Prima - the perl graphic toolkit

Dmitry Karasik

December 21, 2007

Contents

1 Introduction

Preface

Prima is an extensible Perl toolkit for multi-platform GUI development. Platforms supported include Linux, Windows NT/9x/2K, OS/2 and UNIX/X11 workstations (FreeBSD, IRIX, SunOS, Solaris and others). The toolkit contains a rich set of standard widgets and has emphasis on 2D image processing tasks. A Perl program using PRIMA looks and behaves identically on X, Win32 and OS/2 PM.

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

This document describes programming with Prima graphic toolkit, and is a collection of manual pages of Prima application program interface (API), written by D.Karasik, except Prima::IniFile and Prima::ScrollBar, written by A.Berezin.

Requirements

Prima supports perl versions 5.004 and above. The recommended perl versions are 5.005 and above. In UNIX(tm) environments, Prima can use the following graphic libraries: libjpeg, libungif, libtiff, libpng, libXpm.

Installation

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

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

There is a mailing list dedicated for various Prima-related discussions, prima@prima.eu.org. This list is also a proper place to send bug reports to. To subscribe to the list, send mail to `<majordomo@prima.eu.org>` and include `subscribe prima <optional address>` in the body of your message.

Authors

Dmitry Karasik, Anton Berezin, Vadim Belman

Credits

David Scott, Kai Fiebach, Johannes Blankenstein, Teo Sankaro, Mike Castle, H.Merijn Brand, Richard Morgan – thank you for your help.

2 Tutorials

2.1 Prima::tutorial

Introductory tutorial

Description

Programming graphic interfaces is often considered somewhat boring, and not without a cause. It is a small pride in knowing that your buttons and scrollbars work exactly as millions of others buttons and scrollbars do, so whichever GUI toolkit is chosen, it is usually regarded as a tool of small importance, and the less 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 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 are more than one way to do it). Perl itself is 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 three lines of code:

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

Line 1 here is the invocation of modules *Prima* and *Prima::Application*. Sure, one can explicitly invoke both `use Prima` and `use Prima::Application` etc etc, but as module Prima doesn't export method names, the exemplified syntax is well-suited for such a compression.

Line 2 creates a window of *Prima::MainWindow* class, which is visualized as a screen window, titled as 'Hello world'. The class terminates the application when the window is closed; this is the only difference from 'Window' windows, that do nothing after their closing. From here, `Prima::` prefix in class names will be omitted, and will be used only when necessary, such as in code examples.

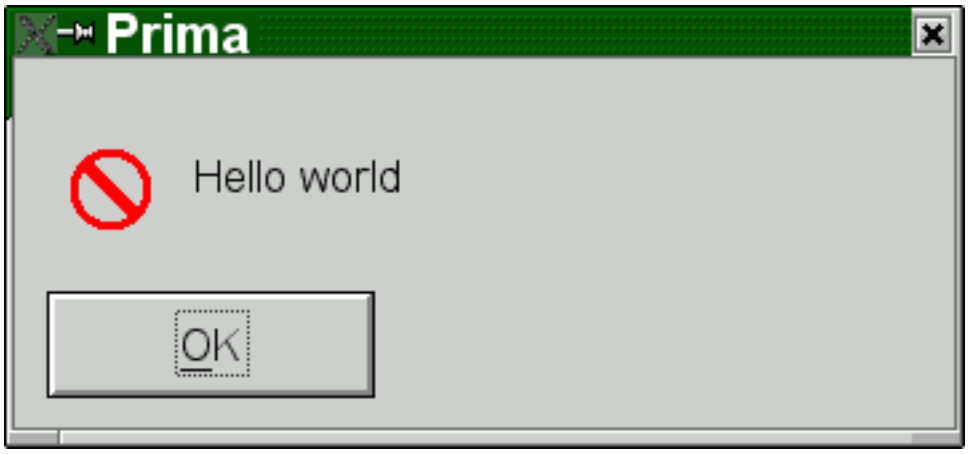
Line 3 enters the Prima event loop. The loop is terminated when the only instance of *Application* class, created by `use Prima::Application` invocation and stored in `$::application` scalar, is destroyed.

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

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

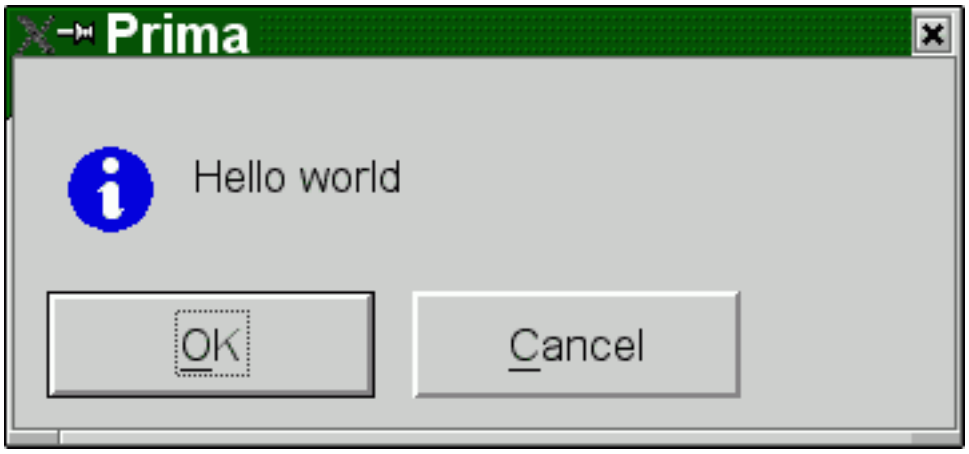
but it is not illustrative and not useful. `Prima::message` is rarely used, and is one of few methods contained in `Prima::` namespace. To display a message, the *MsgBox* module is often preferred, with its control over message buttons and pre-defined icons. With its use, the code above can be rewritten as

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



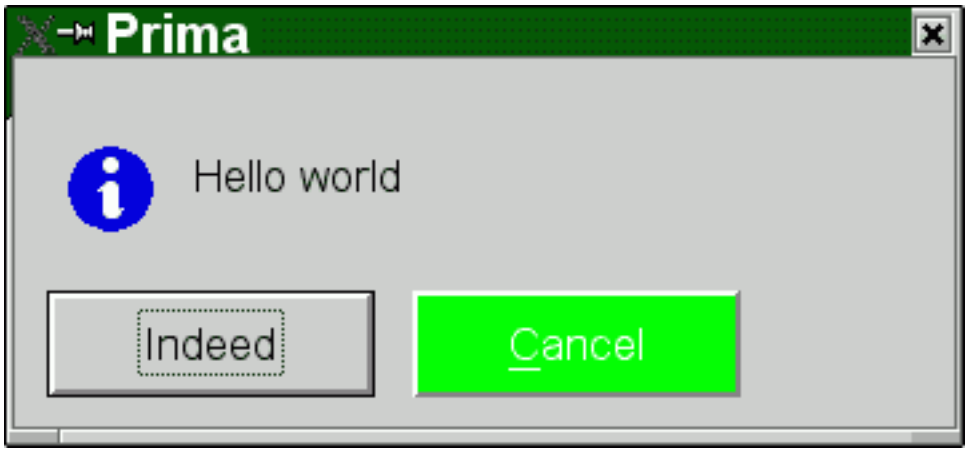
but where `Prima::message` accepts the only text scalar parameters, `Prima::MsgBox::message` can do lot more. For example

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



displays two buttons and an icon. A small achievement, but the following 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',
                },
        }
});
```



The definition of many object properties at once is a major feature of Prima, and is seen throughout the toolkit. Returning back 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 color constant is registered in `cl::` namespace. In Prima there is a number of such two- and three-letter namespaces, containing usually integer constants for various purposes. The design reason for choosing such syntax over string constants (as 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, defined on descendants of *Widget* class. These can be set in `new` (alias `create`) call, or referred 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 have syntax identical to the properties. Changing the code again, we can 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 a standard button is probably should be preferred:

```
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 to position a widget, Prima provides *pack* and *place* interfaces. Here one can replace the line

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

to

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

with exactly the same effect.

Widgets overview

Prima contains a set of standard (in GUI terms) widgets, such as buttons, input lines, list boxes, scroll bars, etc etc. These are diluted with the other more exotic widgets, such as 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 table of these, an excerpt of `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 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::Header* section - a multi-tabbed 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 and input lines, dial widget etc.
the *Prima::TextView* section - rich text browser widget

Building a menu

In Prima, a tree-like menu is built by passing a nested set of 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 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 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 items are treated as a menu separator:

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

The one-item syntax is needed when the separator menu item need to be later 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 a separator. The other types (text,image,sub-menu) are discerned 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 clearly a sub-menu. The reference, as in the example, may contain more menu items, in the recursive fashion:

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

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

Three items - [ID, text, action]

Four items - [text, hot key text, hot key, action]

Five items - [ID, text, hot key text, hot key, action]

Six items - [ID, text, hot key text, hot key, action, data]

Image 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 them 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 with menu actions that can be overridden:

```
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) ],
                ]],
        ],
);
```

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

Another trick is to define a hot key. While its description can be arbitrary, and will be displayed as is, the hot key definition can be constructed in two ways. It is either a literal such as `^A` for Control+A, or `@B` for Alt+B, or `^@#F10` for Control+Alt+Shift+F10. Or, alternatively, it is a combination of `km::` constants either with ordinal of the character letter or the key code, where the key code is one of `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 .

