ), $_->he
```

This way, the window and the widgets would get resized automatically under another font.

Second, in case the widget layout is not that independent, one can position the widgets relatively to each other by explicitly calculating widget extension. For example, an `InputLine` would have a height relative to the font, and to have a widget placed exactly say 2 pixels above the input line, code something like

```
my $input = $owner-> insert( InputLine, ... );
my $widget = $owner-> insert( Widget, bottom => $input-> top + 2 );
```

Of course, one can change the font as well, but it is a bad idea since users would get annoyed by this.

Third, one can use geometry managers, similar to the ones in Tk. See the *Prima::Widget::pack* section and the *Prima::Widget::place* section.

Finally, check the widget layouts with the *Prima::Stress* section written specifically for this purpose:

```
perl -MPrima::Stress myprogram
```

### How would I write a widget class myself?

There are lots and lots of examples of this. Find a widget class similar to what you are about to write, and follow the idea. There are, though, some non-evident moments worth enumerating.

- Test your widget class with different default settings, such as colors, fonts, parent sizes, and widget properties such as `buffered` and `visible`.

- Try to avoid special properties for `new`, where for example a particular property must always be supplied, or never supplied, or a particular combination of properties is expected. See if the DWIM principle can be applied instead.

- Do not be afraid to define and redefine notification types. These have a large number of options, to be programmed once and then used as DWIM helpers. Consider for what notifications the user callback routines ( onXxxx ) would be best to be called first, or last, and whether a notification should allow multiple callbacks or only one.

  If there is a functionality better off performed by the user-level code, consider creating an individual notification for this purpose.

- Repaint only the changed areas, not the whole widget.

  If your widget has scrollable areas, use the `scroll` method.

  Inside `on_paint` check whether the whole or only a part of the widget is about to be repainted. Simple optimizations here increase the speed.

  Avoid using pre-cooked data in `on_paint`, such as when for example only a particular part of a widget was invalidated, and this fact is stored in an internal variable. This is because when the actual `on_paint` call is executed, the invalid area may be larger than was invalidated by the class actions. If you must though, compare values of the `clipRect` property to see whether the invalid area is indeed the same as it is expected.

  Remember, that inside on_paint all coordinates are inclusive-inclusive, while the widget coordinates generally are inclusive-exclusive.

  Note, that the `buffered` property does not guarantee that the widget output would be buffered. The same goes with `antialias` and `layered`; these functions are opportunistic. Use the `is_surface_buffered`, `can_draw_alpha`, and `is_surface_layered` functions to make sure that these requests were respected.

- Write some documentation and examples of use.

### How would I add my widget class to the VB palette?

Check Prima/VB/examples/Widgety.pm . This file, if loaded through the 'Add widget' command in VB, adds the example widget class and example VB property into the VB palette and Object Inspector.

### How would I use unicode/UTF8 in Prima?

Prima by default is unicode-aware, in some areas more than the Perl (as of 5.38) itself.

For example, on win32 Perl has huge problems with filenames with unicode characters, and this is recommended to mitigate using the *Prima::sys::FS* section, which overrides `open`, `opendir` and the like builtin functions with their unicode-friendly versions. It doesn't though overload `-f` and `-e` syntax, so use `_f`,`_e` etc instead.

Displaying UTF8 text is unproblematic because Perl scalars can be unambiguously told whether the text they contain is in UTF8 or not. The text that comes from the user input, ie keyboard and clipboard, can be treated and reported to Prima either as UTF8 or plain text, depending on the `Prima::Application::wantUnicodeInput` property, which is set to 1 by default. Remember though that if data are to be put through file I/O, the `'utf8'` IO layer must be selected ( see the *open* entry ).

The keyboard input is also easy because a character key event comes with the character code, not the character itself, and conversion between these is done via standard perl's `chr` and `ord`.

The clipboard input is more complicated because the clipboard may contain both UTF8 and plain text data at once, and it must be decided by the programmer explicitly which one is desired. See more in the **Unicode** entry in the *Prima::Clipboard* section.

### Is there a way to display the POD text that comes with my program / package ?

```
$::application-> open_help( "file://$0" );
$::application-> open_help( "file://$0|Description" );
$::application-> open_help( 'My::Package/Bugs' );
```

### How to implement parallel processing?

Prima doesn't work if called from more than one thread, since Perl scalars cannot be shared between threads automatically, but only if explicitly told, by using the *thread::shared* entry. Prima does work in multithread environments though, but only given it runs within a dedicated thread. It is important not to call Prima methods from any other thread because scalars that may be created inside these calls will be unavailable to the Prima core, which would result in strange errors.

It is possible to run things in parallel by calling the event processing by hand: instead of entering the main loop with

```
run Prima;
```

one can write

```
while ( $::application-> yield) {
   ... do some calculations ..
}
```

That'll give Prima a chance to handle accumulated events, but that technique is only viable if calculations can be quantized into relatively short time frames.

The generic solution would be harder to implement and debug, but it scales well. The idea is to fork a process, then communicate with it via its stdin and/or stdout ( see *perlipc* how to do that), and use the *Prima::File* section to asynchronously read data passed through a pipe or a socket.

Note: the Win32 runtime library does not support asynchronous pipes, only asynchronous sockets. Cygwin does support both asynchronous pipes and sockets.

### How do I use Prima with AnyEvent or POE ?

- the *Prima::sys::AnyEvent* section can be used to organize event loops driven by Prima with AnyEvent support:

  ```
  use Prima qw(sys::AnyEvent);
  use AnyEvent;

  my $ev = AnyEvent->timer(after => 1, cb => sub { print "waited 1 second!\n" });
  run Prima;
  ```

  this is the preferred, but not the only solution.

- If you need AnyEvent to drive the event loop, you can fire up the Prima yield() call once in a while:

  ```
  my $timer = AnyEvent->timer(after => 0, interval => 1/20, cb => sub {
      $::application->yield;
  });
  ```

- If you want to use Prima's internal event loop system you have to install the *POE::Loop::Prima* entry and include it in your code before Prima is loaded like below: use POE 'Loop::Prima'; use Prima qw/Application/; use AnyEvent;

  You can call `AnyEvent::detect` to check if the implementation is `'AnyEvent::Impl::POE'` if you want to use Prima's event loop or if it should be the event loop implementation you expect such as `'AnyEvent::Impl::EV'`;

  If you use the *POE::Loop::Prima* entry then you can continue to call `run Prima` and should not call the *AnyEvent* entry's condition variable `recv` function.

- If you want to use another event library implementation of the *AnyEvent* entry, you have to not call `run Prima` but instead call the *AnyEvent* entry's condition variable `recv` function.

  See full examples in *examples/socket_anyevent1.pl*, *examples/socket_anyevent2.pl*, and *examples/socket_anyevent_poe.pl*.

### How do I post an asynchronous message?

The `Prima::Component::post_message` method posts a message through the system event dispatcher and returns immediately; when the message arrives, the `onPostMessage` notification is triggered:

```
use Prima qw(Application);
my $w = Prima::MainWindow-> create( onPostMessage => sub { shift; print "@_\n" });
$w-> post_message(1,2);
print "3 4 ";
run Prima;

output: 3 4 1 2
```

This technique is fine when all calls to the `post_message` on the object are controlled. To multiplex callbacks one can use one of the two scalars passed to `post_message` as callback identification. This is done by the **post** entry in the *Prima::Utils* section, which internally intercepts `$::application`'s `PostMessage` and provides the procedural interface to the same function:

```
use Prima qw(Application);
use Prima::Utils qw(post);

post( sub { print "@_\n" }, 'a');
print "b";
run Prima;

output: ba
```

### Now to address widgets inside TabbedNotebook / TabbedScrollNotebook ?

The tabbed notebooks work as parent widgets for `Prima::Notebook`, which doesn't have any interface elements on its own and provides only a page-flipping function. The sub-widgets, therefore, are to be addressed as `$TabbedNotebook-> Notebook-> MyButton`.

### How to compile a Prima-based module using XS?

Take a look at the *Prima::IPA* section, the *Prima::OpenGL* section, the *Prima::Image::Magick* section, the *PDL::PrimaImage* entry, and the *PDL::Drawing::Prima* entry . These modules compile against the Prima dynamic module and start from there. Note - it's important to include PRIMA_VERSION_BOOTCHECK in the "BOOT:" section, to avoid binary incompatibilities if there should be any.

### How do I generate Prima executables with PAR?

You'll need some files that PAR cannot detect automatically. During the compilation phase Makefile.PL creates the *utils/par.txt* file that contains these files. Include them with this command:

```
pp -A utils/par.txt -o a.out my_program
```

## 11.2 Prima::Const

Predefined constants

### Description

`Prima::Const` and the *Prima::Classes* section for a minimal set of perl modules needed for the toolkit. Since the module provides bindings for the core constants, it is required to be included in every Prima-related module and program.

The constants are collected under the top-level package names, with no `Prima::` prefix. This violates the perl guidelines about package naming, however, it was considered way too inconvenient to prefix every constant with a `Prima::` string.

This document describes all constants defined in the core. The constants are also described in the articles together with the corresponding methods and properties. For example, the `nt` constants are also described in the the **Flow** entry in the *Prima::Object* section article.

### API

#### am:: - Prima::Icon auto masking

See also the **autoMasking** entry in the *Prima::Image* section

```
am::None           - no mask update performed
am::MaskColor      - mask update based on Prima::Icon::maskColor property
am::MaskIndex      - mask update based on Prima::Icon::maskIndex property
am::Auto           - mask update based on corner pixel values
```

#### apc:: - OS type

See the **get_system_info** entry in the *Prima::Application* section

```
apc::Win32
apc::Unix
```

#### bi:: - border icons

See the **borderIcons** entry in the *Prima::Window* section

```
bi::SystemMenu  - the system menu button and/or close button
                  ( usually with the icon )
bi::Minimize    - minimize button
bi::Maximize    - maximize/restore button
bi::TitleBar    - the window title
bi::All         - all of the above
```

#### bs:: - border styles

See the **borderStyle** entry in the *Prima::Window* section

```
bs::None      - no border
bs::Single    - thin border
bs::Dialog    - thick border
bs::Sizeable  - border that can be resized
```

**ci:: - color indices**

See the **colorIndex** entry in the *Prima::Widget* section

```
ci::NormalText or ci::Fore
ci::Normal or ci::Back
ci::HiliteText
ci::Hilite
ci::DisabledText
ci::Disabled
ci::Light3DColor
ci::Dark3DColor
ci::MaxId
```

**cl:: - colors**

See the **colorIndex** entry in the *Prima::Widget* section

**Direct color constants**

```
cl::Black
cl::Blue
cl::Green
cl::Cyan
cl::Red
cl::Magenta
cl::Brown
cl::LightGray
cl::DarkGray
cl::LightBlue
cl::LightGreen
cl::LightCyan
cl::LightRed
cl::LightMagenta
cl::Yellow
cl::White
cl::Gray
```

**Indirect color constants**

```
cl::NormalText, cl::Fore
cl::Normal, cl::Back
cl::HiliteText
cl::Hilite
cl::DisabledText
cl::Disabled
cl::Light3DColor
cl::Dark3DColor
cl::MaxSysColor
```

**Special constants**

See the **Colors** entry in the *Prima::gp_problems* section

```
cl::Set      - logical all-1 color
cl::Clear    - logical all-0 color
```

```
cl::Invalid  - invalid color value
cl::SysFlag  - indirect color constant bit set
cl::SysMask  - indirect color constant bit clear mask
```

**Color functions**

**from_rgb R8,G8,B8 -> RGB24**

**to_rgb RGB24 -> R8,G8,B8**

**from_bgr B8,G8,R8 -> RGB24**

**to_bgr RGB24 -> B8,G8,R8**

**to_gray_byte RGB24 -> GRAY8**

**to_gray_rgb RGB24 -> GRAY24**

**from_gray_byte GRAY8 -> GRAY24**

**premultiply RGB24,A8 -> RGB24**

**distance RGB24,RGB24 -> distance between colors**

**blend RGB24,RGB24,AMOUNT_FROM_0_TO_1 - RGB24**

**cm:: - commands**

**Keyboard and mouse commands**

See the **key_down** entry in the *Prima::Widget* section, the **mouse_down** entry in the *Prima::Widget* section

```
cm::KeyDown
cm::KeyUp
cm::MouseDown
cm::MouseUp
cm::MouseClick
cm::MouseWheel
cm::MouseMove
cm::MouseEnter
cm::MouseLeave
```

**Internal commands ( used in core only or not used at all )**

```
cm::Close
cm::Create
cm::Destroy
cm::Hide
cm::Show
cm::ReceiveFocus
cm::ReleaseFocus
cm::Paint
cm::Repaint
cm::Size
cm::Move
cm::ColorChanged
cm::ZOrderChanged
cm::Enable
cm::Disable
cm::Activate
```

```
cm::Deactivate
cm::FontChanged
cm::WindowState
cm::Timer
cm::Click
cm::CalcBounds
cm::Post
cm::Popup
cm::Execute
cm::Setup
cm::Hint
cm::DragDrop
cm::DragOver
cm::EndDrag
cm::Menu
cm::EndModal
cm::MenuCmd
cm::TranslateAccel
cm::DelegateKey
```

**cr:: - pointer cursor resources**

See the **pointerType** entry in the *Prima::Widget* section

```
cr::Default                 same pointer type as owner's
cr::Arrow                   arrow pointer
cr::Text                    text entry cursor-like pointer
cr::Wait                    hourglass
cr::Size                    general size action pointer
cr::Move                    general move action pointer
cr::SizeWest, cr::SizeW     right-move action pointer
cr::SizeEast, cr::SizeE     left-move action pointer
cr::SizeWE                  general horizontal-move action pointer
cr::SizeNorth, cr::SizeN    up-move action pointer
cr::SizeSouth, cr::SizeS    down-move action pointer
cr::SizeNS                  general vertical-move action pointer
cr::SizeNW                  up-right move action pointer
cr::SizeSE                  down-left move action pointer
cr::SizeNE                  up-left move action pointer
cr::SizeSW                  down-right move action pointer
cr::Invalid                 invalid action pointer
cr::DragNone                pointer for an invalid dragging target
cr::DragCopy                pointer to indicate that a dnd::Copy action can be accepted
cr::DragMove                pointer to indicate that a dnd::Move action can be accepted
cr::DragLink                pointer to indicate that a dnd::Link action can be accepted
cr::Crosshair               the crosshair pointer
cr::UpArrow                 arrow directed upwards
cr::QuestionArrow           question mark pointer
cr::User                    user-defined icon
```

**dbt:: - device bitmap types**