3 Core toolkit classes

3.1 Prima

A perl graphic toolkit

Synopsis

```
use Prima qw(Application Buttons);

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

run Prima;
```

Description

The toolkit is combined from two basic set of classes - core and external. The core classes are coded in C and form a base line 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 easily expandable set of widgets, written completely in perl and communicating with the system using Prima library calls.

The core classes form an 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::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 'Hello, world' title and a centered button with the same text. The program terminates after the button is pressed.

A basic construct for a program written with Prima obviously requires

```
use Prima;
```

code; however, the effective programming requires usage of the other modules, for example, `Prima::Buttons`, which contains set of button widgets. `Prima.pm` module can be invoked 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;
```

sentence and requires a `Prima::Application` object to be created beforehand. Invoking `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 window is created by invoking

```
new Prima::Window();
```

or

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

code with the additional parameters. Actually, all Prima objects are created by such a scheme. The class name is passed as the first parameter, and a custom set of parameters is passed afterwards. These parameters are usually represented in a hash syntax, although actually passed as an array. The hash syntax is preferred for the code readability:

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

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

In the example, the following properties are set :

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

Property values can be of any type, given that they are scalar. As depicted here, `::text` property accepts a string, `::size` - an anonymous array of two integers and `onClick` - a sub.

onXxxx are special properties that form a class of *events*, which share 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 `Prima::` namespace.

message TEXT

Displays a system message box with TEXT.

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 `--help` argument to get the list of supported arguments. Programmaticaly, setting and obtaining these options can be done by using `Prima::options` routine.

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

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

will also do.

3.2 Prima::Object

Prima toolkit base 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',
    # replace the old values, while onPostMessage 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 objects of the Prima toolkit hierarchy. All the other objects are derived from the Component class, which in turn is the only descendant of Object class. Both of these classes are never used for spawning their instances, although this is possible using

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

call. This document describes the basic concepts of the OO programming with Prima toolkit. Although Component has wider functionality than Object, all examples will be explained on Component, since Object has no descendant classes and all the functionality of Object is present in Component. Some of the information here can be found in the *Prima::internals* section as well, the difference is that the *Prima::internals* section considers the coding tasks from a C programmer's view, whereas this document is wholly about perl programming.

Object base features

Creation

Object creation has fixed syntax:

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

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

create

create() calls 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 (see Classes.pm):

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

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

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

$a1's owner will be $::application

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

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

profile_check_in

A profile_check_in() method merges the default and the parameter profiles. By default all specified parameters have the ultimate precedence over the default ones, but in case the specification is incomplete or ambiguous, the profile_check_in()'s task is to determine actual parameter values. In case of Component, this method maintains a simple automatic naming of the newly created objects. If the object name was not specified with create(), it is assigned to a concatenated class name with an integer - Component1, Component2 etc.

Another example can be taken from Prima::Widget::profile_check_in(). Prima::Widget horizontal position can be specified by using basic `left` and `width` parameters, and as well by auxiliary `right`, `size` and `rect`. The default of both `left` and `width` is 100. But if only `right` parameter, for example, was passed to create() it is profile_check_in() job to determine `left` value, given that `width` is still 100.

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

init

init() duty is to map the profile content into object, e.g., assign `name` property to `name` parameter value, and so on - for all relevant parameters. After that, it has to return the profile in order the overridden subsequent init() methods can perform same actions. This stage along with the previous is exemplified in almost all Prima modules.

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

After init() finishes, create() calls setup() method

setup

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

As can be noticed from the code pieces above, a successful create() call returns a newly created object. If an error condition occurred, undef is returned. It must be noted, that only errors that were generated via die() during init() stage result in undef. Other errors raise an exception instead. It is not recommended to frame create() calls in an `eval{}` block, because the error conditions can only occur in two situations. The first is a system error, either inside perl or Prima guts, and not much can be done here, since that error can very probably lead to an unstable program and almost always signals an implementation bug. The second reason is a caller's error, when an unexistent parameter key or invalid value is passed; such conditions are not subject to a runtime error handling as are not the syntax errors.

After create(), the object is subject to the event flow. As `onCreate` event is the first event the object receives, only after that stage other events can be circulated.

Destruction

Object destruction can be caused by many conditions, but all execution flow is finally passed through destroy() method. destroy(), as well as create() performs several finalizing steps:

cleanup

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

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

done

done() method is the pair to init(), and is the place where all object resources are freed. Although it is as safe to overload done() as init(), it almost never gets overloaded, primarily because overloading `onDestroy` is easier.

The typical conditions that lead to object destructions are direct destroy() call, garbage collections mechanisms, user-initiated window close (on `Prima::Window` only), and exception during init() stage. Thus, one must be careful implementing done() which is called after init() throws an exception.

Methods

The class methods are declared and used with perl OO syntax, which allow both method of object referencing:

```
$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

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

syntax. Most Prima methods have fixed number of parameters.

Properties

Properties are methods that combine functionality of two ephemeral "get" and "set" methods. 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 large amount of these get's and set's, but later these method pairs were deprecated in the favor of properties. Currently, there is only one method name() (referred as `::name` later in the documentation).

The property returns a value if no parameters (except the object) are passed, and changes the internal data to the passed parameters otherwise. Here's a sketch code for `::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 set() method. The set() method accepts a hash, that is much like to create(), and assigns the 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 positive effect of a possible speed-up is gained by eliminating C-to-perl and perl-to-C calls, especially if the code called is implemented in C. The negative effect of such 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 an only way to achieve the result. An example of the latter case from the *Prima::internals* section 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 call for such a change using

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

when it is known beforehand that `Prima::Image::set` is aware of such combinations 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 on an attempt to do so.

Links between objects

Prima::Component descendants can be used as containers, as objects that are on a higher hierarchy level than the others. This scheme is implemented in a child-owner relationship. The 'children' objects have the `::owner` property value assigned to a reference to a 'owner' object, while the 'owner' object conducts the list of its children. It is a one-to-many hierarchy scheme, as a 'child' object can have only one owner, but 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.

Prima::Component::owner property maintains this relation, and is writable - the object can change its owner dynamically. There is no corresponding property that manages children objects, but is a 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) in the geometrical interior of 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 cleaning up the widgets and quitting gracefully.

Implementation note: name 'owner' was taken instead of initial 'parent', because the 'parent' is a fixed term for widget hierarchy relationship description. 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 boundaries.

The special convenience variety of create(), the insert() method is used to explicitly select owner of the newly created object. insert() can be considered a 'constructor' in OO-terms. It makes the construct

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

more readable by introducing

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

scheme. These two code blocks are identical to each other.

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 many-to-many scheme, where every object can have many links to other objects. This functionality is managed by attach() and detach() methods.

Events

Prima::Component descendants employ a well-developed event propagation mechanism, which allows handling 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 are present in namespaces of all Prima objects. New classes can register their own events and define their execution flow, using notification_types() method. This method returns all available information about the events registered in a class.

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

Propagation

The event propagation mechanism has three layers of user-defined callback registration, that are called in different order and contexts when an event is triggered. The examples below show the usage of these layers. It is assumed that an implicit

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

call is issued for all these examples.

Direct methods

As it is usual in OO programming, event callback routines are declared as methods. 'Direct methods' employ such a paradigm, so if a class method with name `on_postmessage` is present, it will be called as a method (i.e., in the object context) when `onPostMessage` event is triggered. Example:

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

The callback name is a modified lower-case event name: the name for 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 the internal object book-keeping, reacting on the events that are not designed to be passed higher. For example, a Prima::Button class catches mouse and keyboard events in such a fashion, because usually the only notification that is interesting for the code that employs push-buttons is `Click`. This scheme is convenient when an event handling routine serves the 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 direct methods, except that they get called in context of two, not one, objects. If, for example, a $obj's owner, $owner would be interested in $obj's PostMessage event, it would register the notification callback by

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

where the actual callback sub will be

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

Note that the naming style is different - the callback name is constructed from object name (let 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 onbjects). Note also that context objects are $self (that equals $owner) and $obj.

The delegated methods can be used not only for the 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 owner-child relation, it is a good practice to add Destroy notification to the object which 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 developed, but 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 method using perl anonymous subroutines (subs) for the easy customization.

Contrary to the usual OO event implementations, when only one routine per class dispatches an event, and calls inherited handlers when it is appropriate, 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 multiple by the design. There are three syntaxes for setting such a event hook; the example below sets a hook on $obj using each syntax for a different situation:

- during create():

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

- after create using set()

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

- after create using event name:

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

As was noted in the *Prima* section, the events can be addressed as properties, with the exception that they are not substitutive but additive. The additivity is that when the latter type of syntax is used, the subs already registered do not get overwritten or discarded but stack in 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.

This, it is a distinctive feature of a toolkit is that two objects of 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 is risen about what are callbacks call priorities and when does the event processing stop. One of ways to regulate the event flow is based on prototyping events, by using notification_types() event type description. This function returns a hash, where keys are the event names and the values are the constants that describe the event flow. The constant can be a bitwise OR combination of several basic flow constants, that control the three aspects of the event flow.

Order

If both anonymous subs and direct/delegated methods are present, it must be decided which callback class must be called first. Both 'orders' are useful: for example, if it is designed that a class's default action is to be overridden, it is better to call the custom actions first. If, on the contrary, the class action is primary, and the others are supplementary, the reverse order is preferred. One of two `nt::PrivateFirst` and `nt::CustomFirst` constants defines the order.

Direction

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

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

results in `21`, not `12` because PostMessage event type is prototyped `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 governed by `nt::Single` bit in 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 contrary of `nt::Single`, the `nt::Multiple` constant sets the execution control to call all the available callbacks, with respect to direction and order bits.

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

Although there are 12 possible event type combinations, a half of them are not viable. Another half were assigned to unique more-less intelligible 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 event success state. The 0 and 1 results in general do not mean either success or failure, they simply reflect the fact whether 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 from Component::eventFlag property. Since it is allowed to call notify() inside event callbacks, the object maintains a stack for those states. Component::eventFlags always works with the topmost one, and fails if is called from outside the event processing stage. Actually, clear_event() is an alias for ::eventFlag(0) call. The state stack is operated by push_event() and pop_event() methods.

Implementation note: a call of clear_event() inside a `nt::Event`-prototyped event call does not automatically stops the execution. The execution stops if the state value equals to 0 after the callback is finished. A ::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 notify() logic is somewhat complicated, it is rather simple with `nt::Single` case. The helper function get_notify_sub() returns the context of callback to-be-called, so it can be used to emulate notify() behavior. Example:

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

can be expressed in more cumbersome, but efficient code if `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;
```

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);
```

can NAME, CACHE = 1

Checks if an object namespace contains a NAME method. Returns the code reference to it, if found, and undef if not. If CACHE is true, caches the result to speed-up subsequent calls.

cleanup

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

create CLASS, %PARAMETERS

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

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

Is never called in an object context.

Alias: new()

destroy

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

done

Called by destroy() 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 done() call.

get @PARAMETERS

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

init %PARAMETERS

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

insert CLASS, %PARAMETERS

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

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

is adequate to

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

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

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

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

new CLASS, %PARAMETERS

Same as the *create* entry.

profile_add PROFILE

The first stage of object creation process. PROFILE is a reference to a PARAMETERS hash, passed to create(). It is merged with profile_default() after passing both to profile_check_in(). The merge result is stored back in PROFILE. Is never called directly.

profile_check_in CUSTOM_PROFILE, DEFAULT_PROFILE

The second stage of object creation process. Resolves eventual ambiguities in CUSTOM_PROFILE, which is the reference to PARAMETERS passed to create(), by comparing to and using default values from DEFAULT_PROFILE, which is the result of profile_default() method. Is never called directly.

profile_default

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

Can be called in a context of 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 an equivalent to

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

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

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

setup

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

Prima::Component methods

add_notification NAME, SUB, REFERER = undef, INDEX = -1

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

INDEX is a desired insert position in the notification list. By default it is -1, what 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 positive integer value on success, 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 OBJECT to the attached objects list and increases OBJECT's reference count. The list can not hold more than one reference to the same object. The warning is issued on such an attempt.

See also: `detach`.

bring NAME

Looks for a immediate child object that has name equals 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;
```

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 nt::Event-prototyped events.

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

detach OBJECT, KILL

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

See also: `attach`

event_error

Issues a system-dependent warning sound signal.

event_hook [SUB]

Installs a SUB to receive all events on all Prima objects. SUB receives 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 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 *Prima::EventHook* section for multiplexing of the hook access.

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

get_components

Returns array of the child objects.

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

get_handle

Returns a system-dependent handle for the object. For example, Prima::Widget return 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 an eventual interprocess communication scheme.

get_notification NAME, @INDEX_LIST

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

See also: `remove_notification`, `add_notification`.

get_notify_sub NAME

A convenience method for nt::Single-prototyped events. Returns code reference and 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 a context of 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 context of the object. The calling order is described by `nt::` constants, contained in the notification_types() result hash.

notify() accepts variable number of parameters, and while it is possible, it is not recommended to call notify() with the exceeding 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 event processing stage brackets.

See `push_event`, the *Events* entry

post_message SCALAR1, SCALAR2

Calls `PostMessage` event with parameters SCALAR1 and SCALAR2 once during 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 event processing stage brackets.

See `pop_event`, the *Events* entry

remove_notification ID

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

See also: `add_notification`, `get_notification`.

set_notification NAME, SUB

Adds SUB to the event NAME notification list. Almost never used directly, but is a key point in enabling the following notification add syntax

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

or

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

that are shortcuts for

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

unlink_notifier REFERER

Removes all notification subs from all event lists bound to REFERER object.

Prima::Component properties

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 [<REFERER>, NAME, <NAME>, < <REFERER>, NAME, ... >]

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

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

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

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

See also: the *Delegated methods* entry.

name NAME

Maintains object name. NAME can be an arbitrary string, however it is recommended against 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 predefine some object parameters by its name (or class), then it is impossible to know beforehand the system naming restrictions. For example, in X 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 X resource unusable.

owner OBJECT

Selects an owner of the object, which may be any Prima::Component descendant. Setting an owner to a 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 owner set to undef. Prima::Image objects are indifferent to the value of the owner property.

Changing owner dynamically is allowed, but it is a main source of implementation bugs, since the whole hierarchy tree is needed 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 `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 at runtime 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 an event sees. Called automatically after init() is finished. Is never called directly.

Destroy

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

PostMessage SCALAR1, SCALAR2

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

3.3 Prima::Classes

Binder module for the built-in classes.

Description

`Prima::Classes` and the *Prima::Const* section is a minimal set of perl modules needed for the toolkit. 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

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 Prima::Component. It provides access to the object-bound 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 Prima::Drawable class by itself provides only the interface. It provides a three-state object access - when drawing and painting is enabled, when these are disabled, and the information acquisition state. By default, the object is created in paint-disabled state. To switch to the enabled state, begin_paint() method is used. Once in the enabled state, the object drawing and painting methods apply to the object-bound canvas. To return to the disabled state, 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. These states differ on how do they apply to a graphic context and a canvas.

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, bound to the object, where the drawing and painting methods are applied to.

In all three states a graphic context is allowed to be modified, but in different ways. In the disabled state the graphic context values form a template values; when a object enters the information or the enabled state, the 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 new 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`, `::clipRect` and `::translate` properties are exceptions. They can not be used in the disabled state; their values are neither recorded nor used as a template).

That is, in disabled state any Drawable maintains only the graphic context. 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 might 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;
}
```

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

The information state is as same as the enabled state, but the changes to a 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 get_text_width() method, that 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.

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 standard 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 in the object disabled state, it is recorded verbatim; color properties are no exception. After the object switched to the enabled state, a color value is transformed to a system color representation, which might be different from Prima's. For example, if a display color depth is 15 bits, 5 bits for every component, then 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 bit, 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 small, dithering algorithms might apply.

Note: not only color properties, but all graphic context properties allow all possible values in the disabled state, which transformed into system-allowed values in 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 (at least) two color properties on each drawable - `::color` and `::backColor`. The values they operate are integers in the discussed above RRGGBB format, however, the toolkit defines some mnemonic 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
```

As stated before, it is not unlikely that if a device color depth is small, the primitives plotted in particular colors will be drawn with dithered or incorrect colors. This usually happens on paletted displays, with 256 or less colors.

There exists 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 possibility to avoid mixed colors drawing by obtaining indirect information about solid colors, supported by a device. Another method is to use `::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 system palette flash when a window focus toggles.

Both of these methods are applicable both with drawing routines and image output. An image desired to output with least distortion is advised to export its palette to an output device, because

images usually are not subject to automatic dithering algorithms. Prima::ImageViewer module employs this scheme.

Fonts

Prima maintains its own font naming convention, that usually does not conform to system's. Since its 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 object-bound graphic context. The font hash keys that are acceptable on set-call are:

name

The font name string. If there is no such font, a default font name is used. To select 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