```
dbt::Bitmap                 monochrome 1-bit bitmap
dbt::Pixmap                 bitmap compatible with display format
dbt::Layered                bitmap compatible with display format with alpha channel
```

**dnd:: - drag and drop action constants and functions**

```
dnd::None                   no DND action was selected or performed
dnd::Copy                   copy action
dnd::Move                   move action
dnd::Link                   link action
dnd::Mask                   combination of all valid actions
```

**is_one_action ACTIONS**

Returns true if `ACTIONS` is not a combination of `dnd::` constants.

**pointer ACTION**

Returns a `cr::` constant corresponding to the `ACTION`

**to_one_action ACTIONS**

Selects the best single action from a combination of allowed `ACTIONS`

**keymod ACTION**

Returns a `km::` keyboard modifier constant that would initiate `ACTION` if the user presses it during a DND session. Returns 0 for `dnd::Copy` which is the standard action to be performed without any modifiers.

**dt:: - drive types**

See the **query_drive_type** entry in the *Prima::Utils* section

```
dt::None
dt::Unknown
dt::Floppy
dt::HDD
dt::Network
dt::CDROM
dt::Memory
```

**dt:: - Prima::Drawable::draw_text constants**

```
dt::Left              - text is aligned to the left boundary
dt::Right             - text is aligned to the right boundary
dt::Center            - text is aligned horizontally in the center
dt::Top               - text is aligned to the upper boundary
dt::Bottom            - text is aligned to the lower boundary
dt::VCenter           - text is aligned vertically in the center
dt::DrawMnemonic      - tilde-escapement and underlining is used
dt::DrawSingleChar    - sets tw::BreakSingle option to
                        Prima::Drawable::text_wrap call
dt::NewLineBreak      - sets tw::NewLineBreak option to
                        Prima::Drawable::text_wrap call
dt::SpaceBreak        - sets tw::SpaceBreak option to
                        Prima::Drawable::text_wrap call
dt::WordBreak         - sets tw::WordBreak option to
                        Prima::Drawable::text_wrap call
dt::ExpandTabs        - performs tab character ( \t ) expansion
dt::DrawPartial       - draws the last line, if it is visible partially
dt::UseExternalLeading- text lines positioned vertically with respect to
                        the font external leading
```

```
dt::UseClip           - assign ::clipRect property to the boundary rectangle
dt::QueryLinesDrawn   - calculates and returns the number of lines drawn
                        ( contrary to dt::QueryHeight )
dt::QueryHeight       - if set, calculates and returns vertical extension
                        of the lines drawn
dt::NoWordWrap        - performs no word wrapping by the width of the boundaries
dt::WordWrap          - performs word wrapping by the width of the boundaries
dt::Default           - dt::NewLineBreak|dt::WordBreak|dt::ExpandTabs|
                        dt::UseExternalLeading
```

**fdo:: - find / replace dialog options**

See the *Prima::FindDialog* section

```
fdo::MatchCase
fdo::WordsOnly
fdo::RegularExpression
fdo::BackwardSearch
fdo::ReplacePrompt
```

**fds:: - find / replace dialog scope type**

See the *Prima::FindDialog* section

```
fds::Cursor
fds::Top
fds::Bottom
```

**fe:: - file events constants**

See the *Prima::File* section

```
fe::Read
fe::Write
fe::Exception
```

**fm:: - fill modes**

See the **fillMode** entry in the *Prima::Drawable* section

```
fp::Alternate
fp::Winding
fp::Overlay
```

**fp:: - standard fill pattern indices**

See the **fillPattern** entry in the *Prima::Drawable* section

```
fp::Empty
fp::Solid
fp::Line
fp::LtSlash
fp::Slash
fp::BkSlash
fp::LtBkSlash
fp::Hatch
```

```
fp::XHatch
fp::Interleave
fp::WideDot
fp::CloseDot
fp::SimpleDots
fp::Borland
fp::Parquet
```

**builtin $FILL_PATTERN**

Given a result from `Drawable::fillPattern`, an 8x8 array of integers, checks whether the array matches one of the builtin `fp::` constants, and returns one if found. Returns undef otherwise.

**is_empty $FILL_PATTERN**

Given a result from `Drawable::fillPattern`, an 8x8 array of integers, checks if the array is all zeros

**is_solid $FILL_PATTERN**

Given a result from `Drawable::fillPattern`, an 8x8 array of integers, checks if the array is all ones (ie 0xff)

**patterns**

Returns a set of string-encoded fill patterns that correspond to the builtin `fp::` constants. These are not suitable for use in `Drawable::fillPatterns`.

### fp:: - font pitches

See the **pitch** entry in the *Prima::Drawable* section

```
fp::Default
fp::Fixed
fp::Variable
```

### fr:: - fetch resource constants

See the **fetch_resource** entry in the *Prima::Widget* section

```
fr::Color
fr::Font
fs::String
```

### fs:: - font styles

See the **style** entry in the *Prima::Drawable* section

```
fs::Normal
fs::Bold
fs::Thin
fs::Italic
fs::Underlined
fs::StruckOut
fs::Outline
```

**fw:: - font weights**

See the **weight** entry in the *Prima::Drawable* section

```
fw::UltraLight
fw::ExtraLight
fw::Light
fw::SemiLight
fw::Medium
fw::SemiBold
fw::Bold
fw::ExtraBold
fw::UltraBold
```

**ggo:: - glyph outline commands**

```
ggo::Move
ggo::Line
ggo::Conic
ggo::Cubic
```

See also the **render_glyph** entry in the *Prima::Drawable* section

**gm:: - grow modes**

See the **growMode** entry in the *Prima::Widget* section

**Basic constants**

```
gm::GrowLoX     widget's left side is kept in constant
                distance from the owner's right side
gm::GrowLoY     widget's bottom side is kept in constant
                distance from the owner's top side
gm::GrowHiX     widget's right side is kept in constant
                distance from the owner's right side
gm::GrowHiY     widget's top side is kept in constant
                distance from the owner's top side
gm::XCenter     widget is kept in the center on its owner's
                horizontal axis
gm::YCenter     widget is kept in the center on its owner's
                vertical axis
gm::DontCare    widgets origin is constant relative
                to the screen
```

**Derived or aliased constants**

```
gm::GrowAll      gm::GrowLoX|gm::GrowLoY|gm::GrowHiX|gm::GrowHiY
gm::Center       gm::XCenter|gm::YCenter
gm::Client       gm::GrowHiX|gm::GrowHiY
gm::Right        gm::GrowLoX|gm::GrowHiY
gm::Left         gm::GrowHiY
gm::Floor        gm::GrowHiX
```

**gui:: - GUI types**

See the **get_system_info** entry in the *Prima::Application* section

```
gui::Default
gui::Windows
gui::XLib
gui::GTK
```

**le:: - line end styles**

See the **lineEnd** entry in the *Prima::Drawable* section

```
le::Flat
le::Square
le::Round

le::Arrow
le::Cusp
le::InvCusp
le::Knob
le::Rect
le::RoundRect
le::Spearhead
le::Tail
```

Functions:

```
le::transform($matrix)
le::scale($scalex, [$scaley = $scalex])
```

**lei:: - line end indexes**

```
lei::LineTail
lei::LineHead
lei::ArrowTail
lei::ArrowHead
lei::Max
lei::Only
```

See the **lineEndIndex** entry in the *Prima::Drawable* section

**lj:: - line join styles**

See the **lineJoin** entry in the *Prima::Drawable* section

```
lj::Round
lj::Bevel
lj::Miter
```

**lp:: - predefined line pattern styles**

See the **linePattern** entry in the *Prima::Drawable* section

```
lp::Null           #    ""              /*              */
lp::Solid          #    "\1"            /* ___________  */
lp::Dash           #    "\x9\3"         /* __ __ __ __  */
lp::LongDash       #    "\x16\6"        /* _____ _____  */
lp::ShortDash      #    "\3\3"          /* _ _ _ _ _ _  */
lp::Dot            #    "\1\3"          /* . . . . . .  */
lp::DotDot         #    "\1\1"          /* ............ */
lp::DashDot        #    "\x9\6\1\3"     /* _._._._._._  */
lp::DashDotDot     #    "\x9\3\1\3\1\3" /* _.._.._.._.. */
```

**im:: - image types**

See the **type** entry in the *Prima::Image* section.

**Bit depth constants**

```
im::bpp1
im::bpp4
im::bpp8
im::bpp16
im::bpp24
im::bpp32
im::bpp64
im::bpp128
```

**Pixel format constants**

```
im::Color
im::GrayScale
im::RealNumber
im::ComplexNumber
im::TrigComplexNumber
im::SignedInt
```

**Mnemonic image types**

```
im::Mono          - im::bpp1
im::BW            - im::bpp1 | im::GrayScale
im::16            - im::bpp4
im::Nibble        - im::bpp4
im::256           - im::bpp8
im::RGB           - im::bpp24
im::Triple        - im::bpp24
im::Byte          - gray 8-bit unsigned integer
im::Short         - gray 16-bit unsigned integer
im::Long          - gray 32-bit unsigned integer
im::Float         - float
im::Double        - double
im::Complex       - dual float
im::DComplex      - dual double
im::TrigComplex   - dual float
im::TrigDComplex  - dual double
```

**Extra formats**

```
im::fmtBGR
im::fmtRGBI
im::fmtIRGB
im::fmtBGRI
im::fmtIBGR
```

**Masks**

```
im::BPP      - bit depth constants
im::Category - category constants
im::FMT      - extra format constants
```

**ict:: - image conversion types**

See the **conversion** entry in the *Prima::Image* section.

```
ict::None            - no dithering, with static palette or palette optimized by the sour
ict::Posterization   - no dithering, with palette optimized by the source pixels
ict::Ordered         - 8x8 ordered halftone dithering
ict::ErrorDiffusion  - error diffusion dithering with a static palette
ict::Optimized       - error diffusion dithering with an optimized palette
```

Their values are combinations of `ictp::` and `ictd::` constants, see below.

**ictd:: - image conversion types, dithering**

These constants select the color correction (dithering) algorithm when downsampling an image

```
ictd::None            - no dithering, pure colors only
ictd::Ordered         - 8x8 ordered halftone dithering (checkerboard)
ictd::ErrorDiffusion  - error diffusion dithering (2/5 down, 2/5 right, 1/5 down/right)
```

**ictp:: - image conversion types, palette optimization**

These constants select how the target palette is made up when downsampling an image.

```
ictp::Unoptimized  - use whatever color mapping method is fastest,
                     image quality can be severely compromised
ictp::Cubic        - use static cubic palette; a bit slower,
                     guaranteed mediocre quality
ictp::Optimized    - collect available colors in the image;
                     slowest, gives the best results
```

Not all combinations of ictp and ictd constants are valid

**is:: - image statistics indices**

See the **stats** entry in the *Prima::Image* section.

```
is::RangeLo  - minimum pixel value
is::RangeHi  - maximum pixel value
is::Mean     - mean value
is::Variance - variance
is::StdDev   - standard deviation
is::Sum      - the sum of pixel values
is::Sum2     - the sum of squares of pixel values
```

**ist:: - image scaling types**

```
ist::None      - image stripped or padded with zeros
ist::Box       - the image will be scaled using a simple box transform
ist::BoxX      - columns behave as ist::None, rows as ist::Box
ist::BoxY      - rows behave as in ist::None, columns as ist::Box
ist::AND       - shrunken pixels AND-end together (black-on-white images)
ist::OR        - shrunken pixels OR-end together (white-on-black images)
ist::Triangle  - bilinear interpolation
ist::Quadratic - 2nd order (quadratic) B-Spline approximation of the Gaussian
ist::Sinc      - sine function
ist::Hermite   - B-Spline interpolation
ist::Cubic     - 3rd order (cubic) B-Spline approximation of the Gaussian
ist::Gaussian  - Gaussian transform with gamma=0.5
```

See the **scaling** entry in the *Prima::Image* section.

**kb:: - keyboard virtual codes**

See also the **KeyDown** entry in the *Prima::Widget* section.

**Modificator keys**

```
kb::ShiftL   kb::ShiftR   kb::CtrlL      kb::CtrlR
kb::AltL     kb::AltR     kb::MetaL      kb::MetaR
kb::SuperL   kb::SuperR   kb::HyperL     kb::HyperR
kb::CapsLock kb::NumLock  kb::ScrollLock kb::ShiftLock
```

**Keys with character code defined**

```
kb::Backspace  kb::Tab    kb::Linefeed   kb::Enter
kb::Return     kb::Escape kb::Esc        kb::Space
```

**Function keys**

```
kb::F1 .. kb::F30
kb::L1 .. kb::L10
kb::R1 .. kb::R10
```

**Other**

```
kb::Clear    kb::Pause   kb::SysRq  kb::SysReq
kb::Delete   kb::Home    kb::Left   kb::Up
kb::Right    kb::Down    kb::PgUp   kb::Prior
kb::PageUp   kb::PgDn    kb::Next   kb::PageDown
kb::End      kb::Begin   kb::Select kb::Print
kb::PrintScr kb::Execute kb::Insert kb::Undo
kb::Redo     kb::Menu    kb::Find   kb::Cancel
kb::Help     kb::Break   kb::BackTab
```

**Masking constants**

```
kb::CharMask - character codes
kb::CodeMask - virtual key codes ( all other kb:: values )
kb::ModMask  - km:: values
```

### km:: - keyboard modifiers

See also the **KeyDown** entry in the *Prima::Widget* section.