```
           height - internal_leading
size =  --------------------------- * 72.27
                resolution
```

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

width

A 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 hight

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

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

pitch

A one of three constants:

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

fp::Default specifies no interest about font pitch selection. fp::Fixed is set when a monospaced (all glyphs are of 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.

direction

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

encoding

A string value, one of the strings returned by `Prima::Application::font_encodings`. Selects 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 set() call, like in Prima::Object:

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

is adequate to

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

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

In order to query the full list of fonts available to a graphic device, a `::fonts` method is used. This method is not present in 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 a 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 `Prima::Drawable::font` method.

family

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

vector

A boolean; true if the font is vector (e.g. can be scaled with no quality loss), false otherwise. The false value does not show if the font can be scaled at all - the behavior is system-dependent. Win32 and OS/2 can scale all non-vector fonts; X11 only the fonts specified as the scalable.

ascent

Number of pixels between a glyph baseline and descent line.

descent

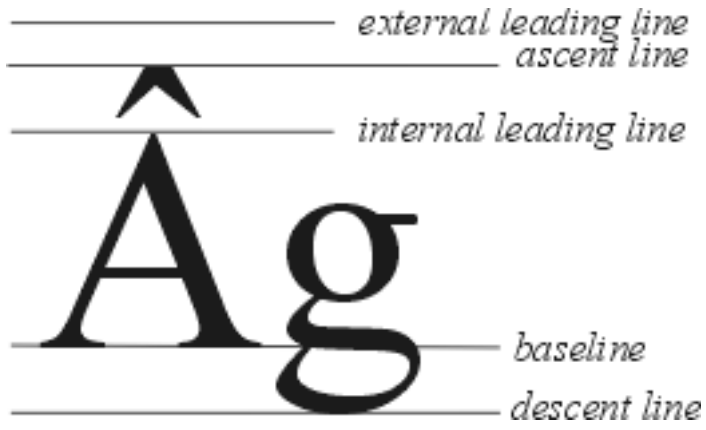
Number of pixels between a glyph baseline and descent line.

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.



weight

A 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

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

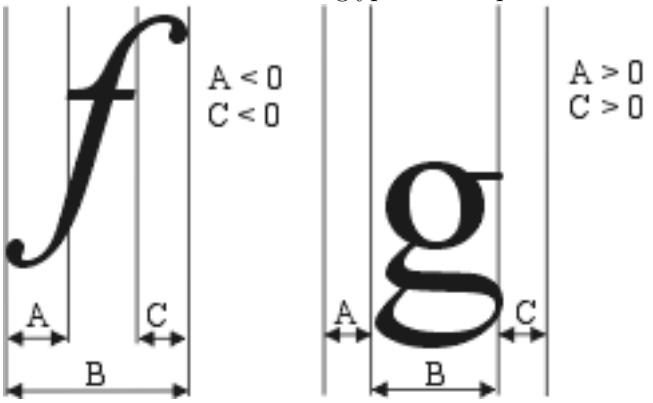
Index of the default character used to divide words. In a typical western language font it is 32, ASCII space character.

defaultChar

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

Font ABC metrics

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



A and C are negative, if a glyphs 'hangs' over it neighbors, as shown in 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 picture on the right. As can be seen, B is the width of a glyph's black part.

ABC metrics returned by get_font_abc() method.

Raster operations

A drawable 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.

The toolkit defines the following operations:

```
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
```

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 - it is only OS/2 system that supports currently rop2 modes others than Copy and NoOp.

The raster operations apply to all graphic primitives except SetPixel.

Coordinates

The Prima toolkit employs a geometrical 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 `::translate` property, that translates the (0,0) point to the given offset. Calls to `::translate`, `::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 can not be used in when an object is in its disabled state.

API

Graphic context properties

backColor COLOR

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

color COLOR

Reflects foreground color in 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).

fillWinding BOOLEAN

Affect filling style of complex polygonal shapes filled by `fillpoly`. If 1, the filled shape contains no holes; otherwise, holes are present where the shape edges cross.

Default value: 0

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

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, each representing 8 bits of each line in a pattern, where the first integer is the topmost pattern line, and the bit 0x80 is the leftmost pixel in the line.

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

```
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
```

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

An example below shows encoding of 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 ]);
```

On a get-call always returns an array, never a `fp::` constant.

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().

lineEnd VALUE

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

```
le::Flat
le::Square
le::Round

constants. le::Round is the default value.
```

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 a length of a dash, and each odd byte is a length of a gap.

The predefined constants are:

```
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" /* _.._.._.._.. */
```

Not all systems are capable of accepting user-defined line patterns, and in such situation the `lp::` constants are mapped to the system-defined patterns. In Win9x, for example, lp::DashDotDot is much different from its string definition therefore.

Default value is lp::Solid.

lineWidth WIDTH

Selects a line width for plotting primitives. 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 a way the lines are joined.

Default value is 0.

palette [@PALETTE]

Selects solid colors in a system palette, as many as possible. PALETTE is an array of integer triplets, where each is R, G and B component. The call

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

selects a gray-cyan color, for example.

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

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 the *Graphic context and canvas* entry).

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.

splinePrecision INT

Selects number of steps to use for each spline segment in `spline` and `fill_spline` calls. In other words, determines smoothness of a curve. Minimum accepted value, 1, produces straight lines; maximum value is not present, though it is hardly practical to set it higher than the output device resolution.

Default value: 24

textOpaque FLAG

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

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 font descent line. Default 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 center in X, Y and DIAMETER_X and DIAMETER_Y axis from START_ANGLE to END_ANGLE.

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

bar X1, Y1, X2, Y2

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

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

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

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

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

clear <X1, Y1, X2, Y2>

Draws rectangle filled with pure background color with (X1,Y1) - (X2,Y2) extents. Can be called without parameters, in this case fills all canvas area.

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 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 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 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 center in X, Y and DIAMETER_X and DIAMETER_Y axis.

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 center in X, Y and DIAMETER_X and DIAMETER_Y axis from START_ANGLE to END_ANGLE (see chord()).

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

fill_ellipse X, Y, DIAMETER_X, DIAMETER_Y

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

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

fillpoly \@POLYGON

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

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

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

See also: polyline().

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

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

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

fill_spline \@POLYGON

Fills a polygonal area defined by a curve, projected by applying cubic spline interpolation to POLYGON set of points. Number of vertices between each polygon equals to current value of `splinePrecision` property. POLYGON must present an array of integer pair in (X,Y) format. Example:

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

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

See also: spline, splinePrecision, render_spline

flood_fill X, Y, COLOR, SINGLEBORDER = 1

Fills an area of the canvas in 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, rop, rop2

line X1, Y1, X2, Y2

Plots a 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 a straight line per quartet.

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

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.

polyline \@POLYGON

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

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

See also: fillpoly().

put_image X, Y, OBJECT, [ROP]

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, value of `::rop` property is used.

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

Copies a OBJECT from a source rectangle into a destination rectangle, stretching or compressing the OBJECT 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 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

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

Draws 3d-shaded rectangle in boundaries 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_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.

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 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, rop, rop2

spline \@POLYGON

Draws a cubic spline defined by set of POLYGON points. Number of vertices between each polygon equals to current value of `splinePrecision` property. POLYGON must present an array of integer pair in (X,Y) format.

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

See also: fill_spline, splinePrecision, render_spline.

stretch_image X, Y, DEST_WIDTH, DEST_HEIGHT, OBJECT, [ROP]

Copies a OBJECT into a destination rectangle, stretching or compressing the OBJECT 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, value of `::rop` property is used.

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

Context used: rop

text_out TEXT, X, Y

Draws TEXT string at (X,Y).

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

Methods

begin_paint

Enters the enabled (active paint) state, returns success flag. Once the object is in enabled state, painting and drawing methods can perform write operations on a canvas.

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

begin_paint_info

Enters the information state, returns success flag. The object information state is same as 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

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 full set of elements described in the *Fonts* entry.

If called without parameters, returns an array of same hashes where each hash represents a member of font family from every system font set. It this special case, each font hash contains 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_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 parameters are -1, the default range (0 and 255) is assumed. UNICODE boolean flag is responsible of representation of characters in 127-255 range. If 0, the default,

encoding-dependent characters are assumed. If 1, the U007F-U00FF glyphs from Latin-1 set are used.

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

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

Context used: font

get_nearest_color COLOR

Returns a nearest possible solid color in representation of object-bound graphic device. Always returns same color if the device bit depth is equals or greater than 24.

get_paint_state

Returns paint state value - 0 if the object is in the disabled state, 1 for the enabled state, 2 for the information state.

get_physical_palette

Returns an anonymous array of integers, in (R,G,B) format, every color entry described by three values, in range 0 - 255.

The physical palette array is non-empty only on paletted graphic devices, the true color devices 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 change the system colors at any moment.

get_text_width TEXT, ADD_OVERHANG = 0

Returns TEXT string width if it would be drawn using currently selected font.

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 TEXT string extensions if it would be drawn using currently selected font.

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

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

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

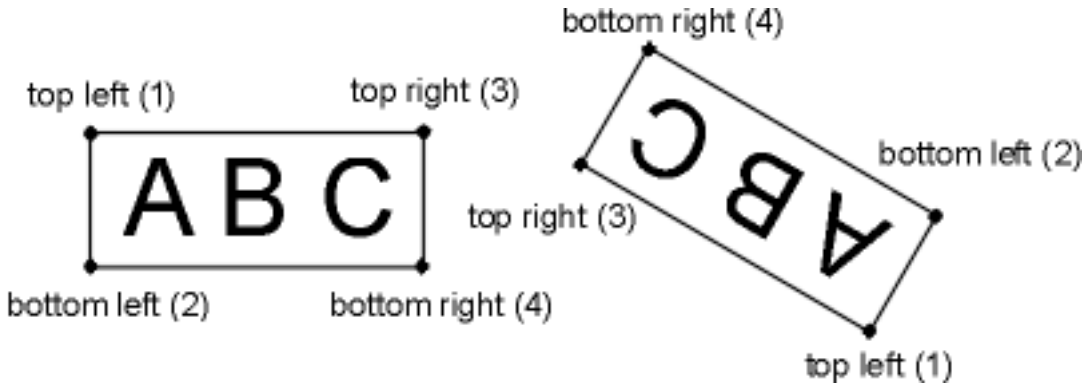
3: end of string at ascent line (top right)

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

5: concatenation point

The concatenation point coordinates (XC,YC) are coordinated passed to consequent text_out() call so the conjoint string would plot as if it was a part of 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



render_spline VERTICES, [PRECISION]

Renders cubic spline from set of VERTICES to a polyline with given precision. The method can be called as static, i.e. with no object initialized. PRECISION integer, if not given, is read from `splinePrecision` property if the method was called on an alive object; in case of static call, default value 24 is used.

The method is internally used by `spline` and `fill_spline`, and is provided for cases when these are insufficient.

text_wrap TEXT, WIDTH, OPTIONS, TAB_INDENT = 8

Breaks TEXT string in chunks that would fit into WIDTH pixels wide box.

The break algorithm and its result are governed by OPTIONS integer value which is a combination of `tw::` constants:

tw::CalcMnemonic

Use 'hot key' semantics, when a character preceded by ~ has special meaning - it gets underlined. If this bit is set, the first tilde character used as an escapement is not calculated, and never appeared in the result apart from the escaped character.

tw::CollapseTilde

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

tw::CalcTabs

If set, calculates a tab ('\t') character as TAB_INDENT times space characters.

tw::ExpandTabs

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

tw::BreakSingle

Defines procedure behavior when the text cannot be fit in WIDTH, does not affect anything otherwise.

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

tw::NewLineBreak

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

tw::SpaceBreak

Forces new chunk after a space character (' ') or a tab character ('\t') are met.

tw::ReturnChunks

Defines the result of text_wrap() function.

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

If unset, the resulting array consists from text chunks.

tw::ReturnLines

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

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. keeps breaking TEXT by words, not by characters.

tw::ReturnFirstLineLength

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

If OPTIONS has tw::CalcMnemonic or tw::CollapseTilde bits set, then the last scalar in the array result is a special hash reference. The hash contains extra information regarding the 'hot key' underline position - it is assumed that '~' - escapement denotes an underlined character. The hash contains the following keys:

tildeLine

Chunk index that contains the escaped character. Set to undef if no ~ - escapement was found. The other hash information is not relevant in this case.

tildeStart

Horizontal offset of a beginning of the line that underlines the escaped character.

tildeEnd

Horizontal offset of an end of the line that underlines the escaped character.

tildeChar

The escaped character.

Context used: font

3.5 Prima::Image

Bitmap routines

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;

# mangle
$i-> size( 64, 64);

# file operations
$i-> save('a.gif') or die "Error saving:$@\n";
$i-> load('a.gif') or die "Error loading:$@\n";

# draw on application
$::application-> begin_paint;

# an image is drawn as specified by its palette
$::application-> set( color => cl::Red, backColor => cl::Green);
$::application-> put_image( 100, 100, $i);

# a bitmap is drawn as specified by destination device colors
$::application-> put_image( 200, 100, $i-> bitmap);
```

Description

Prima::Image, *Prima::Icon* and *Prima::DeviceBitmap* are classes for bitmap handling, including 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 contains a transparency mask along with the regular data.

Usage

Images usually are represented as a memory area, where pixel data are stored row-wise. The Prima toolkit is no exception, however, it does not assume that the 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 the different needs, but can be easily converted to each other, with `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 number of conversion and scaling functions. The *Prima::DeviceBitmap* class has almost none of additional functionality, and is targeted to efficient graphic input and output.

Graphic input and output

As descendants of *Prima::Drawable*, all *Prima::Image*, *Prima::Icon* and *Prima::DeviceBitmap* objects are subject to three-state painting mode - normal (disabled), painting (enabled) and informational. *Prima::DeviceBitmap* is, however, exists only in the enabled state, and can not be switched to the other two.

When an object enters the enabled state, it serves as a canvas, and all *Prima::Drawable* operations can be performed on it. When the object is back to the disabled state, the graphic information is stored into the object associated memory, in the pixel format, supported by the toolkit. This information can be visualized by using one of `Prima::Drawable::put_image` group methods. If the object enters the enabled state again, the graphic information is presented as an initial state of a bitmap.

It must be noted, that if an implicit conversion takes place after an object enters and before it leaves the enabled state, as it is with *Prima::Image* and *Prima::Icon*, the bitmap is converted to the system pixel format. During such conversion some information can be lost, due to down-sampling, and there is no way to preserve the information. This does not happen with *Prima::DeviceBitmap*.

Image objects can be drawn upon images, as well as on the screen and the *Prima::Widget* section objects. This operation is performed via one of *Prima::Drawable::put_image* group methods (see the *Prima::Drawable* section), and can be called with the image object disregarding the paint state. The following code illustrates the dualism of an image object, where it can serve both as a drawing surface and as a drawing tool:

```
my $a = Prima::Image-> create( 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;
```

It must be noted, that `put_image`, `stretch_image` and `put_image_indirect` are only painting methods that allow drawing on an image that is in its paint-disabled state. Moreover, in such context they only allow `Prima::Image` descendants to be passed as a source image object. This functionality does not imply that the image is internally switched to the paint-enabled state and back; the painting is performed without switching and without interference with the system's graphical layer.

Another special case is a 1-bit (monochrome) DeviceBitmap. When it is drawn upon a drawable with bit depth greater than 1, the drawable's color and backColor properties are used to reflect 1 and 0 bits, respectively. On a 1-bit drawable this does not happen, and the color properties are not used.

File input and output

Depending on the toolkit configuration, images can be read and written in different formats. This functionality in accessible via `load()` and `save()` methods. the *Prima::image-load* section is dedicated to the description of loading and saving parameters, that can be passed to the 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 format etc. In this document, `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 occurred or an unknown image format was used.

When an image is loaded, the old bitmap memory content is discarded, and the image attributes are changed accordingly to the loaded image. Along with these, an image palette is loaded, if available, and a pixel format is assigned, closest or identical to the pixel format in the image file.

Pixel formats

Prima::Image supports a number of pixel formats, governed by the `::type` property. It is reflected by an integer value, a combination of `im::XXX` constants. The whole set of pixel formats is represented by colored formats, like, 16-color, 256-color and 16M-color, and by gray-scale formats, mapped to C data types - unsigned char, unsigned short, unsigned long, float and double. The gray-scale formats are subdivided to real-number formats and complex-number format; the last ones are represented by two real values per pixel, containing the real and the imaginary values.

Prima::Image can also be initialized from other formats, that it does not support, but can convert data from. Currently these are represented by a set of permutations of 32-bit RGBA format, and 24-bit BGR format. These formats can only be used in conjunction with `::data` property.

The conversions can be performed between any of the supported formats (to do so, `::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 non-integer or complex formats), the bitmap data are implicitly converted. The conversion does not change the data if the image is to be output; the conversion is performed only when the image is to be served as a drawing surface. If, by 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 `end_paint()` call, and reverts it to its previous pixel format.

There are situations, when a pixel format conversion must be made with down-sampling. One of four down-sampling methods can be selected - normal, 8x8 ordered halftoning, error diffusion, and error diffusion combined with optimized palette. These can be set to the `::conversion` property with one of `ict::XXX` constants. When there is no information loss, `::conversion` property is not used.

Another special case of conversion is a conversion with a palette.

```
$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 `::palette` property does not change bitmap pixel data, but overwrites palette information. A proper call syntax is

```
$image-> set(
   palette => $palette,
   type    => im::bpp4,
);
```

This call produces correct results, if palette pixel mapping is desired. The most power of this syntax is available when conversion is `ict::Optimized` (by default). This does not only allows remapping or downsampling to a predefined colors set, but also can be used to limit palette size to a particular number, without actual color cells values knowledge. For example, for an 24-bit image,

```
$image-> set( type => im::bpp8, palette => 32);
```

call would calculate colors in the image, compress them to a palette of 32 cells and converts to a 8-bit format.

Instead of `palette` property, `colormap` can also be used.

Data access

The pixel values can be accessed in *Prima::Drawable* style, via `::pixel` property. However, *Prima::Image* introduces several helper functions, for different aims. The `::data` property is used to set or retrieve a scalar representation of bitmap 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`.

This is the line size for the data as lined up internally in memory, however `::data` should not necessarily should be aligned like this, and can be accompanied with a write-only flag 'lineSize' if data is 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 the farthest away from the memory start; this is consistent with general Y-axis orientation in Prima drawable terminology, but might be inconvenient when importing data 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 the 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 modification with the pixels, returned by `::data`, it is not advisable unless the speed does not matter. Standalone PDL package with help of PDL::PrimaImage package, and Prima-derived IPA package provide routines for data and image analysis. *Prima::Image* itself provides only the simplest statistic information, namely: lowest and highest pixel values, pixel sum, sum of square pixels, mean, variance, and standard deviation.

Standalone usage

The image functionality can be used standalone, with all other parts of the toolkit being uninitialized. This is useful in non-interactive programs, running in evnironments with no GUI access, a cgi-script with no access to X11 display, for example. Normally, Prima fails to start in such situations, but can be told not to initialize its GUI part by explicitly operating system-dependent options. Currently, X11 implementation provides '–no-x11' option which effectively turns off X11 support.