```
km::Shift
km::Ctrl
km::Alt
km::KeyPad
km::DeadKey
km::Unicode
```

### mt:: - modality types

See the **get_modal** entry in the *Prima::Window* section, the **get_modal_window** entry in the *Prima::Window* section

```
mt::None
mt::Shared
mt::Exclusive
```

### nt:: - notification types

Used in `Prima::Component::notification_types` to describe event flow.
See also the **Flow** entry in the *Prima::Object* section.

#### Starting point constants

```
nt::PrivateFirst
nt::CustomFirst
```

#### Direction constants

```
nt::FluxReverse
nt::FluxNormal
```

#### Complexity constants

```
nt::Single
nt::Multiple
nt::Event
```

#### Composite constants

```
nt::Default       ( PrivateFirst | Multiple | FluxReverse)
nt::Property      ( PrivateFirst | Single   | FluxNormal )
nt::Request       ( PrivateFirst | Event    | FluxNormal )
nt::Notification  ( CustomFirst  | Multiple | FluxReverse )
nt::Action        ( CustomFirst  | Single   | FluxReverse )
nt::Command       ( CustomFirst  | Event    | FluxReverse )
```

**mb:: - mouse buttons**

See also the **MouseDown** entry in the *Prima::Widget* section.

```
mb::b1 or mb::Left
mb::b2 or mb::Middle
mb::b3 or mb::Right
mb::b4
mb::b5
mb::b6
mb::b7
mb::b8
```

**mb:: - message box constants**

**Message box and modal result button commands**

See also the **modalResult** entry in the *Prima::Window* section, the **modalResult** entry in the *Prima::Button* section.

```
mb::OK, mb::Ok
mb::Cancel
mb::Yes
mb::No
mb::Abort
mb::Retry
mb::Ignore
mb::Help
```

**Message box composite ( multi-button ) constants**

```
mb::OKCancel, mb::OkCancel
mb::YesNo
mb::YesNoCancel
mb::ChangeAll
```

**Message box icon and bell constants**

```
mb::Error
mb::Warning
mb::Information
mb::Question
```

**ps:: - paint states**

```
ps::Disabled    - can neither draw, nor get/set graphical properties on an object
ps::Enabled     - can both draw and get/set graphical properties on an object
ps::Information - can only get/set graphical properties on an object
```

For brevity, ps::Disabled is equal to 0 so this allows for simple boolean testing if one can get/set graphical properties on an object.

See the **get_paint_state** entry in the *Drawable* section.

**rgn:: - result of Prima::Region.rect_inside**

```
rgn::Inside     - the rectangle is fully inside the region
rgn::Outside    - the rectangle is fully outside the region
rgn::Partially  - the rectangle overlaps the region but is not fully inside
```

**rgnop:: - Prima::Region.combine set operations**

```
rgnop::Copy
rgnop::Intersect
rgnop::Union
rgnop::Xor
rgnop::Diff
```

**rop:: - raster operation codes**

See the **Raster operations** entry in the *Prima::Drawable* section

```
rop::Blackness      #   = 0
rop::NotOr          #   = !(src | dest)
rop::NotSrcAnd      #  &= !src
rop::NotPut         #   = !src
rop::NotDestAnd     #   = !dest & src
rop::Invert         #   = !dest
rop::XorPut         #  ^= src
rop::NotAnd         #   = !(src & dest)
rop::AndPut         #  &= src
rop::NotXor         #   = !(src ^ dest)
rop::NotSrcXor      #     alias for rop::NotXor
rop::NotDestXor     #     alias for rop::NotXor
rop::NoOper         #   = dest
rop::NotSrcOr       #  |= !src
rop::CopyPut        #   = src
rop::NotDestOr      #   = !dest | src
rop::OrPut          #  |= src
rop::Whiteness      #   = 1
```

12 Porter-Duff operators

```
rop::Clear       # same as rop::Blackness, = 0
rop::XorOver     # = src ( 1 - dstA ) + dst ( 1 - srcA )
rop::SrcOver     # = src srcA + dst (1 - srcA)
rop::DstOver     # = dst srcA + src (1 - dstA)
rop::SrcCopy     # same as rop::CopyPut,   = src
rop::DstCopy     # same as rop::NoOper,    = dst
rop::SrcIn       # = src dstA
rop::DstIn       # = dst srcA
rop::SrcOut      # = src ( 1 - dstA )
rop::DstOut      # = dst ( 1 - srcA )
rop::SrcAtop     # = src dstA + dst ( 1 - srcA )
rop::DstAtop     # = dst srcA + src ( 1 - dstA )

rop::Blend       # src + dst (1 - srcA)
                 # same as rop::SrcOver but assumes the premultiplied source

rop::PorterDuffMask - masks out all bits but the constants above
```

Photoshop operators

```
rop::Add
rop::Multiply
rop::Screen
```

```
rop::Overlay
rop::Darken
rop::Lighten
rop::ColorDodge
rop::ColorBurn
rop::HardLight
rop::SoftLight
rop::Difference
rop::Exclusion
```

Special flags

```
rop::SrcAlpha           # The combination of these four flags
rop::SrcAlphaShift      # may encode extra source and destination
rop::DstAlpha           # alpha values in cases either where there is none
rop::DstAlphaShift      # in the images, or as additional blend factors.
                        #
rop::ConstantAlpha      # (same as rop::SrcAlpha|rop::DstAlpha)

rop::AlphaCopy          # source image is treated a 8-bit grayscale alpha
rop::ConstantColor      # foreground color is used to fill the color bits

rop::Default            # rop::SrcOver for ARGB destinations, rop::CopyPut otherwise
```

ROP functions

**alpha ROP, SRC_ALPHA = undef, DST_ALPHA = undef**

Combines one of the alpha-supporting ROPs ( Porter-Duff and Photoshop operators) with source and destination alpha, if defined, and returns a new ROP constant. This is useful when blending with constant alpha is required with/over images that don't have their own alpha channel. Or as an additional alpha channel when using icons.

**blend ALPHA**

Creates a ROP that would effectively execute alpha blending of the source image over the destination image with ALPHA value.

### sbmp:: - system bitmaps indices

See also the *Prima::StdBitmap* section.

```
sbmp::Logo
sbmp::CheckBoxChecked
sbmp::CheckBoxCheckedPressed
sbmp::CheckBoxUnchecked
sbmp::CheckBoxUncheckedPressed
sbmp::RadioChecked
sbmp::RadioCheckedPressed
sbmp::RadioUnchecked
sbmp::RadioUncheckedPressed
sbmp::Warning
sbmp::Information
sbmp::Question
sbmp::OutlineCollapse
sbmp::OutlineExpand
sbmp::Error
```

```
sbmp::SysMenu
sbmp::SysMenuPressed
sbmp::Max
sbmp::MaxPressed
sbmp::Min
sbmp::MinPressed
sbmp::Restore
sbmp::RestorePressed
sbmp::Close
sbmp::ClosePressed
sbmp::Hide
sbmp::HidePressed
sbmp::DriveUnknown
sbmp::DriveFloppy
sbmp::DriveHDD
sbmp::DriveNetwork
sbmp::DriveCDROM
sbmp::DriveMemory
sbmp::GlyphOK
sbmp::GlyphCancel
sbmp::SFolderOpened
sbmp::SFolderClosed
sbmp::Last
```

**scr:: - scroll exposure results**

`Widget::scroll` returns one of these.

```
scr::Error          - failure
scr::NoExpose       - call resulted in no new exposed areas
scr::Expose         - call resulted in new exposed areas, expect a repaint
```

**sv:: - system value indices**

See also the **get_system_value** entry in the *Prima::Application* section

```
sv::YMenu           - the height of the menu bar in top-level windows
sv::YTitleBar       - the height of the title bar in top-level windows
sv::XIcon           - width and height of main icon dimensions,
sv::YIcon             acceptable by the system
sv::XSmallIcon      - width and height of alternate icon dimensions,
sv::YSmallIcon        acceptable by the system
sv::XPointer        - width and height of mouse pointer icon
sv::YPointer          acceptable by the system
sv::XScrollbar      - the width of the default vertical scrollbar
sv::YScrollbar      - the height of the default horizontal scrollbar
sv::XCursor         - width of the system cursor
sv::AutoScrollFirst - the initial and the repetitive
sv::AutoScrollNext    scroll timeouts
sv::InsertMode      - the system insert mode
sv::XbsNone         - widths and heights of the top-level window
sv::YbsNone           decorations, correspondingly, with borderStyle
sv::XbsSizeable       bs::None, bs::Sizeable, bs::Single, and
sv::YbsSizeable       bs::Dialog.
sv::XbsSingle
```

```
sv::YbsSingle
sv::XbsDialog
sv::YbsDialog
sv::MousePresent     - 1 if the mouse is present, 0 otherwise
sv::MouseButtons     - number of the mouse buttons
sv::WheelPresent     - 1 if the mouse wheel is present, 0 otherwise
sv::SubmenuDelay     - timeout ( in ms ) before a sub-menu shows on
                       an implicit selection
sv::FullDrag         - 1 if the top-level windows are dragged dynamically,
                       0 - with marquee mode
sv::DblClickDelay    - mouse double-click timeout in milliseconds
sv::ShapeExtension   - 1 if Prima::Widget::shape functionality is supported,
                       0 otherwise
sv::ColorPointer     - 1 if the system accepts color pointer icons.
sv::CanUTF8_Input    - 1 if the system can generate key codes in unicode
sv::CanUTF8_Output   - 1 if the system can output utf8 text
sv::CompositeDisplay - 1 if the system uses double-buffering and alpha composition for th
                       0 if it doesn't, -1 if unknown
sv::LayeredWidgets   - 1 if the system supports layering
sv::FixedPointerSize - 0 if the system doesn't support arbitrarily sized pointers and wil
sv::MenuCheckSize    - width and height of default menu check icon
sv::FriBidi          - 1 if Prima is compiled with libfribidi and full bidi unicode suppo
sv::Antialias        - 1 if the system supports antialiasing and alpha layer for primitiv
sv::LibThai          - 1 if Prima is compiled with libthai
```

**ta:: - alignment constants**

Used in: the *Prima::InputLine* section, the *Prima::ImageViewer* section, the *Prima::Label* section.

```
ta::Left
ta::Right
ta::Center

ta::Top
ta::Bottom
ta::Middle
```

**to:: - text output constants**

These constants are used in various text- and glyph-related functions and form a somewhat vague group of bit values that may or may not be used together depending on the function

```
to::Plain         - default value, 0
to::AddOverhangs  - used in C<get_text_width> and C<get_text_shape_width>
                    to request text overhangs to be included in the returned
                    text width
to::Glyphs        - used in C<get_font_abc> and C<get_font_def> to select extension of
                    glyph indexes rather than text codepoints
to::Unicode       - used in C<get_font_abc> and C<get_font_def> to select extension of
                    unicode rather than ascii text codepoints
to::RTL           - used in C<get_text_shape_width> to request RTL bidi direction.
                    Also used in C<Prima::Drawable::Glyphs::indexes> values to mark
                    RTL characters.
```

**tw:: - text wrapping constants**

See the **text_wrap** entry in the *Prima::Drawable* section

```
tw::CalcMnemonic          - calculates tilde underline position
tw::CollapseTilde         - removes escaping tilde from text
tw::CalcTabs              - wraps the text with respect to tab expansion
tw::ExpandTabs            - expands tab characters
tw::BreakSingle           - determines if the text is broken into single
                            characters when text cannot be fit
tw::NewLineBreak          - breaks line on newline characters
tw::SpaceBreak            - breaks line on space or tab characters
tw::ReturnChunks          - returns wrapped text chunks
tw::ReturnLines           - returns positions and lengths of wrapped
                            text chunks
tw::WordBreak             - defines if text break by width goes by the
                            characters or by the words
tw::ReturnFirstLineLength - returns the length of the first wrapped line
tw::Default               - tw::NewLineBreak | tw::CalcTabs | tw::ExpandTabs |
                            tw::ReturnLines | tw::WordBreak
```

**wc:: - widget classes**

See the **widgetClass** entry in the *Prima::Widget* section

```
wc::Undef
wc::Button
wc::CheckBox
wc::Combo
wc::Dialog
wc::Edit
wc::InputLine
wc::Label
wc::ListBox
wc::Menu
wc::Popup
wc::Radio
wc::ScrollBar
wc::Slider
wc::Widget, wc::Custom
wc::Window
wc::Application
```

**ws:: - window states**

See the **windowState** entry in the *Prima::Window* section

```
ws::Normal
ws::Minimized
ws::Maximized
ws::Fullscreen
```

## 11.3 Prima::EventHook

Event filtering

### Synopsis

```
use Prima::EventHook;

sub hook
{
        my ( $my_param, $object, $event, @params) = @_;
        ...
        print "Object $object received event $event\n";
        ...
        return 1;
}

Prima::EventHook::install( \&hook,
        param    => $my_param,
        object   => $my_window,
        event    => [qw(Size Move Destroy)],
        children => 1
);

Prima::EventHook::deinstall(\&hook);
```

### Description

The toolkit dispatches notifications by calling subroutines registered on one or more objects. Also, the core part of the toolkit allows a single event hook callback to be installed that would receive all events occurring on all objects. `Prima::EventHook` provides multiplexed access to the core event hook and introduces a set of dispatching rules so that the user hooks can receive only a subset of events.

### API

#### install SUB, %RULES

Installs SUB using a hash of RULES.

The SUB is called with a variable list of parameters, formed so that first come parameters from the `'param'` key ( see below ), then the event source object, then the event name, and finally the parameters to the event. The SUB must return an integer, either 0 or 1, to block or pass the event, respectively. If 1 is returned, other hook subs are called; if 0 is returned, the event is efficiently blocked and no hooks are called further.

Rules can contain the following keys:

**event**

An event is either a string, an array of strings, or an `undef` value. In the latter case, it is equal to a `'*'` string which selects all events to be passed to the SUB. A string is either the name of an event or one of the pre-defined event groups, declared in the `%groups` package hash. The group names are:

```
ability
focus
geometry
```

```
keyboard
menu
mouse
objects
visibility
```

These contain the respective events. See the source for a detailed description.

In case the `'event'` key is an array of strings, each of the strings is also the name of either an event or a group. In this case, if the `'*'` string or event duplicate names are present in the list, SUB is called several times.

**object**

A Prima object, or an array of Prima objects, or undef; in the latter case matches all objects. If an object is defined, the SUB is called if the event source is the same as the object.

**children**

If 1, SUB is called using the same rules as described in `'object'`, but also if the event source is a child of the object. Thus, selecting `undef` as a filter object and setting `'children'` to 0 is almost the same as selecting `$::application`, which is the root of the Prima object hierarchy, as a filter object with `'children'` set to 1.

Setting `object` to `undef` and children to 1 is inefficient.

**param**

A scalar or array of scalars passed as first parameters to SUB

### deinstall SUB

Removes the hook sub

## NOTES

`Prima::EventHook` by default automatically starts and stops the Prima event hook mechanism when appropriate. If it is not desired, for example for your own event hook management, set `$auto_hook` to 0.

## 11.4 Prima::Image::Animate

Animate gif,webp,png files

### Description

The module provides high-level access to GIF, APNG, and WebP animation sequences.

### Synopsis

```
use Prima qw(Application Image::Animate);
my $x = Prima::Image::Animate->load($ARGV[0]);
die $@ unless $x;
my ( $X, $Y) = ( 0, 100);
my $want_background = 1; # 0 for eventual transparency
my $background = $::application-> get_image( $X, $Y, $x-> size);
$::application-> begin_paint;

while ( my $info = $x-> next) {
        my $frame = $background-> dup;
        $frame-> begin_paint;
        $x-> draw_background( $frame, 0, 0) if $want_background;
        $x-> draw( $frame, 0, 0);
        $::application-> put_image( $X, $Y, $frame);

        $::application-> sync;
        select(undef, undef, undef, $info-> {delay});
}

$::application-> put_image( $X, $Y, $g);
```

#### new $CLASS, %OPTIONS

Creates an empty animation container. If `$OPTIONS{images}` is given, it is expected to be an array of images, best if loaded from files with the `loadExtras` and `iconUnmask` parameters set ( see the *Prima::image-load* section for details).

#### detect_animation $HASH

Checks the {`extras`} `hash` obtained from an image loaded with the `loadExtras` flag set to detect whether the image is an animation or not, and if loading of all of its frame is supported by the module. Returns the file format name on success, undef otherwise.

#### load $SOURCE, %OPTIONS

Loads a GIF, APNG, or WebP animation sequence from `$SOURCE` which is either a file or a stream. Options are the same as used by the `Prima::Image::load` method.

Depending on the `loadAll` option, either loads all frames at once (1), or uses `Prima::Image::Loader` to load only a single frame at a time (0, default). Depending on the loading mode, some properties may not be available.

#### add $IMAGE

Appends an image frame to the container.

Only available if the `loadAll` option is on.

**bgColor**

Returns the background color specified by the sequence as the preferred color to use when there is no specific background to superimpose the animation on.

**close**

Releases eventual image file handle for loader-based animations. Sets the {`suspended`} flag so that all image operations are suspended. A later call to `reload` restores the status quo execpt the current frame prior to the `close` call.

Has no effect on animations loaded with the `loadAll` option.

**current**

Returns the index of the current frame

**draw $CANVAS, $X, $Y**

Draws the current composite frame on `$CANVAS` at the given coordinates

**draw_background $CANVAS, $X, $Y**

Fills the background on `$CANVAS` at the given coordinates if the file provides the color to fill. Returns a boolean value whether the canvas was drawn on or not.

**height**

Returns the height of the composite frame

**icon**

Returns a new icon object created from the current composite frame

**image**

Returns a new image object created from the current composite frame The transparent pixels on the image are replaced with the preferred background color

**is_stopped**

Returns true if the animation sequence was stopped, false otherwise. If the sequence was stopped, the only way to restart it is to call `reset`.

**length**

Returns the total animation length (without repeats) in seconds.

**loopCount [ INTEGER ]**

Sets and returns the number of loops left, undef for indefinite.

**next**

Advances one animation frame. The step triggers changes to the internally kept buffer image that creates the effect of transparency if needed. The method returns a hash, where the following fields are initialized:

**left, bottom, right, top**

Coordinates of the changed area since the last frame was updated

**delay**

Time in seconds how long the frame is expected to be displayed

**reload**

Reloads the animation after a `close` call. Returns the success flag.

**reset**

Resets the animation sequence. This call is necessary either when the image sequence was altered, or when the sequence display restart is needed.

**size**

Returns the width and height of the composite frame

**suspended**

Returns true if a call to the `close` method was made.

**total**

Return the number of frames

**warning**

If an error occured during frame loading, it will be stored in the `warning` property. The animation will stop at the last successfully loaded frame

Only available if the `loadAll` option is off.

**width**

Returns the width of the composite frame

## 11.5 Prima::Image::base64

Hardcoded image files

### Description

Loads and saves images from and to base64-encoded data streams. This allows loading images directly from the source code.

### Synopsis

```
my $icon = Prima::Icon->load_stream(<<~'ICON');
        R0lGODdhIAAgAIAAAAAAAP///ywAAAAAIAAgAIAAAAD///8CT4SPqcvtD6OctNqLcwogcK91nEhq
        3gim2Umm4+W2IBzX0fvl8jTr9SeZiU5E4a1XLHZ4yaal6XwFoSwMVUVzhoZSaQW6ZXjD5LL5jE6r
        DQUAOw==
        ICON

print $icon->save_stream;
```

### API

#### load, load_image BASE64_STRING, %OPTIONS

Decodes BASE64_STRING and tries to load an image from it. Returns image reference(s) on success, or `undef, ERROR_STRING` on failure.

#### load_icon BASE64_STRING, %OPTION

Same as `load_image` but returns a `Prima::Icon` instance.

#### save IMAGE_OR_ICON, %OPTIONS

Saves an image to a datastream and encodes it in base64. Unless the `$OPTIONS{codecID}` or `$image-{extras}->{codecID}>` field is set, tries to deduce the best codec for the job.

Returns the encoded content on success, or `undef, ERROR_STRING` on failure.

## 11.6 Prima::Image::Exif

Manipulate Exif records

### Description

The module allows to parse and create Exif records. The records could be read from JPEG files, and stored in these using the extra appdata hash field.

### Synopsis

```
use Prima qw(Image::Exif);

# load image with extras
my $jpeg = Prima::Image->load($ARGV[0], loadExtras => 1);
die $@ unless $jpeg;

my ( $data, $error ) = Prima::Image::Exif->read_extras($jpeg,
        load_thumbnail => 1,
        tag_as_string  => 1
);

if ( $error eq 'XMP data' && defined $data ) {
        require XML::LibXML;
        my $xml = XML::LibXML->load_xml(string => $data);
        for my $node ($xml->findnodes('//*')) {
                my @p = $node->childNodes;
                next unless @p == 1;
                my $p = $node->nodePath;
                $p =~ s{\/\b[-\w]+\:}{.}g;
                $p =~ s{\.(xmpmeta|rdf)}{}ig;
                $p =~ s{^\.}{};
                print "$p: $p[0]\n";
        }
        ($data, $error) = ({}, undef);
}

die "cannot read exif: $error\n" if defined $error;

for my $k ( sort keys %$data ) {
        my $v = $data->{$k};
        if ( $k eq 'thumbnail' ) {
                if ( ref($v)) {
                        print "thumbnail ", $v->width, 'x', $v->height, "\n";
                } else {
                        print "error loading thumbnail: $v\n";
                }
                next;
        }
        for my $dir ( @$v ) {
                my ( $tag, $name, @data ) = @$dir;
                print "$k.$tag $name @data\n";
        }
}
```

```
# create new image
$jpeg->size(300,300);

# create a thumbnail - not too big as jpeg appdata max length is 64k
my $thumbnail = $jpeg->dup;
delete $thumbnail->{extras};
$thumbnail->size(150,150);

# compile an exif chunk
my $ok;
($ok, $error) = Prima::Image::Exif->write_extras($jpeg,
        thumbnail => $thumbnail,
        gpsinfo   => $data->{gpsinfo},
);
die "cannot create exif data: $error\n" unless $ok;

$jpeg->save('new.jpg') or die $@;
```

## API

### parse $CLASS, $EXIF_STRING

Returns two scalars, a data reference and an error. If there is no data reference, the error is fatal, otherwise a warning (i.e. assumed some data were parsed, but most probabyl not all).

The data is a hash where there are the following keys may be set: image, photo, gpsinfo, thumbnail. These are individual categories containing the exif tags. Each hash value contains an array of tags, except `thumbnail` that contains a raw image data. Each tag is an array in the following format: [ tag, format, @data ] where the tag is a numeric tag value, the format is a type descriptor (such as int8 and ascii), and data is 1 or more scalars containing the data.

The module recognized some common tags that can be accessed via `%Prima::Image::Exif::tags`.

### read_extras $CLASS, $IMAGE, %OPTIONS

Given a loaded Prima image, loads exif data from extras; returns two scalar, a data reference and an error.

Options supported:

#### load_thumbnail

If set, tries to load thumbnail as a Prima image. In this case, replaces the thumbnail raw data with the image loaded, or in case of an error, with an error string

#### tag_as_string

If set, replaces known tag numeric values with their string names

### compile $CLASS, $DATA

Accepts DATA in the format described above, creates an exif string. Returns two scalars, am exif string and an error. If the string is not defined, the error is.

### write_extras $CLASS, $IMAGE, %DATA

Checks if image codec is supported, creates Exif data and saves these in `$IMAGE->{extras}` . Return two scalars, a success flag and an error.

## 11.7 Prima::Image::Loader

Progressive loading and saving for multiframe images

### Description

The toolkit provides functionality for session-based loading and saving of multiframe images so that it is not needed to store all image frames in memory at once. Instead, the `Prima::Image::Loader` and `Prima::Image::Saver` classes provide the API for operating on a single frame at a time.

### Prima::Image::Loader

```
use Prima::Image::Loader;
my $l = Prima::Image::Loader->new($ARGV[0]);
printf "$ARGV[0]: %d frames\n", $l->frames;
while ( !$l->eof ) {
        my ($i,$err) = $l->next;
        die $err unless $i;
        printf "$n: %d x %d\n", $i->size;
}
```

**new FILENAME|FILEHANDLE, %OPTIONS**

Opens a filename or a file handle, tries to deduce if the toolkit can recognize the image, and creates an image loading handler. The `%OPTIONS` are the same as recognized by the **load** entry in the *Prima::Image* section except `map`, `loadAll`, and `profiles`. The other options apply to each frame that will be consequently loaded, but these options could be overridden by supplying parameters to the `next` call.

Returns either a new loader object or `undef` and an error string.

Note that it is possible to supply the `onHeaderReady` and `onDataReady` callbacks in the options, however, note that the first arguments in these callbacks will point to the newly created image, not the loader object.

**close**

Releases the image file handle. The image can be reopened again by calling `reload`.

**current INDEX**

Manages the index of the frame that will be loaded next. When set, requests repositioning of the frame pointer so that the next call to `next` would load the INDEXth image.

**eof**

Returns the boolean flag if the end of the file is reached.

**extras**

Returns the hash of the extra file data as filled by the codec

**frames**

Returns the number of frames in the file

**next %OPTIONS**

Loads the next image frame.

Returns either a newly loaded image or `undef` and an error string.

**reload**

In case an animation file is defective and cannot be loaded in full, the toolkit will not allow to continue the loading session and will close it automatically. If it is desired to work around this limitation, a new session must be opened. The `reload` method does this by reopening the loading session with all the parameters supplied to the initial `new` call. The programmer thus has a chance to record how many successful frames were loaded, and only navigate these after the reload.

**rescue BOOLEAN**

If set, reopens the input stream or file on every new frame. This may help recover broken frames.

**source**

Returns the filename or the file handle passed to the `new` call.

## Prima::Image::Saver

```
my $fn = '1.webp';
open F, ">", $fn or die $!;
my ($s,$err) = Prima::Image::Saver->new(\*F, frames => scalar(@images));
die $err unless $s;
for my $image (@images) {
        my ($ok,$err) = $s->save($image);
        next if $ok;
        unlink $fn;
        die $err;
}
```

**new FILENAME|FILEHANDLE, %OPTIONS**

Opens a filename or a file handle. The `%OPTIONS` are the same as recognized by the **save** entry in the *Prima::Image* section except the `images` option. The other options apply to each frame that will be consequently saved, but these options could also be overridden by supplying parameters to the `save` call.

Returns either a new saver object or `undef` and an error string.

**save %OPTIONS**

Saves the next image frame.

Returns a success boolean flag and an eventual error string

## 11.8 Prima::IniFile

Support of Windows-like initialization files

### Description

The module provides mapping of a text initialization file to a two-level hash structure. The first level is *sections*, which groups the second level hashes, *items*. Sections must have unique keys. The values of the *items* hashes are arrays of text strings. The methods that operate on these arrays are the *get_values* entry, the *set_values* entry, the *add_values* entry, and the *replace_values* entry.

### Synopsis

```
use Prima::IniFile;

my $ini = create Prima::IniFile;
my $ini = create Prima::IniFile FILENAME;
my $ini = create Prima::IniFile FILENAME,
                                default => HASHREF_OR_ARRAYREF;
my $ini = create Prima::IniFile file => FILENAME,
                                default => HASHREF_OR_ARRAYREF;

my @sections = $ini->sections;
my @items = $ini->items(SECTION);
my @items = $ini->items(SECTION, 1);
my @items = $ini->items(SECTION, all => 1);

my $value = $ini-> get_values(SECTION, ITEM);
my @vals = $ini-> get_values(SECTION, ITEM);
my $nvals = $ini-> nvalues(SECTION, ITEM);

$ini-> set_values(SECTION, ITEM, LIST);
$ini-> add_values(SECTION, ITEM, LIST);
$ini-> replace_values(SECTION, ITEM, LIST);