Prima::Icon

Prima::Icon inherits all properties of *Prima::Image*, and it also provides a 1-bit depth transparency mask. This mask can also be loaded and saved into image files, if the format supports a transparency information.

Alike *Prima::Image* `::data` property, *Prima::Icon* `::mask` property provides access to the binary mask data. The mask can be updated automatically, after an icon object was subject to painting or other change. The auxiliary properties `::autoMasking` and `::maskColor` regulate mask update procedure. For example, if an icon was loaded with the color (vs. bitmap) transparency information, the binary mask will be generated anyway, but it will be also recorded that a particular color serves as a transparent indicator, so eventual conversions can rely on the color value, instead of the mask bitmap.

If an icon is drawn upon a graphic canvas, the image output is constrained to the mask. On raster displays it is typically simulated by a combination of and- and xor- operation modes, therefore attempts to put an icon with `::rop`, different from `rop::CopyPut`, usually fail.

API

Prima::Image properties

colormap @PALETTE

A color palette, used for representing 1, 4, and 8-bit bitmaps, when an image object is to be visualized. @PALETTE contains individual colors component triplets, in RGB format. For example, black-and-white monochrome image may contain colormap as `0,0xffffff`.

See also `palette`.

conversion TYPE

Selects type of dithering algorithm, when down-sampling takes place. TYPE is one of `ict::XXX` constants:

```
ict::None            - no dithering
ict::Halftone        - 8x8 ordered halftone dithering
ict::ErrorDiffusion  - error diffusion dithering with static palette
ict::Optimized       - error diffusion dithering with optimized palette
```

As an example, if a 4x4 color image with every pixel set to RGB(32,32,32), converted to a 1-bit image, the following results occur:

```
ict::None:
  [ 0 0 0 0 ]
  [ 0 0 0 0 ]
  [ 0 0 0 0 ]
  [ 0 0 0 0 ]

ict::Halftone:
  [ 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 ]
```

data SCALAR

Provides access to the bitmap data. On get-call, returns all bitmap pixels, aligned to 4-byte boundary. On set-call, stores the provided data with same alignment. The alignment can be altered by submitting 'lineSize' write-only flag to set call; the ordering of scan lines can be altered by setting 'reverse' write-only flag (see the *Data access* entry).

height INTEGER

Manages the vertical dimension of the image data. On set-call, the image data are changed accordingly to the new height, and depending on `::vScaling` property, the pixel values are either scaled or truncated.

hScaling BOOLEAN

If 1, the bitmap data will be scaled when image changes its horizontal extent. If 0, the data will be stripped or padded with zeros.

lineSize INTEGER

A read-only property, returning the length of an image row in bytes, as represented internally in memory. Data returned by `::data` property are aligned with `::lineSize` bytes per row, and setting `::data` expects data aligned with this value, unless `lineSize` is set together with `data` to indicate another alignment. See the *Data access* entry for more.

mean

Returns mean value of pixels. Mean value is `::sum` of pixel values, divided by number of pixels.

palette [@PALETTE]

A color palette, used for representing 1, 4, and 8-bit bitmaps, when an image object is to be visualized. @PALETTE contains individual color component triplets, in BGR format. For example, black-and-white monochrome image may contain palette as `[0,0,0,255,255,255]`.

See also `colormap`.

pixel (X_OFFSET, Y_OFFSET) PIXEL

Provides per-pixel access to the image data when image object is in disabled paint state. Otherwise, same as `Prima::Drawable::pixel`.

preserveType BOOLEAN

If 1, reverts the image type to its old value if an implicit conversion was called during `end_paint()`.

rangeHi

Returns maximum pixel value in the image data.

rangeLo

Returns minimum pixel value in the image data.

size WIDTH, HEIGHT

Manages dimensions of the image. On set-call, the image data are changed accordingly to the new dimensions, and depending on `::vScaling` and `::hScaling` properties, the pixel values are either scaled or truncated.

stats (INDEX) VALUE

Returns one of calculated values, that correspond to INDEX, which is 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      - sum of pixel values
is::Sum2     - sum of squares of pixel values
```

The values are re-calculated on request and cached. On set-call VALUE is stored in the cache, and is returned on next get-call. The cached values are discarded every time the image data changes.

These values are also accessible via set of alias properties: `::rangeLo`, `::rangeHi`, `::mean`, `::variance`, `::stdDev`, `::sum`, `::sum2`.

stdDev

Returns standard deviation of the image data. Standard deviation is the square root of `::variance`.

sum

Returns sum of pixel values of the image data

sum2

Returns sum of squares of pixel values of the image data

type TYPE

Governs the image pixel format type. TYPE is a combination of `im::XXX` constants. The constants are collected in groups:

Bit-depth constants provide size of pixel is bits. Their actual value is same as number of bits, so `im::bpp1` value is 1, `im::bpp4` - 4, etc. The valid constants represent 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 designate the pixel format category:

```
im::Color
im::GrayScale
im::RealNumber
im::ComplexNumber
im::TrigComplexNumber
```

Value of `im::Color` is 0, whereas other category constants represented by unique bit value, so combination of `im::RealNumber` and `im::ComplexNumber` is possible.

There 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
```

Bit depths of float- and double- derived pixel formats depend on a platform.

The groups can be masked out with 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 `::type`, but recognized within the combined set-call, like

```
$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 variance of pixel values of the image data. Variance is `::sum2`, divided by number of pixels minus square of `::sum` of pixel values.

vScaling BOOLEAN

If 1, the bitmap data will be scaled when image changes its vertical extent. If 0, the data will be stripped or padded with zeros.

width INTEGER

Manages the horizontal dimension of the image data. On set-call, the image data are changed accordingly to the new width, and depending on `::hScaling` property, the pixel values are either scaled or truncated.

Prima::Icon properties

autoMasking TYPE

Selects whether the mask information should be updated automatically with `::data` change or not. Every `::data` change is mirrored in `::mask`, using TYPE, one of `am::XXX` constants:

```
am::None          - no mask update performed
am::MaskColor     - mask update based on ::maskColor 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 majority of the corner pixels represents a transparent color. Once such color is found, the mask is generated as in `am::MaskColor` case.

When image `::data` is stretched, `::mask` is stretched accordingly, disregarding the `::autoMasking` value.

mask SCALAR

Provides access to the transparency bitmap. On get-call, returns all bitmap pixels, aligned to 4-byte boundary in 1-bit format. On set-call, stores the provided transparency data with same alignment.

maskColor COLOR

When `::autoMasking` set to `am::MaskColor`, COLOR is used as a transparency value.

Prima::DeviceBitmap properties

monochrome BOOLEAN

A read-only property, that can only be set during creation, reflects whether the system bitmap is black-and-white 1-bit (monochrome) or not. The color depth of a bitmap can be read via `get_bpp()` method; monochrome bitmaps always have bit depth of 1.

Prima::Image methods

bitmap

Returns newly created *Prima::DeviceBitmap* instance, with the image dimensions and with the bitmap pixel values copied to.

codecs

Returns array of hashes, each describing the supported image format. If the array is empty, the toolkit was set up so it can not load and save images.

See the *Prima::image-load* section for details.

This method can be called without object instance.

dup

Returns a duplicate of the object, a newly created *Prima::Image*, with all information copied to it.

extract X_OFFSET, Y_OFFSET, WIDTH, HEIGHT

Returns a newly created image object with WIDTH and HEIGHT dimensions, initialized with pixel data from X_OFFSET and Y_OFFSET in the bitmap.

get_bpp

Returns the bit depth of the pixel format. Same as `::type & im::BPP`.

get_handle

Returns a system handle for an image object.

load (FILENAME or FILEGLOB) [%PARAMETERS]

Loads image from file FILENAME or stream FILEGLOB into an object, and returns the success flag. The semantics of `load()` is extensive, and can be influenced by PARAMETERS hash. `load()` can be called either in a context of an existing object, then a boolean success flag is returned, or in a class context, then a newly created object (or `undef`) is returned. If an error occurs, `$@` variable contains the error description string. These two invocation semantics are equivalent:

```
my $x = Prima::Image-> create();
die "$@" unless $x-> load( ... );
```

and

```
my $x = Prima::Image-> load( ... );
die "$@" unless $x;
```

See the *Prima::image-load* section for details.

NB! When loading from streams on win32, mind `binmode`.

map COLOR

Performs iterative mapping of bitmap pixels, setting every pixel to `::color` property with respect to `::rop` type if a pixel equals to COLOR, and to `::backColor` property with respect to `::rop2` type otherwise.

`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 ]
```

resample SRC_LOW, SRC_HIGH, DEST_LOW, DEST_HIGH

Performs linear scaling of gray pixel values from range (SRC_LOW - SRC_HIGH) to 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);
```

save (FILENAME or FILEGLOB), [%PARAMETERS]

Stores image data into image file FILENAME or stream FILEGLOB, and returns the success flag. The semantics of `save()` is extensive, and can be influenced by PARAMETERS hash. If error occurs, `$@` variable contains error description string.

Note that when saving to a stream, `codecID` must be explicitly given in `%PARAMETERS`.