$ini-> write;
$ini-> clean;
$ini-> read( FILENAME);
$ini-> read( FILENAME, default => HASHREF_OR_ARRAYREF);

my $sec = $ini->section(SECTION);
$sec->{ITEM} = VALUE;
my $val = $sec->{ITEM};
delete $sec->{ITEM};
my %everything = %$sec;
%$sec = ();
for ( keys %$sec) { ... }
while ( my ($k,$v) = each %$sec) { ... }
```

### Methods

**add_values SECTION, ITEM, @LIST**

Adds LIST of string values to the ITEM in SECTION.

**clean**

Cleans all internal data in the object, including the name of the file.

**create PROFILE**

Creates an instance of the class. The PROFILE is treated partly as an array, and partly as a hash. If PROFILE consists of a single item, the item is treated as a filename. Otherwise, PROFILE is treated as a hash, where the following keys are allowed:

**file FILENAME**

Selects the name of the file.

**default %VALUES**

Selects the initial values for the file, where VALUES is a two-level hash of sections and items. It is passed to the *read* entry, where it is merged with the file data.

**get_values SECTION, ITEM**

Returns an array of values for ITEM in SECTION. If called in scalar context and there is more than one value, the first value in the list is returned.

**items SECTION [ HINTS ]**

Returns items in SECTION. HINTS parameters are used to tell if a multiple-valued item must be returned as several items of the same name; HINTS can be supplied in the following forms:

items( $section, 1 ) items( $section, all => 1);

**new PROFILE**

Same as the *create* entry.

**nvalues SECTION, ITEM**

Returns the number of values in ITEM in SECTION.

**read FILENAME, %PROFILE**

Flushes the old content and opens a new file. FILENAME is a text string, PROFILE is a two-level hash of default values for the new file. PROFILE is merged with the data from the file, and the latter keeps the precedence. Does not return any success values but warns if any error is occurred.

**replace_values SECTION, ITEM, @VALUES**

Removes all values from ITEM in SECTION and assigns it to the new list of VALUES.

**section SECTION**

Returns a tied hash for SECTION. All its read and write operations are reflected in the caller object which allows the following syntax:

```
my $section = $inifile-> section( 'Sample section');
$section-> {Item1} = 'Value1';
```

which is identical to

```
$inifile-> set_items( 'Sample section', 'Item1', 'Value1');
```

**sections**

Returns an array of section names.

**set_values SECTION, ITEM, @VALUES**

Assigns VALUES to ITEM in SECTION. If the number of new values is equal to or greater than the number of the old, the method is the same as the *replace_values* entry. Otherwise, the values with indices higher than the number of new values are not touched.

**write**

Rewrites the file with the object content. The object keeps an internal modification flag {`changed`}; in case it is `undef`, no actual write is performed.

## 11.9 podview

Graphical pod viewer

### Description

A small GUI browser for POD-formatted files. Accepts either a file path or a perl module name (f.ex. *File::Copy*) as a command line agrument, displays the documentation found.

### SEE ALSO

*perlpod* - the Plain Old Documentation format
the *Prima* section - perl graphic toolkit the viewer is based on
the *Prima::HelpViewer* section - menu commands explained
the **Adding help to your program** entry in the *Prima::tutorial* section - how to add help content

## 11.10 prima-pod2pdf

Convert pod file to a pdf document

### Synopsis

format: prima-pod2pdf [options] input.pod [output.pdf|-]

## 11.11 Prima::StdBitmap

Shared access to the standard bitmaps

### Description

The toolkit provides the *sysimage.gif* file that contains the standard Prima image library and consists of a predefined set of images used by different modules. To provide unified access to the images inside the file, this module's API can be used. Every image is assigned to a `sbmp::` constant that is used as an index for an image loading request. If an image is loaded successfully, the result is cached and the successive requests use the cached image.

The images can be loaded as `Prima::Image` and `Prima::Icon` instances, by two methods, correspondingly `image` and `icon`.

### Synopsis

```
use Prima::StdBitmap;
my $logo = Prima::StdBitmap::icon( sbmp::Logo );
```

### API

#### Methods

**icon INDEX**

Loads the INDEXth image frame and returns a `Prima::Icon` instance.

**image INDEX**

Loads the INDEXth image frame and returns a `Prima::Image` instance.

**load_std_bmp %OPTIONS**

Loads the `index`th image frame from `file` and returns it as either a `Prima::Image` or a `Prima::Icon` instance, depending on the value of the boolean `icon` flag. If the `copy` boolean flag is unset, a cached image can be used. If this flag is set, a cached image is never used and the created image is neither stored in the cache. Since the module's intended use is to provide shared and read-only access to the image library, `copy` set to 1 can be used to return non-shareable images.

The loader automatically scales images if the system dpi suggests so. If layering is supported, the icon scaling will use that as well. To disable these optimizations use the `raw => 1` flag to disable all optimizations, and `argb => 0` to disable producing ARGB icons.

#### Constants

An index value passed to the methods must be one of the `sbmp::` constants:

```
sbmp::Logo
sbmp::CheckBoxChecked
sbmp::CheckBoxCheckedPressed
sbmp::CheckBoxUnchecked
sbmp::CheckBoxUncheckedPressed
sbmp::RadioChecked
sbmp::RadioCheckedPressed
sbmp::RadioUnchecked
sbmp::RadioUncheckedPressed
sbmp::Warning
```

```
sbmp::Information
sbmp::Question
sbmp::OutlineCollapse
sbmp::OutlineExpand
sbmp::Error
sbmp::SysMenu
sbmp::SysMenuPressed
sbmp::Max
sbmp::MaxPressed
sbmp::Min
sbmp::MinPressed
sbmp::Restore
sbmp::RestorePressed
sbmp::Close
sbmp::ClosePressed
sbmp::Hide
sbmp::HidePressed
sbmp::DriveUnknown
sbmp::DriveFloppy
sbmp::DriveHDD
sbmp::DriveNetwork
sbmp::DriveCDROM
sbmp::DriveMemory
sbmp::GlyphOK
sbmp::GlyphCancel
sbmp::SFolderOpened
sbmp::SFolderClosed
sbmp::Last
```

### Scalars

The `$sysimage` scalar is initialized to the file name to be used as a source of standard images. It is possible to alter this scalar at run-time, which causes all subsequent image frame requests to be redirected to the new file.

### Scaling and ARGB-shading

The loading routine scales and visually enhances the images automatically according to the system settings that are reported by the `Prima::Application` class. It is therefore advisable to load images after the Application object is created.

## 11.12 Prima::Stress

Stress test module

### Description

The module is intended for use in test purposes, to check the functionality of a program or a module under particular conditions that might be overlooked during the design. Currently, the stress factors implemented are the default font size, default scrollbar sizes, and the UI-scaling factor, which are set to different random values every time the module is invoked.

To use the module it is enough to include a typical

```
use Prima::Stress;
```

code, or, if the program is invoked by calling perl, by using the

```
perl -MPrima::Stress program
```

syntax. The module does not provide any methods, however, one may address individual aspects of the UI defaults.

### API

#### Font size

```
use Prima::Stress q(fs=18);
perl -MPrima::Stress=fs=18 program
```

This syntax changes the default font size to 18 points.

#### Display resolution

```
use Prima::Stress q(dpi=192);
perl -MPrima::Stress=dpi=192 program
```

This syntax changes the display resolution to 192 pixels per inch.

#### Scrollbar sizes

```
use Prima::Stress q(src=40);
perl -MPrima::Stress=src=40 program
```

This syntax changes the default width of vertical scrollbars, and the default height of horizontal scrollbars to 40 pixels

## 11.13 Prima::Themes

Object themes management

### Description

Provides a layer for theme registration in Prima. Themes are loosely grouped alternations of default class properties and behaviors, by default stored in the `Prima/themes` subdirectory. The theme realization is implemented as interception of the object profile during its creation inside `::profile_add`. Various themes apply various alterations, one way only - once an object is applied to a theme, it cannot be either changed or revoked thereafter.

Theme configuration can be stored in an RC file, *~/.prima/themes*, and is loaded automatically unless `$Prima::Themes::load_rc_file` is explicitly set to `0` before loading the `Prima::Themes` module. In effect, any Prima application not aware of themes can be coupled with themes in the RC file by the following:

```
perl -MPrima::Themes program
```

The `Prima::Themes` namespace provides API for the theme registration and execution. `Prima::Themes::Proxy` is a class for overriding certain methods, for internal realization of a theme.

For the interactive theme selection see the *examples/theme.pl* sample program.

### Synopsis

```
# register a theme file
use Prima::Themes qw(color);
# or
use Prima::Themes; load('color');
# list registered themes
print Prima::Themes::list;

# install a theme
Prima::Themes::install('cyan');
# list installed themes
print Prima::Themes::list_active;
# create an object with another theme while 'cyan' is active
Class->new( theme => 'yellow');
# remove a theme
Prima::Themes::uninstall('cyan');
```

### Prima::Themes

#### load @THEME_MODULES

Loads THEME_MODULES from files via the `use` clause, dies on error. Can be used instead of the explicit `use` call.

A loaded theme file may register one or more themes.

#### register $FILE, $THEME, $MATCH, $CALLBACK, $INSTALLER

Registers a previously loaded theme. $THEME is a unique string identifier. $MATCH is an array of pairs where the first item is a class name, and the second is an arbitrary scalar parameter. When a new object is created, its class is matched via `isa` to each given class name, and if matched, the $CALLBACK routine is called with the following parameters: object, default profile, user profile, and second item of the matched pair.

If the $CALLBACK is `undef`, the default the *merger* entry routine is called, which treats the second items of the pairs as hashes of the same format as the default and user profiles.

The theme is inactive until `install` is called. If the $INSTALLER subroutine is passed, it is called during install and uninstall with two parameters, the name of the theme and the boolean install/uninstall flag. When the install flag is 1, the theme is about to be installed; the subroutine is expected to return a boolean success flag. Otherwise, the subroutine's return value is not used.

$FILE is used to indicate the file in which the theme is stored.

**deregister $THEME**

Un-registers $THEME.

**install @THEMES**

Installs previously loaded and registered THEMES; the installed themes will be applied to match new objects.

**uninstall @THEMES**

Uninstalls loaded THEMES.

**list**

Returns the list of registered themes.

**list_active**

Returns the list of installed themes.

**loaded $THEME**

Return 1 if $THEME is registered, 0 otherwise.

**active $THEME**

Return 1 if $THEME is installed, 0 otherwise.

**select @THEMES**

Uninstalls all currently installed themes, and installs THEMES instead.

**merger $OBJECT, $PROFILE_DEFAULT, $PROFILE_USER, $PROFILE_THEME**

Default profile merging routine, merges $PROFILE_THEME into $PROFILE_USER by the keys from $PROFILE_DEFAULT.

**load_rc [ $INSTALL = 1 ]**

Reads the *~/.prima/themes* file and loads the listed modules. If $INSTALL = 1, installs the themes from the RC file.

**save_rc**

Writes configuration of currently installed themes into the RC file, and returns the success flag. If the success flag is 0, `$!` contains the error.

## Prima::Themes::Proxy

An instance of `Prima::Themes::Proxy`, created as

Prima::Themes::Proxy-> new( $OBJECT)

that would return a new non-functional wrapper for any Perl object $OBJECT. All methods of the $OBJECT, except `AUTOLOAD`, `DESTROY`, and `new`, are forwarded to the $OBJECT itself transparently. The class can be used, for example, to deny all changes to `lineWidth` inside the object's painting routine:

```
package ConstLineWidth;
use base 'Prima::Themes::Proxy';

sub lineWidth { 1 } # line width is always 1 now!

Prima::Themes::register( '~/lib/constlinewidth.pm', 'constlinewidth',
        [ 'Prima::Widget' => {
                onPaint => sub {
                        my ( $object, $canvas) = @_;
                        $object-> on_paint( ConstLineWidth-> new( $canvas));
                },
        } ]
);
```

## Files

*~/.prima/themes*

## 11.14 Prima::Tie

Tie widget properties to scalars and arrays

### Description

Prima::Tie contains two abstract classes `Prima::Tie::Array` and `Prima::Tie::Scalar` which tie an array or a scalar to a widget's arbitrary array or scalar property. Also, it contains classes `Prima::Tie::items`, `Prima::Tie::text`, and `Prima::Tie::value`, which tie a variable to a widget's *items*, *text*, and *value* properties respectively.

### Synopsis

```
use Prima::Tie;

tie @items, 'Prima::Tie::items', $widget;

tie @some_property, 'Prima::Tie::Array', $widget, 'some_property';

tie $text, 'Prima::Tie::text', $widget;

tie $some_property, 'Prima::Tie::Scalar', $widget, 'some_property';
```

### Usage

These classes provide immediate access to a widget's array and scalar properties, in particular to popular properties *items* and *text*. It is considerably simpler to say

```
splice(@items,3,1,'new item');
```

than to say

```
my @i = @{$widget->items};
splice(@i,3,1,'new item');
$widget->items(\@i);
```

That way, you can work directly with the text or items. Furthermore, if the only reason you keep an object around after creation is to access its text or items, you no longer need to do so:

```
tie @some_array, 'Prima::Tie::items', Prima::ListBox->new(@args);
```

As opposed to:

```
my $widget = Prima::ListBox->new(@args);
tie @some_array, 'Prima::Tie::items', $widget;
```

`Prima::Tie::items` requires the `::items` property to be available on the widget. Also, it takes advantage of additional `get_items`, `add_items`, and the like methods if available.

#### Prima::Tie::items

The class is applicable to `Prima::ListViewer`, `Prima::ListBox`, `Prima::Widget::Header`, and their descendants, and in a limited fashion to `Prima::OutlineViewer` and its descendants `Prima::StringOutline` and `Prima::Outline`.

#### Prima::Tie::text

The class is applicable to any widget.

**Prima::Tie::value**

The class is applicable to `Prima::GroupBox`, `Prima::Dialog::ColorDialog`, `Prima::SpinEdit`, `Prima::Gauge`, `Prima::Slider`, `Prima::CircularSlider`, and `Prima::ScrollBar`.

## 11.15 Prima::types

Builtin types

### Description

This document describes the auxiliary second-citizen classes that are used as results of Prima methods and accepted as inputs. Objects that instantiate from these classes are usually never created manually. The names of some of these classes begin with a lower-case letter, to underscore their auxiliary nature.

### Prima::array

An overloaded C array that can be used transparently as a normal perl array. The array can only hold numbers. The reason the `Prima::array` class exists is so Prima methods won't need to do expensive conversions between a perl array of scalars to a C array of integers or floats.

**new LETTER = [idsS], BUF = undef**

Creates a new C array with the type of either int, double, short, or unsigned short. There are also methods `new_int`, `new_double`, `new_short`, and `new_ushort` that do the same.

BUF, a normal perl string, can be used to initialize the array, if any (and can be pre-populated with pack()). Otherwise, an array is created empty.

**is_array SCALAR**

Checks whether the SCALAR is a `Prima::array` object.

**substr OFFSET, LENGTH, REPLACEMENT**

Emulates perl's `substr` except operates not on characters but on the individual numeric entries of the array. Returns a new `Prima::array` object.

**append ARRAY**

Assuming that two arrays have the same type, appends the ARRAY's contents to its content.

**clone**

Clones the array.

### Prima::matrix

An array of 6 doubles with some helper methods attached.

**A,B,C,D,X,Y**

Named accessory properties for the 6 members. The members can just as well be accessed directly with the array syntax.

**clone**

Clones the matrix object

**identity**

Sets the matrix to `Prima::matrix::identity`, or (1,0,0,1,0,0)

**inverse_transform @POINTS | $POINTS_ARRAY**

Applies the inverse matrix tranformations to an array or an arrayref of points and returns the result matrix in the same format (i e array for array, ref for ref).

**new [ @ARRAY ]**

Creates a new object and optionally initializes it with @ARRAY

**multiply MATRIX**

Multiplies the matrices and stores the result

**rotate ANGLE**

Rotates the matrix

**scale MX,MY**

Scales the matrix

**shear DX,DY**

Shears the matrix

**set @ARRAY**

Assigns all the 6 members at once

**translate DX,DY**

Translates the matrix

**transform @POINTS | $POINTS_ARRAY**

Applies matrix tranformations to an array or arrayref of points and returns the result matrix in the same format (i e array for array, ref for ref).

See also: the *Prima::Matrix* section

## Prima::Matrix

Same as `Prima::matrix` but explicitly binds to drawable objects so that all changes to the matrix object are immediately reflected in the drawable.

Features all the methods available to `Prima::matrix` (except `apply`), plus the ones described below.

**new CANVAS**

Creates a new matrix object instance

**canvas DRAWABLE**

Accesses the associated drawable object

**get**

Returns the current matrix

**reset**

Sets the matrix to `Prima::matrix::identity`, or (1,0,0,1,0,0)

**save, restore**

Saves and restores the matrix content in the internal stack

## Prima::rect

Represents a rectangular object either as a *rectangle* ( X1,Y1,X2,Y2 ) or a *box* ( X,Y,WIDTH,HEIGHT ).

**box**

Returns X, Y, WIDTH, HEIGHT

**clone**

Clones the object

**enlarge N**

Enlarges the rectangle by N

**inclusive**

Rectangle itself is agnostic of its 2D presentation, but assuming the coordinates are inclusive-exclusive, `inclusive` returns X1,Y1,X2,Y2 as the inclusive-inclusive rectangle.

**intersect RECT**

Intersects with the RECT rectangle and stores the result

**is_empty**

Returns true if the rectangle width and height are zero

**is_equal RECT**

Returns true if both rectangles are equal

**new () | (WIDTH,HEIGHT) | (X1,Y1,X2,Y2)**

Creates a new object assuming the rectangle syntax

**new_box X,Y,WIDTH,HEIGHT**

Creates new object assuming the box syntax

**origin**

Returns X and Y

**shrink N**

Shrinks the rectangle by N

**size**

Returns the WIDTH and HEIGHT of the rectangle

**union RECT**

Joins the rectangle with the RECT rectangle and stores the result

## 11.16 Prima::Utils

Miscellaneous routines

### Description

The module contains miscellaneous helper routines

### API

**alarm $TIMEOUT, $SUB, @PARAMS**

Calls SUB with PARAMS after TIMEOUT milliseconds. Returns 0 on failure, and the active timer on success. The timer can be stopped to disarm the alarm.

**beep [ FLAGS = mb::Error ]**

Invokes the system-depended sound and/or visual bell, corresponding to one of the following constants:

```
mb::Error
mb::Warning
mb::Information
mb::Question
```

**get_gui**

Returns one of the `gui::XXX` constants that report the graphic user interface used in the system:

```
gui::Default
gui::Windows
gui::XLib
gui::GTK
```

**get_os**

Returns one of the `apc::XXX` constants that report the system platform. Currently, the list of the supported platforms is:

```
apc::Win32
apc::Unix
```

**find_image PATH**

Converts PATH from a perl module notation into a file path and searches for the file in the `@INC` paths set. If the file is found, its full filename is returned; otherwise `undef` is returned.

**last_error**

Returns last system error, if any

**nearest_i NUMBERS**

Performs `floor($_ + .5)` operation over NUMBERS which can be an array or an arrayref. Returns converted integers in either an array or an arrayref form, depending on the calling syntax.

**nearest_d NUMBERS**

Performs `floor($_ * 1e15 + .5) / 1e15` operation over NUMBERS which can be an array or an arrayref. Returns converted NVs in either an array or an arrayref form, depending on the calling syntax. Used to protect against perl configurations that calculate `sin`, `cos` etc with only 15 significant digits in the mantissa. This function prevents the accumulation of error in these configurations.

**path [ FILE ]**

If called with no parameters, returns the path to a directory, usually *~/.prima*, that can be used to store the user settings of a toolkit module or a program. If FILE is specified, appends it to the path and returns the full file name. In the latter case, the path is automatically created by `File::Path::mkpath` unless it already exists.

**post $SUB, @PARAMS**

Postpones a call to SUB with PARAMS until the next event loop tick.

**query_drives_map [ FIRST_DRIVE = "A:" ]**

Returns an anonymous array to drive letters used by the system. FIRST_DRIVE can be set to another value to start enumeration from. Win32 can probe removable drives there, so to increase the responsiveness of the function it might be reasonable to call it with FIRST_DRIVE set to `C:` .

If the system supports no drive letters, an empty array reference is returned ( unix ).

**query_drive_type DRIVE**

Returns one of the `dt::XXX` constants that describe the type of a drive, where DRIVE is a 1-character string. If there is no such drive, or the system supports no drive letters ( unix ), `dt::None` is returned.

```
dt::None
dt::Unknown
dt::Floppy
dt::HDD
dt::Network
dt::CDROM
dt::Memory
```

**sleep SECONDS**

Same as perl's native `sleep` (i.e. `CORE::sleep`) but with the event loop running. Note that the argument it takes is seconds, for the sake of compatibility, while the rest of the toolkit operates in milliseconds.

**username**

Returns the login name of the user. Sometimes is preferred to the perl-provided `getlogin` ( see getlogin in *perlfunc* ) .

**xcolor COLOR**

Accepts COLOR string in one of the three formats:

```
#rgb
#rrggbb
#rrrgggbbb
```

and returns a 24-bit RGB integer value

**wait CONDITION [, TIMEOUT ]**

Waits for a condition for max TIMEOUT milliseconds, or forever if TIMEOUT is undefined.

Returns undef on failure, 0 on TIMEOUT, 1 on a successful CONDITION.

CONDITION is either a scalar reference, or a sub to be polled, where their values are treated as 0 as a signal to continue the waiting, and 1 as a stop signal.

## Unicode-aware filesystem functions

Since perl's win32 unicode support for files is unexistent, Prima has its own parallel set of functions mimicking native functions, ie `open`, `chdir` etc. This means that files with names that cannot be converted to ANSI (ie user-preferred) codepage are not visible in perl, but the functions below mitigate that problem.

The following fine points need to be understood before using these functions:

- Prima makes a distinction between whether scalars have their utf8 bit set or not throughout the whole toolkit. For example, text output in both unix and windows is different depending on the bit, treating non-utf8-bit text as locale-specific, and utf8-bit text as unicode. The same model is applied to file names.
- Perl implementation for native Win32 creates virtual environments for each thread and may keep more than one instance of the current directory, environment variables, etc. This means that under Win32, calling `Prima::Utils::chdir` will NOT automatically make `CORE::chdir` assume that value, even if the path is convertible to ANSI. Keep that in mind when mixing Prima and core functions. To add more confusion, under the unix, these two chdir calls are identical when the path is fully convertible.
- Under unix, reading entries from the environment or the file system is opportunistic: if a text string (file name, environment entry) is a valid utf8 string, then it is treated and reported as one. Mostly because the .UTF-8 locales are default and found everywhere. Note that Prima ignores `$ENV{LANG}` here. This is a bit problematic on Perls under 5.22 as these don't provide > means to check for the utf8 string validity, so every string will be slapped > a utf8 bit on here -- beware.
- Setting environment variables may or may not sync with `%ENV` , depending on how perl is built. Also, `%ENV` will warn when trying to set scalars with utf-8 bit there.

**access PATH, MODE**

Same as `POSIX::access`.

**chdir DIR**

Same as `CORE::chdir` but disregards the thread-local environment on Win32.

**chmod PATH, MODE**

Same as `CORE::chmod`

**closedir, readdir, rewinddir, seekdir, telldir DIRHANDLE**

Mimic homonymous perl functions

**getcwd**

Same as `Cwd::getcwd`

**getdir PATH**

Reads the content of the PATH directory and and returns array of string pairs where the first item is a file name and the second is a file type.

The file type is a string, one of the following:

```
"fifo" - named pipe
"chr"  - character special file
"dir"  - directory
"blk"  - block special file
"reg"  - regular file
"lnk"  - symbolic link
"sock" - socket
"wht"  - whiteout
```

This function was implemented for faster directory reading, to avoid successive calls of `stat` for every file.

Also, getdir is consistently inclined to treat filenames in utf8, disregarding both perl unicode settings and the locale.

**getenv NAME**

Reads directly from environment, possibly bypassing `%ENV` , and disregarding thread-local environment on Win32.

**link OLDNAME, NEWNAME**

Same as `CORE::link`.

**local2sv TEXT**

Converts 8-bit text into either 8-bit non-utf8-bit or unicode utf8-bit string. May return undef on memory allocation failure.

**mkdir DIR, [ MODE = 0666 ]**

Same as `CORE::mkdir`.

**open_file PATH, FLAGS**

Same as `POSIX::open`

**open_dir PATH**

Returns directory handle to be used on `readdir`, `closedir`, `rewinddir`, `telldir`, `seekdir`.

**rename OLDNAME, NEWNAME**

Same as `CORE::rename`

**rmdir PATH**

Same as `CORE::rmdir`

**setenv NAME, VAL**

Directly sets environment variable, possibly bypassing `%ENV` , depending on how perl is built. Also disregards the thread-local environment on Win32.

Note that effective synchronization between this call and `%ENV` is not always possible, since Win32 perl implementation simply does not allow that. One is advised to assign to `%ENV` manually, but only if both NAME and VAL don't have their utf8 bit set, otherwise perl will warn about wide characters.

**stat PATH**

Same as `CORE::stat`, except on systems that provide sub-second time resolution, in which case returns the atime/mtime/ctime entries as floats, the same as `Time::HiRes::stat` does.

**sv2local TEXT, FAIL_IF_CANNOT = 1**

Converts either 8-bit non-utf8-bit or unicode utf8-bit string into a local encoding. May return undef on memory allocation failure, or if TEXT contains unconvertible characters when FAIL_IF_CANNOT = 1

**unlink PATH**

Same as `CORE::unlink`.

**utime ATIME, MTIME, PATH**

Same as `CORE::utime`, except on systems that provide sub-second time resolution, in which case returns the atime/mtime/ctime entries as floats, the same as `Time::HiRes::utime` does.

# 12 System-specific modules and documentation

## 12.1 Prima::gp-problems

Problems, questionable or intricate topics in 2-D Graphics

### Introduction

One of the most important goals of the Prima project is the portability between different operating systems. Independently to efforts in keeping Prima's internal code that behaves more or less identically on different platforms, it is always possible to write non-portable and platform-dependent code. Here are some guidelines and suggestions for 2-D graphics programming.

### Minimal display capabilities

A compliant display is expected to have a minimal set of capabilities, that a programmer can rely upon. The following items are always supported by Prima:

**Minimal capabilities**

Distinct black and white colors

At least one monospaced font

Solid fill

rop::Copy and rop::NoOper

**Plotting primitives**

SetPixel,GetPixel

Line,PolyLine

Rectangle

FillPoly

TextOut

PutImage,GetImage

**Information services**

GetTextWidth,GetFontMetrics,GetCharacterABCWidths

GetImageBitsLayout

### Properties

color

backColor

rop

rop2

fillPattern

fillMode

textOpaque

clipRect

All these properties must be present, however, it is not required for them to be changeable. Even if an underlying platform-specific code can only support one mode for a property, it has to follow all obligations for the mode. For example, if the platform supports full functionality for black color but limited functionality for the other colors, the wrapping code should not allow the color property to be writable.

## Inevident issues

### Colors

#### Black and white colors on paletted displays

Since paletted displays use indexed color representation, the 'black' and 'white' indices are not always 0 and 2^n-1, so if one uses the indexes for the actual black and white colors in the palette, the result of raster image operations may look garbled (X11). Win32 protects itself from this condition by forcing white to be the last color in the system palette.

Example: if the white color on the 8-bit display occupies palette index 15 then the desired masking effect wouldn't work for *xoring* transparent areas with `cl::White`.

Workaround: Use two special color constants `cl::Clear` and `cl::Set`, that represent all zeros and all ones values for bit-sensitive raster operations.

#### Black might be not 0, and white not 0xffffff

This obscure issue happens mostly on the 15- and 16-bit pixel displays. Internal color representation for the white color on a 15-color display ( assuming R,G and B are 5-bit fields) is