See the *Prima::image-load* section for details.

NB! When saving to streams on win32, mind `binmode`.

Prima::Image events

`Prima::Image`-specific events occur only from inside the *load* entry call, to report image loading progress. Not all codecs (currently JPEG,PNG,TIFF only) are able to report the progress to the caller. See Loading with progress indicator in *Prima::image-load* for details, the **watch_load_progress** entry in the *Prima::ImageViewer* section and the **load** entry in the *Prima::ImageDialog* section for suggested use.

HeaderReady

Called whenever image header is read, and image dimensions and pixel type is changed accordingly to accomodate image data.

DataReady X, Y, WIDTH, HEIGHT

Called whenever image data that cover area designated by X,Y,WIDTH,HEIGHT is acquired. Use `load` option `eventDelay` to limit the rate of `DataReady` event.

Prima::Icon methods

split

Returns two new *Prima::Image* objects of same dimension. Pixels in the first is are duplicated from `::data` storage, in the second - from `::mask` storage.

combine DATA, MASK

Copies information from DATA and MASK images into `::data` and `::mask` property. DATA and MASK are expected to be images of same dimension.

Prima::DeviceBitmap methods

icon

Returns a newly created *Prima::Icon* object instance, with the pixel information copied from the object.

image

Returns a newly created *Prima::Image* object instance, with the pixel information copied from the object.

get_handle

Returns a system handle for a system bitmap object.

3.6 Prima::image-load

Using image subsystem

Description

Details on image subsystem - image loading, saving, and codec managements

Loading

Simple loading

Simplest case, loading a single image would look like:

```
my $x = Prima::Image-> load( 'filename.duf');
die "$@" unless $x;
```

Image functions can work being either invoked from package, or from existing Prima::Image object, in latter case the caller object itself is changing. The code above could be also written as

```
my $x = Prima::Image-> create;
die "$@" unless $x-> load( 'filename.duf');
```

In both cases $x contains image data upon success. Error is returned into $@ variable (see perldoc perlvar for more info).

Loading from stream

`Prima::Image` can also load image 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 call can be also issued in two ways:

```
my @x = Prima::Image-> load( 'filename.duf', loadAll => 1);
die "$@" unless $x[-1];

my $x = Prima::Image-> create;
my @x = $x-> load( 'filename.duf', loadAll => 1);
die "$@" unless $x[-1];
```

In second case, the content of the first frame comes to $x and $x[0]. Sufficient check for error is whether last item of a returned array is defined. This check works also if an empty array is returned. Only this last item can be an undefined value, others are guaranteed to be valid objects.

Multiframe syntax is expressed in a set of extra hash keys. These keys are:

loadAll

Request for loading all frames that can be read from a file. Example:

```
loadAll => 1
```

index

If present, returns a single frame with index given. Example:

```
index => 8
```

map

Contains an anonymous array of frame indices to load. Valid indices are above zero, negative ones can't be counted in a way perl array indices are. Example:

```
map => [0, 10, 15..20]
```

Querying extra information

By default Prima loads image data and palette only. For any other information that can be loaded, anonymous hash 'extras' can be defined. To notify a codec that this extra information is desired, loadExtras boolean value is used. Example:

```
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 libungif would look like:

```
codecID : 1
transparentColorIndex : 1
comment : created by GIMP
frames : 18
```

'codecID' is a Prima-defined extra field, which is an index of the codec which have loaded the file. This field's value is useful for explicit indication of codec on the save request.

'frames' is also a Prima-defined extra field, with integer value set to a number of frames in the image. It might be set to -1, signaling that codec is incapable of quick reading of the frame count. If, however, it is necessary to get actual frame count, a 'wantFrames' profile boolean value should be set to 1 - then frames is guaranteed to be set to a 0 or positive value, but the request may take longer time, especially on a large file with sequential access. Real life example is a gif file with more than thousand frames. 'wantFrames' is useful in null load requests.

Multiprofile loading requests

The parameters that are accepted by load, are divided into several categories - first, those that apply to all loading process and those who apply only to a particular frame. Those who are defined by Prima, are enumerated above - loadExtras, loadAll etc. Only loadExtras, noImageData and iconUnmask are applicable to a frame, other govern the loading process. 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 codec documentation;

The parameters that applicable to any frame, can be specified separately to every desirable frame in single call. For that purpose, parameter 'profiles' is defined. 'profiles' is expected to be an anonymous array of hashes, each hash where corresponds to a request number. Example:

```
$x-> load( $f, loadAll => 1, profiles => [
     {loadExtras => 0},
     {loadExtras => 1},
]);
```

First hash there applies to frame index 0, second - to frame index 1. Note that in code

```
$x-> load( $f,
   map => [ 5, 10],
   profiles => [
     {loadExtras => 0},
     {loadExtras => 1},
]);
```

first hash applies to frame index 5, and second - to frame index 10.

Null load requests

If it is desired to peek into image, reading type and dimensions only, one should set 'noImageData' boolean value to 1. Using 'noImageData', empty objects with read type are returned, and with extras 'width' and 'height' set to 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 information about image can be loaded even without frame loading - if the codec provides such a functionality. This is the only request that cannot be issued on a package:

```
$x-> load( $f, map => [], loadExtras => 1);
```

Since no frames are required to load, an empty array is returned upon success and an array with one undefined value on failure.

Using Prima::Image descendants

If Prima needs to create a storage object, it is by default Prima::Image, or a class name of an caller object, or a package the request was issued on. This behavior can be altered using parameter 'className', which defines the class to be used for the frame.

```
my @x = Prima::Image-> load( $f,
    map => [ 1..3],
    className => 'Prima::Icon',
    profiles => [
        {},
        { className => 'Prima::Image' },
        {}
    ],
```

In this example @x will be (Icon, Image, Icon) upon success.

When loading to an Icon object, the default toolkit action is to build the transparency mask based on image data. When it is not the desired behavior, e.g., there is no explicit knowledge of image, but the image may or may not contain transparency information, `iconUnmask` boolean option can be used. When set to a `true` value, and the object is `Prima::Icon` descendant, `Prima::Icon::autoMasking` is set to `am::None` prior to the file loading. By default this options is turned off.

Loading with progress indicator

Some codecs (PNG,TIFF,JPEG) can notify the caller as they read image data. For this purpose, `Prima::Image` has two events, `onHeaderReady` and `onDataReady`. If either (or both) are present on image object that is issuing load call, and the codec supports progressive loading, these events are called. `onHeaderReady` is called when image header data is acquired, and empty image with the dimensions and pixel type is allocated. `onDataReady` is called whenever a part of image is ready and is loaded in the memory of the object; the position and dimensions of the loaded area is reported also. The format of the events is:

```
onHeaderReady $OBJECT
onDataReady   $OBJECT, $X, $Y, $WIDTH, $HEIGHT
```

`onHeaderReady` is called only once, but `onDataReady` is called as soon as new image data is available. To reduce frequency of these calls, that otherwise would be issued on every scanline loaded, `load` has parameter `eventDelay`, a number of seconds, which limits event rate. The default `eventDelay` is 0.1 .

The handling on `onDataReady` must be performed with care. First, the image must be accessed read-only, which means no transformations with image size and type are allowed. Currently there is no protection for such actions (because codec must perform these), 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 partial image as it loads. All of these share overloading of `onHeaderReady` and `onDataReady`. The simpler is to call `put_image` from inside `onDataReady`:

```
$i = Prima::Image-> new(
        onDataReady => sub {
                $progress_widget-> put_image( 0, 0, $i);
        },
);
```

but that will most probably loads heavily underlying OS-dependent conversion of image data to native display bitmap data. A more smarter, but more complex solution is to copy loaded (and only loaded) bits to a preexisting device bitmap:

```
$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 `Prima::ImageViewer` when it is setup to monitor image loading progress. See the **watch_load_progress** entry in the *Prima::ImageViewer* section for details.

Saving

Simple saving

Typical saving code will be:

```
die "$@" unless $x-> save( 'filename.duf');
```

Upon a single-frame invocation save returns 1 upon success an 0 on failure. Save requests also can be performed with package syntax:

```
die "$@" unless Prima::Image-> save( 'filename.duf',
    images => [ $x]);
```

Saving to a stream

Saving to a stream requires explicit `codecID` to be supplied. When an image is loaded with `loadExtras`, this field is always present on the image object, and is an integer that selects image encoding 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 => $png_id[0])
   or die "Cannot save:$@";
```

Multiframe saving

In multiframe invocation save returns number of successfully saved frames. File is erased though, if error occurred, 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 object hash reference 'extras', is assumed to be saved as an extra information. It is a codec's own business how it reacts on invalid and/or inacceptable information - but typical behavior is that keys that were not recognized by the codec just get ignored, and invalid values raise an error.

```
    $x-> {extras}-> {comments} = 'Created by Prima';
    $x-> save( $f);
```

Selecting a codec

Extras field 'codecID', the same one that is defined after load requests, selects explicitly a codec for an image to handle. If the codec selected is incapable of saving an error is returned. Selecting a codec is only possible with the object-driven syntax, and this information is never extracted from objects but passed to 'images' array instead.