```
11111000 11111000 11111000
--R----- --G----- --B-----
```

that equals 0xf8f8f8.

A bit of advice: do not check for 'blackness' and 'whiteness' merely by comparing to the 0x000000 or 0xffffff constant.

### Filled shapes

### Dithering

If a non-solid pattern is selected and the background and/or foreground color cannot be drawn as a solid, the correct rendering may require 3 or even 4 colors. Some rendering engines (Win9X) fail to produce correct results.

#### Pattern offset

If a widget contains a pattern-filled shape, its picture will always be garbled after scrolling because it is impossible to provide an algorithm for a correct rendering without prior knowledge of the widget's nature. (All)

Workaround: Do not use patterned backgrounds or use `fillPatternOffset` property. Since the same effect is visible on dithered backgrounds, check if the color used is pure.

### Lines

#### Dithering

Dithering might be not used for line plotting. (Win9X)

### Fonts

#### Font metric inconsistency

A font is loaded by request with one size but claims another afterward (X11).

Impact: system-dependent font description may not match Prima's.

An advice: do not try to deduce Prima font metrics from system-dependent ones and vice versa.

#### Transparent plotting

No internal function for drawing transparent bitmaps (to implement text plotting). Therefore, if a font emulation is desired, special ROPs cannot be reproduced. (Win9X, WinNT)

Impact: font emulation is laborious, primarily because the glyphs have to be plotted by consequential *and*-ing and *xor*-ing a bitmap. A full spectrum of the raster operations cannot be achieved with this approach.

#### Text background

If a text is drawn with a non-CopyPut raster operation, the text background is not expected to be mixed with glyphs - however, this is too hard to implement uniformly, so results may differ for different platforms.

Text background may be only drawn with pure ( non-dithered ) color (win32) - but this is (arguably) a more correct behavior.

A bit of advice: Do not use ::rop2 and text background for special effects

#### Internal platform features

Font change notification is not provided. (X11)

Raster fonts cannot be synthesized (partly X11)

### Raster operations ( ROPs)

The background raster operations are not supported (X11,Win9X,WinNT. Not all ROPs can be emulated for certain primitives, like fonts, complex shapes, and patterned shapes.

It is yet unclear which primitives have to support ROPs, - like FloodFill and SetPixel. The behavior of the current implementation is that they do not.

### Palettes

#### Static palettes

Some displays are unable to change their hardware palette, so detecting an 8- or 4-bit display doesn't automatically mean that the palette is writable.(X11)

#### Widget::palette

The `Widget::palette` property is used for an explicit declaration of extra colors needed by a widget. The request might be satisfied in different ways, or might not be at all. It is advisable not to rely on platform behavior for the palette operations.

#### Dynamic palette change

It is possible (usually on 8-bit displays) for a display to change asynchronously its hardware palette to process different color requests. All platforms behave differently.

Win9X/WinNT - only one top-level window at a time and its direct children ( not ::clipOwner(0)) can benefit from using the Widget::palette property. The system palette is switched every time as different windows move to the front.

X11 - Any application can easily ruin the system color table. Since this behavior is such by design, no workaround can be applied here.

### Bitmaps

#### Invalid scaling

Bitmap scaling is invalid (Win9X) or not supported (X11 without XRender). A common mistake is to not take into account the fractional pixels that appear when the scaling factor is more than 1

Workaround: described in the **zoom** entry in the *Prima::ImageViewer* section .

#### Large scale factors

Request for drawing a bitmap might fail if a large scaling factor is selected. (Win9X,WinNT). This is probably because these platforms scale the bitmap into memory before the plotting takes place.

### Layering

On win32, layered widgets with pixels assigned zero alpha component, will not receive mouse events.

## Platform-specific peculiarities

### Windows 9X

The number of GDI objects can not exceed some unknown threshold - experiments show that 128 objects is a number that is safe enough.

The color cursor creation routine is broken.

Filled shapes are broken.

### X11

No bitmap scaling (if compiled without XRender)

No font rotation (if compiled without Xft)

No GetPixel, FloodFill ( along with some other primitives)

White is not 2^n-1 on n-bit paletted displays (tested on XFree86).

Filled shapes are broken.

Color bitmaps cannot be drawn onto mono bitmaps.

## Implementation notes

### Win32

The plotting speed of DeviceBitmaps is somewhat less on 8-bit displays than when plotting Images and Icons. It is because DeviceBitmaps are bound to their original palette, so putting a DeviceBitmap on a drawable with a different palette uses inefficient algorithms to provide correct results.

### X11

If an image was first drawn on a paletted Drawable, then drawing it on another paletted Drawable would reduce the image to 8 safe colors.

This is by design and is so because the image has a special cache in the display pixel format. Refreshing the cache on every PutImage call is very inefficient (although technically possible). It is planned to fix the problem by checking the palette difference for every PutImage invocation. NB - the effect is seen on dynamic color displays only.

## 12.2 Prima::X11

Usage guide for the X11 environment

### Description

This document describes subtle topics one must be aware of when programming or using Prima programs under X11. The document covers various aspects of the toolkit and its implementation details with the guidelines of the expected use. Also, some of the X11 programming techniques are visited.

### Basic command-line switches

`--help`

Prints the available command-line arguments and exits

`--display`

Sets the X display address in the Xlib notation. If not set, the standard Xlib ( `XOpenDisplay(null)` ) behavior applies.

Example:

```
--display=:0.1
```

`--visual`

Sets the X visual to be used by default. Example:

```
--visual=0x23
```

`--sync`

Turn on the X synchronization

`--bg`, `--fg`

Set the default background and foreground colors. Example:

```
--bg=BlanchedAlmond
```

`--font`

Sets the default font in either XLFD or Fontconfig format. Examples:

```
--font=serif
--font=Arial-16:bold
--font='adobe-helvetica-medium-r-*-*--*-120-*-*-*-*-*-*'
```

`--no-x11`

Runs Prima without the X11 display initialized. This switch can be used for programs that use only the OS-independent parts of Prima, such as the image subsystem or the PDF generator, in environments where X is not present, for example, from a CGI script. Any attempt to create an instance of the `Prima::Application` class or otherwise access the X-depended code under such conditions causes the program to abort.

There are alternatives to the command switch. First, there is module `Prima::noX11` for the same purpose but that is more convenient to use as the

```
perl -MPrima::noX11
```

construct. Second, there is the technique to continue execution even if the connection to the X server fails:

```
use Prima::noX11;
use Prima;

my $error = Prima::XOpenDisplay();
if ( defined $error) {
     print "not connected to display: $error\n";
} else {
     print "connected to the X display\n";
}
```

The the *Prima::noX11* section module exports the single function `XOpenDisplay` into the `Prima` namespace, to connect to the X display explicitly. The display to be connected to is the `$ENV{DISPLAY}` unless stated otherwise on the command line ( with the --display option) or with a parameter to the `XOpenDisplay` function.

This technique may be useful to programs that use Prima imaging functionality and may or may not use the windowing capabilities.

## The X11 resources database

X11 provides XRDB, the X resource database, a named list of arbitrary string values stored on the X server. Each key is a combination of names and classes of widgets in the text format. The key is constructed so that the leftmost substring ( the name or the class ) corresponds to the top-level item in the hierarchy, usually the application name or class. Although the XRDB can be also written via the native X API, it is rarely done by applications. Instead, the user creates a file usually named .Xdefaults which contains the database in the text form.

The format of the .Xdefaults file directly reflects the XRDB capabilities, one of the most important of which is globbing, manifested via the * ( star ) character. With the use of the globbing, the user can set up a property value that corresponds to multiple targets:

```
*.ListBox.backColor: yellow
```

The string above means that all widgets of the ListBox class must have a yellow background.

The application itself is responsible for parsing the strings and querying the XRDB. Also, both class and widget names, as well as the database values are fully defined in terms of the application. There are some guidelines though, for example, the colors and fonts are best described in the terms native to the X server. Also, classes and names are distinguished by the case: classes must begin with the uppercase letter. Finally, not every character can be stored in the XRDB database (space, for example, cannot be) and therefore the XRDB API automatically converts these to the _ ( underscore ) characters.

Prima defines its own set of resources, divided into two parts: general toolkit settings and per-widget settings. The general settings functionality is partially overlapping with the command-line arguments. The per-widget settings are the fonts and colors that can be defined for each Prima widget.

All of the general settings apply to the top-level item of the widget hierarchy, named after the application, and the `Prima` class. Some of these though needed to be initialized before the application instance itself is created, so these can be accessed via the `Prima` class only, for example, `Prima.Visual`. Some, on the contrary, may occasionally overlap with the per-widget syntax. In particular, one must be wary not to write

```
Prima*font: myfont
```

instead of

```
Prima.font: myfont
```

The latter syntax is a general setting and changes the default Prima font only. The former is a per-widget assignment, and explicitly sets the font to **all** Prima widgets, effectively ruining the toolkit font inheritance scheme. The same is valid for an even more powerful

```
*font: myfont
```

record.

The allowed per-widget settings are the color and font settings only ( see the corresponding sections ). It is an arguably useful feature to map all the widget properties onto XRDB, but Prima does not implement this, primarily because no one asked for it, and also because this creates unnecessary latency when the enumeration of all possible widget properties takes place for every widget.

All of the global settings' classes and names are identical except for their first letter. For example, to set the `Submenudelay` value, one can do it either by the

```
Prima.Submenudelay: 10
```

or by the

```
Prima.submenudelay: 10
```

syntax. Despite that, these calls are different, in a way that one reaches for the whole class and another for the name, for the majority of these properties it does not matter. To avoid confusion all class names are camelcase while the property names are lowercase.

## Fonts

### Default fonts

Prima::Application defines the set of `get_default_XXX_font` functions, where each returns the font that is predefined by the system or by the user through the system settings, to be displayed correspondingly in menus, messages, window captions, and all other widgets. While in f ex Win32 these are indeed the configurable user options, the raw X11 protocol doesn't define any. If the toolkit is compiled with the GTK, then the default fonts can be read from the GTK settings. Nevertheless, as the high-level code relies on these, the corresponding resources are defined. These are:

- font - Application::get_default_font
- caption_font - Application::get_caption_font. Used in `Prima::MDI`.
- menu_font - Widget::get_default_menu_font. The default font for the pull-down and pop-up menus.
- msg_font - Application::get_message_font. Used in `Prima::MsgBox`.
- widget_font - Widget::get_default_font.

All of the global font properties can only be set via the `Prima` class, no application name is recognized. Also, these properties are identical to `--font`, `--menu-font`, `--caption-font`, `--msg-font`, and `--widget-font` command-line arguments. The per-widget properties are `font` and `popupFont`, of class `Font`, settable via XRDB only:

```
Prima*Dialog.font: my-fancy-dialog-font
Prima.FontDialog.font: some-conservative-font
```

By default, Prima font is 12.helvetica .

### X core fonts

The values of the font entries are standard XLFD strings, formatted with the default `*-*-*-*-*-*-*-*-*-*-*-*-*-*` pattern, where each star character can be replaced by a particular font property, such as name, size, charset, and so on. To interactively select an appropriate font, use the standard `xfontsel` program from the Xorg distribution.

Note, that the encoding part of the font is recommended to be left unspecified, otherwise it may clash with the LANG environment variable that is used by the Prima font subsystem to determine which font to select when no encoding is given. This advice, though, is correct only when both the LANG and encoding part of the desired font match. To force a particular font encoding, the property `Prima.font` must contain one.

Alternatively, and/or to reduce X font traffic, one may set the `IgnoreEncodings.ignoreEncodings` property, which is a semicolon-separated list of encodings Prima must not use. This feature has limited usability when for example fonts in the Asian encodings result in large font requests. Another drastic measure to decrease font traffic is the boolean property `Noscaledfonts.noscaledfonts`, which, if set to 1, restricts the choice of fonts to the non-scalable fonts only.

### Xft fonts

Prima can compile with the Xft library, which contrary to core X font API, can make use of the client-side fonts. Plus, the Xft library offers appealing features such as font antialiasing, unicode, and arguably a better font syntax. The Xft font syntax is inherited from the `fontconfig` library and is to be consulted from `man fonts-conf`. For example:

```
Palatino-12
```

A font with the name `Palatino` and a size of 12 points.

```
Arial-10:BI
```

A font with the name `Arial`, size of 10 points, bold, and italic. The `fontconfig` syntax allows more than that, for example, arbitrary matrix transformations, but Prima can make use only of the font name, size, and style flags.

`--no-xft`

The `--no-xft` command-line argument, and the corresponding boolean `UseXFT.usexft` XRDB property can be used to disable the use of the Xft library.

`--no-core-fonts`

Disables all X11 core fonts, except the `fixed` font. The `fixed` font is selected for the same reasons that the X server is designed to provide at least one font, which usually is `fixed`.

It is valid to combine `--no-core-fonts` and `--no-xft`. Moreover, adding `--noscaled` to these gives Prima programs the very classic X look.

`--font-priority`

Can be set to either `xft` or `core`, to select the font provider mechanism to match unknown or incompletely specified fonts against.

Default value: `xft` ( if compiled in ), `core` otherwise.

`--no-aa`

If set, turns off the Xft font antialiasing.

## Colors

### XRDB conventions

The X11 is traditionally shipped with the color names database, usually a text file named *rgb.txt*. Check your X manual where exactly this file resides and what is its format. The idea behind it is that users can benefit from portable literal color names, with color values transparently adjustable to display capabilities. Thus, it is customary to write

```
color: green
```

for many applications, and these in turn call the `XParseColor` function to convert strings into RGB values.

Prima can also support this functionality. Each widget can assign eight color properties: `color`, `hiliteBackColor`, `disabledColor`, `dark3DColor backColor`, `hiliteColor`, `disabledBackColor`, `light3DColor` by their name:

```
Prima.backColor: #cccccc
```

Additionally, the following command-line arguments allow overriding the default values for these properties:

- `--fg` - color
- `--bg` - backColor
- `--hilite-fg` - hiliteColor
- `--hilite-bg` - hiliteBackColor
- `--disabled-fg` - disabledColor
- `--disabled-bg` - disabledBackColor
- `--light` - light3DColor
- `--dark` - dark3DColor

### Visuals

The colors in the X11 protocol require the pixel values to be explicitly defined. A pixel value is a 32-bit unsigned integer that encodes color in the display format. There are two different color coding schemes - the direct color and the indexed color. The direct color-coded pixel value can unambiguously be converted into an RGB value without any additional information. The indexed-color scheme represents the pixel value as an index in a palette that resides on the X server. The X11 display can contain more than one palette, and allow ( or disallow ) modification of the palette color cells depending on the visual the palette is attached to.

A *visual* is an X server resource with a specific representation of the color coding scheme, color bit depth, and modifiability of the palette. The X server can ( and usually does ) provide more than one visual, as well as different pixel bit depths. There are six classes of visuals in the X11 paradigm. In each, Prima behaves differently, also depending on the display bit depth available. In particular, the color dithering can be used on the displays with less than 12-bit color depth. On the displays with the modifiable color palette Prima can install its own values in palettes, which may result in an effect known as *palette flashing*.

To switch to a non-default visual, use the `Prima.Visual` XRDB property or the `--visual` command-line argument. The list of visuals can be produced by the standard `xdpyinfo` command from the Xorg distribution, where each class of the visual corresponds to one of the six following classes:

**StaticGray**

All color cells are read-only and contain monochrome values only. A typical example is a two-color, black-and-white monochrome display. This visual is extremely rare.

**GrayScale**

Contains a modifiable color palette, and is capable of displaying monochrome values only. Theoretically, any paletted display on a monochrome monitor can be treated as a GrayScale visual. For both *GrayScale* and *StaticGray* visuals Prima resorts to dithering if it cannot get at least 32 evenly spaced gray values from black to white.

**StaticColor**

All color cells are read-only. A typical example is a PC display in the 16-color EGA mode. This visual is extremely rare.

**PseudoColor**

All color cells are modifiable. Typically, the 8-bit displays define this class as the default visual. For both *StaticColor* and *PseudoColor* visuals dithering is always used, although on the `PseudoColor` visuals Prima resorts to that only if the X server cannot allocate a required color.

On the `PseudoColor` and `GrayScale` visuals Prima allocates a small fixed set of colors, not used for palette modifications. When a pixmap is to be exported via clipboard, displayed in the menu, or sent to the window manager as an icon to be attached to a window, it is resampled so that it uses these colors only, which are guaranteed to stay immutable through the life of the application.

**TrueColor**

Each pixel value is explicitly coded as RGB. Typical examples are 16, 24, or 32-bit display modes. This visual class is the best in terms of visual quality.

**DirectColor**

Same as *TrueColor*, but additionally each pixel value can be reprogrammed. Not all hardware supports this visual, and usually this visual is not set as the default one. Prima supports this mode in the same way as it does the *TrueColor* visual without any additional features.

## Images

The X11 protocol does not standardize the pixel memory format for the *TrueColor* and *DirectColor* visuals, so there is a chance that Prima won't work on some bizarre hardware. Currently, Prima knows how to compose pixels of 15, 16, 24, and 32 bit depth, of contiguous ( not interspersed ) red-green-blue memory layout. Any other pixel memory layout causes Prima to fail.

Prima supports the shared memory image X extension that greatly speeds up displaying images on the X servers running on the same machine as the X client. The price for this is that if the Prima program aborts, the shared memory will never be returned to the OS. To remove the leftover segments, use your OS facilities, for example, `ipcrm` on Linux and BSD.

To disable the user of the shared memory with images use the `--no-shmem` switch in the command-line arguments.

The clipboard exchange of images is incompletely implemented, since Prima does not accompany ( and neither reads ) COLORMAP, FOREGROUND, and BACKGROUND clipboard data, which contains the RGB values for the paletted image. As a palliative, the clipboard-bound images are downgraded to the safe immutable set of colors.

A note on the images in the clipboard: contrary to the text in the clipboard, which can be used several times, images seemingly cannot. The Bitmap or Pixmap descriptor, stored in the clipboard, is rendered invalid after it has been read once. This does not apply to the more modern

clipboard exchange protocol based on images being encoded as binary data, f ex in PNG format. Prima prefers this exchange protocol whenever possible.

## Window managers

The original design of the X protocol did not include the notion of a window manager, and the latter was implemented as an ad-hoc patch, which results in possible race conditions when configuring widgets.

Prima was tested with alternating success under the following window managers: mutter, marco, mwm, kwin, wmaker, fvwm, fvwm2, enlightenment, sawfish, blackbox, 9wm, olvm, twm, and in no-WM environment.

### Protocols

Prima makes use of the `WM_DELETE_WINDOW` and `WM_TAKE_FOCUS` protocols. While the `WM_DELETE_WINDOW` protocol usage is straightforward and needs no further attention, the `WM_TAKE_FOCUS` protocol can be tricky, since X11 defines several of the input modes for a widget, which behave differently for each WM. In particular, the 'focus follows pointer' policy gives problems under twm and mwm when the navigation of drop-down combo boxes is greatly hindered by the window manager. The drop-down list is programmed so it is dismissed as soon its focus is gone; these window managers withdraw focus even if the pointer is over the focused widget's border.

### Hints

Size, position, icons, and other standard X hints are passed to WM in a standard way, and, as the inter-client communication manual ( ICCCM ) allows, are ften misinterpreted by window managers. Many ( wmaker, for example ) apply the coordinates given by the program not to the top-level widget itself, but to its decoration. mwm defines a list of the accepted icon sizes so these can be absurdly large, which adds to the confusion for an X client that can create an icon of any size but is unable to determine the best one.

### Non-standard properties

Prima tries to use the WM-specific hints for two window managers it knows about: mwm and kwin. For mwm ( Motif window manager ) Prima sets hints for the decoration border width and icons. For kwin ( and probably to others that conform to the specifications of http://www.freedesktop.org/ ) Prima uses the `NET_WM_STATE` property, in particular for the implementation of the window maximization and the visibility of windows in the taskbar.

Use of these properties explicitly contradicts ICCCM and definitely might lead to bugs in the future ( at least with `NET_WM_STATE`, since the Motif interface can hardly expected to be changed ). To disable the use of the non-standard WM properties, the `--icccm` command-line argument can be set.

## Unicode

The core X11 protocol does not support unicode, and a number of patches were applied to X servers and clients to make the situation change. Prima can only effectively support unicode text shaping and rendering if compiled with the Xft, fontconfig, harfbuzz, and the fribidi libraries.

The core X11 protocol supports text rendering when the text is sent as either 8-bit or 16-bit integers, but neither can be used to display unicode strings properly. Also, the core font transfer protocol suffers from ineffective memory representation, which creates latency when fonts with a large span of glyphs are loaded. Such fonts, in the still uncommon but standard iso10646 encoding, are the only media to display multi-encoding text if the Xft services are unavailable.

These and some other problems are efficiently solved by the Xft library, a superset of X core font functionality. Xft features Level 1 ( November 2003 ) unicode display and supports 32-bit text strings as well as UTF8-coded strings. Xft does not operate with charset encodings, and these are implemented in Prima using the iconv charset converter library.

### Clipboard

Prima supports the UTF8 text in the clipboard via the `UTF8_STRING` format.

Because any application can take ownership of the clipboard at any time, `open`/`close` brackets are not strictly respected in the X11 implementation. Practically, this means that when modern X11 clipboard daemons ( KDE klipper, for example ) interfere with the Prima clipboard, the results may not be consistent from the programmer's view, for example, the clipboard contains data after a `clear` call. It must be noted though that this behavior is expected by the users.

## Other XRDB resources

### Timeouts

The X11 protocol provides no such GUI helpers as the double-click event, cursor, or menu. Neither does it provide the related time how often, for example, a cursor should blink. Therefore Prima emulates these but allows the user to reprogram the corresponding timeouts. Prima recognizes the following properties, accessible either via the application name or the Prima class key. All timeouts are integer values, the number of milliseconds for the corresponding timeout property.

**Blinkinvisibletime.blinkinvisibletime: MSEC**

The cursor stays invisible in MSEC milliseconds.

Default value: 500

**Blinkvisibletime.blinkvisibletime: MSEC**

The cursor stays visible in MSEC milliseconds.

Default value: 500

**Clicktimeframe.clicktimeframe MSEC**

If the 'mouse down' and 'mouse up' events follow each other within MSEC milliseconds, the 'mouse click' event is synthesized.

Default value: 200

**Doubleclicktimeframe.doubleclicktimeframe MSEC**

If the 'mouse click' and 'mouse down' events follow each other within MSEC milliseconds, the 'mouse double click' event is synthesized.

Default value: 200

**Submenudelay.submenudelay MSEC**

When the user clicks on a menu item that points to a lower-level menu window, the latter is displayed after MSEC milliseconds.

Default value: 200

**Scrollfirst.scrollfirst MSEC**

When an auto-repetitive action, similar to keystroke events resulting from a long key press on the keyboard, is to be simulated, two timeout values are used, the 'first' and the 'next' delay. These actions are not simulated within Prima core, and the corresponding timeouts are advisory for the programmer. Prima widgets use it for automatic scrolling, either by a

scrollbar or by any other means. Also, the `Prima::Button` widgets can use these timeouts for the emulation of a key press in the `autoRepeat` mode.

`Scrollfirst` is a 'first' timeout.

Default value: 200

**Scrollnext.scrollnext MSEC**

Same as `Scrollfirst` but for the 'next' delay event.

Default value: 50

**Miscellaneous**

**Visual.visual: VISUAL_ID**

Selects the display visual VISUAL_ID which usually has a form of `0x??` hexadecimal number. Different visuals provide different color depth and pixel encoding schemes. Some X servers have badly chosen default visuals (for example, the default IRIX workstation setup has an 8-bit default visual selected), so this property can be used to fix things. A list of the visuals supported by the X display can be produced interactively by the standard `xdpyinfo` command from Xorg distribution.

Identical to the `--visual` command-line argument.

See the *Color* entry for more information.

**Wheeldown.wheeldown BUTTON**

BUTTON is the numeric ID of the X mouse button, which corresponds to the mouse wheel 'down' event.

Default value: 5 ( default values for wheeldown and wheelup are current de-facto most popular settings ).

**Wheelup.wheelup BUTTON**

BUTTON is the numeric ID of the X mouse button, which that is corresponds to the mouse wheel 'up' event.

Default value: 4

## Debugging

The famous 'use the source' call is highly actual with Prima. However, some debug information comes already compiled in, and can be activated by the `--debug` command-line key. Its parameter is a combination of letters where each activates the debugging of different subsystems:

- C - clipboard
- E - events subsystem
- F - fonts
- M - miscellaneous debug info
- P - palettes and colors
- X - XRDB
- A - all of the above

Example:

```
--debug=xf
```

Also, the built-in X API `XSynchronize` call, which enables the X protocol synchronization ( at the expense of operation slowdown though ) is activated with the `--sync` command-line argument, and can be used to ease the debugging.

### GTK

Prima can be compiled with GTK and can use its colors and font schemes, and also the standard GTK file dialogs. This can be disabled with the `--no-gtk` command line switch.

On MacOSX, GTK usually comes compiled with the Quartz backend, which means that Prima will get into problems with the remote X11 connections. Prima tries to detect this condition, but if the trouble persists, please use the `--no-gtk` switch (and please file a bug report so this can be fixed, too).

### Quartz

Prima can be compiled with the Cocoa library on MacOSX that gives access to the screen scraping functionality that is used by the Application.get_image method and which otherwise is non-functional with XQuartz. To disable this feature use the `--no-quartz` command-line switch.

## 12.3 Prima::sys::gtk::FileDialog

GTK file system dialogs

### Description

The module mimics the Prima file dialog classes `Prima::Dialog::OpenDialog` and `Prima::Dialog::SaveDialog`, defined in the *Prima::Dialog::FileDialog* section. The class names registered in the module are the same, but in the `Prima::sys::gtk` namespace.

## 12.4 Prima::sys::win32::FileDialog

Windows file system dialogs.

### Description

The module mimics the Prima file dialog classes `Prima::Dialog::OpenDialog` and `Prima::Dialog::SaveDialog`, defined in the *Prima::Dialog::FileDialog* section. The class names registered in the module are the same, but in the `Prima::sys::win32` namespace.

## 12.5 Prima::sys::XQuartz

MacOSX/XQuartz facilities

### Description

XQuartz emulates the X11 environment with certain limits, namely, it cannot grab bits from the screen, and it also hides the top-level menu from screen coordinates accessible for X11 clients. For example, a Mac with 1024x768 resolution will only report f.ex. 1024x746 size to Prima. If Prima is compiled with the Cocoa library, the `get_fullscreen_image` method circumvents these limitations and returns a shot of the whole screen, including the application menu.

Note that screen grabbing has to be allowed by the user or the administrator. To do that, Choose the Apple menu, System Preferences, click Security & Privacy, then click Privacy. Click on an icon on the left lower corner to allow changes. Then, in the screen recording tab, add XQuartz to the list of allowed applications. Note that it might not work if you run your application from a (remote) ssh session - I couldn't find how to enable screen grabbing for sshd.

## 12.6 Prima::sys::FS

Unicode-aware core file functions

### Description

Since perl win32 unicode support for files is unexistent, Prima has its own parallel set of functions mimicking native functions, ie open, chdir etc. This means that files with names that cannot be converted to ANSI (ie user-preferred) codepage are not visible in perl, but the functions below mitigate this problem.

The module exports the unicode-aware functions from `Prima::Utils` to override the core functions. Read more in the **Unicode-aware file system functions** entry in the *Prima::Utils* section.

#### Synopsis

```
use Prima::sys::FS;

my $fn = "\x{dead}\x{beef};
if ( _f $fn ) {
   open F, ">", $fn or die $!;
   close F;
}
print "ls: ", getdir, "\n";
print "pwd: ", getcwd, "\n";
```

### API

The module exports by default three groups of functions:

These are described in the **API** entry in the *Prima::Utils* section:

```
chdir chmod getcwd link mkdir open rename rmdir unlink utime
getenv setenv stat access getdir
opendir closedir rewinddir seekdir readdir telldir
```

The underscore-prefixed functions are same as the ones in -X in *perlfunc* (all are present except -T and -B ).

```
_r _w _x _o _R _W _X _O _e _z _s _f _d _l _p _S _b _c _t _u _g _k _M _A _C
```

The functions that are implemented in the module itself:

**abs_path**

Same as `Cwd::abs_path`.

**glob PATTERN**

More or less same as `CORE::glob` or `File::Glob::glob`.

**lstat PATH**

Same as `CORE::lstat`