```
    $x-> {extras}-> {codecID} = 1;
    $x-> save( $f);
```

Actual correspondence between codecs and their indices is described latter.
NB - if codecID is not given, codec is selected by the file extension.

Type conversion

Codecs usually are incapable of saving images in all formats, so Prima either converts an image to an appropriate format or signals an error. This behavior is governed by profile key 'autoConvert', which is 1 by default. 'autoConvert' can be present in image 'extras' structures. With autoConvert set it is guaranteed that image will be saved, but original image information may be lost. With autoConvert unset, no information will be lost, but Prima may signal an error. Therefore general-purpose save routines should be planned carefully. As an example the Prima::ImageDialog::SaveImageDialog code might be useful.

When the conversion takes place, Image property 'conversion' is used for selection of an error distribution algorithm, if down-sampling is required.

Appending frames to an existing file

This functionality is under design, but the common outlines are already set. Profile key 'append' (0 by default) triggers this behavior - if it is set, then an append attempt is made.

Managing codecs

Prima provides single function, Prima::Image-> codecs, which returns an anonymous array of hashes, where every hash entry corresponds to a registered codec. 'codecID' parameter on load and save requests is actually an index in this array. Indexes for a codecs registered once never change, so it is safe to manipulate these numbers within single program run.

Codec information that is contained in these hashes is divided into following parameters:

codecID

Unique integer value for a codec, same as index of the codec entry in results of `Prima::Image->codecs`;

name

codec full name, string

vendor

codec vendor, string

versionMajor and versionMinor

usually underlying library versions, integers

fileExtensions

array of strings, with file extensions that are typical to a codec. example: ['tif', 'tiff']

fileType

Description of a type of a file, that codec is designed to work with. String.

fileShortType

Short description of a type of a file, that codec is designed to work with. (short means 3-4 characters). String.

featuresSupported

Array of strings, with some features description that a codec supports - usually codecs implement only a part of file format specification, so it is always interesting to know, what part it is.

module and package

Specify a perl module, usually inside Prima/Image directory into Prima distribution, and a package inside the module. The package contains some specific functions for work with codec-specific parameters. Current implementation defines only ::save_dialog() function, that returns a dialog that allows to change these parameters. See Prima::ImageDialog::SaveImageDialog for details. Strings, undefined if empty.

canLoad

1 if a codec can load images, 0 if not

canLoadStream

1 if a codec can load images from streams, 0 otherwise

canLoadMultiple

1 if a codec can handle multiframe load requests and load frames with index more than zero. 0 if not.

canSave

1 if a codec can save images, 0 if not.

canSaveStream

1 if a codec can save images to streams, 0 otherwise

canSaveMultiple

1 if codec can save and/or append more that one frame. 0 if not.

types

Array of integers - each is a combination of im:: flags, an image type, which a codec is capable of saving. First type in list is a default one; if image type that to be saved is not in that list, the image will be converted to this default type.

loadInput

Hash, where keys are those that are accepted by Prima::Image-> load, and values are default values for these keys.

loadOutput

Array of strings, each of those is a name of extra information entry in 'extras' hash.

saveInput

Hash, where keys are those that are accepted by Prima::Image-> save, and values are default values for these keys.

3.7 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 Prima::Component, a class, especially crafted to reflect and govern properties of a system-dependent window, such as its position, hierarchy, outlook etc. Prima::Widget is mapped into the screen space as a rectangular area, with distinct boundaries, pointer and sometimes cursor, and a user-selectable input focus.

Usage

Prima::Widget class and its descendants are used widely throughout the toolkit, and, indeed provide almost all its user interaction and input-output. The notification system, explained in the *Prima::Object* section, is employed in Prima::Widget heavily, providing the programmer with unified access to the system-generated events, that occur when the user moves windows, clicks the mouse, types the keyboard, etc. Descendants of Prima::Widget use the internal, the direct method of overriding the notifications, whereas end programs tend to use the toolkit widgets equipped with anonymous subroutines (see the *Prima::Object* section for the details).

The class functionality is much more extensive comparing to the other built-in classes, and therefore the explanations are grouped in several topics.

Creation and destruction

The widget creation syntax is the same as for the other Prima objects:

```
Prima::Widget-> create(
   name => 'Widget',
   size => [ 20, 10],
   onMouseClick => sub { print "click\n"; },
   owner => $owner,
);
```

In the real life, a widget must be almost always explicitly told about its owner. The owner object is either a Prima::Widget descendant, in which case the widget is drawn inside its inferior, or the application object, and in the latter case a widget becomes top-level. This is the reason

why the `insert` syntax is much more often used, 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 Prima::Component, Prima::Widget sends `Create` notification when created (more precisely, after its init stage is finished. See the *Prima::Object* section for details). This notification is called and processed within `create()` call. In addition, another notification `Setup` is sent after the widget is created. This message is *posted*, so it is called within `create()` but processed in the application event loop. This means that the execution time of `Setup` is uncertain, as it is with all posted messages; its delivery time is system-dependent, so its use must be considered with care.

After a widget is created, it is usually asked to render its content, provided that the widget is visible. This request is delivered by means of `Paint` notification.

When the life time of a widget is over, its method `destroy()` is called, often implicitly. If a widget gets destroyed because its owner also does, it is guaranteed that the children widgets will be destroyed first, and the owner afterwards. In such situation, widget can operate with a limited functionality both on itself and its owners (see the *Prima::Object* section, **Creation** section).

Graphic content

A widget can use two different ways for representing its graphic content to the user. The first method is event-driven, when the `Paint` notification arrives, notifying the widget that it must re-paint itself. The second is the 'direct' method, when the widget generates graphic output unconditionally.

Event-driven rendering

A notification responsible for widget repainting is `Paint`. It provides a single (besides the widget itself) parameter, an object, where the drawing is performed. In an event-driven call, it is always equals to the widget. However, if a custom mechanism should be used that directly calls, for example,

```
$widget-> notify('Paint', $some_other_widget);
```

for whatever purpose, it is recommended (not required, though), to use this parameter, not the widget itself for painting and drawing calls.

The example of `Paint` callback is quite simple:

```
Prima::Widget-> create(
    ...
    onPaint => sub {
       my ( $self, $canvas) = @_;
       $canvas-> clear;
       $canvas-> text_out("Clicked $self->{clicked} times", 10, 10);
    },
    onMouseClick => sub {
       $_[0]-> {clicked}++;
       $_[0]-> repaint;
    },
);
```

The example uses 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 custom refresh of the widget's graphic content is needed, no code like `notify(q(Paint))` is used - `repaint()` method is used instead. It must be noted, that the actual execution of `Paint` callbacks might or might not occur inside the `repaint()` call. This behavior is governed by the `::syncPaint` property. `repaint()` marks the whole widget's area to be refreshed, or *invalidates* the area. For the finer gradation of the area that should be repainted, `invalidate_rect()` and `validate_rect()` pair of functions is used. Thus,

```
$x-> repaint()
```

code is a mere alias to

```
$x-> invalidate_rect( 0, 0, $x-> size);
```

call. It must be realized, that the area, passed to `invalidate_rect()` only in its ideal (but a quite often) execution case will be pertained as a clipping rectangle when a widget executes its `Paint` notification. The user and system interactions can result in exposition of other parts of a widget (like, moving windows over a widget), and the resulting clipping rectangle can be different from the one that was passed to `invalidate_rect()`. Moreover, the clipping rectangle can become empty as the result of these influences, and the notification will not be called at all.

Invalid rectangle is presented differently inside and outside the drawing mode. The first, returned by `::clipRect`, employs inclusive-inclusive coordinates, whereas `invalidate_rect()`, `validate_rect()` and `get_invalid_rect()` - inclusive-exclusive coordinates. The ideal case exemplifies the above said:

```
$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
```

As noted above, `::clipRect` property is set to the clipping rectangle of the widget area that is needed to be refreshed, and an event handler code can take advantage of this information, increasing the efficiency of the painting procedure.

Further assignments of `::clipRect` property do not make possible over-painting on the screen area that lies outside the original clipping region. This is also valid for all paint operations, however since the original clipping rectangle is the full area of a canvas, this rule is implicit and unnecessary, because whatever large the clipping rectangle is, drawing and painting cannot be performed outside the physical boundaries of the canvas.

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 `begin_paint()` is called, and, if successful, the part of the screen occupied by the widget is accessible to the drawing and painting routines.

This method is useful, for example, for graphic demonstration programs, that draw continuously without any input. Another field is the screen drawing, which is performed with Prima::Application class, that does not have `Paint` notification. Application's graphic canvas represents the whole screen, allowing over-drawing the graphic content of other programs.

The event-driven rendering method adds implicit `begin_paint()/end_paint()` brackets (plus some system-dependent actions) and is a convenience version of the direct rendering. Sometimes,

however, the changes needed to be made to a widget's graphic context are so insignificant, so the direct rendering method is preferable, because of the cleaner and terser code. As an example might serve a simple progress bar, that draws a simple colored bar. The event-driven code would be (in short, omitting many details) as such:

```
$bar = Widget-> create(
  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, ( but clumsier )
# $bar-> invalidate_rect( $bar->{progress}-10, 0,
#                 $bar->{progress}, $bar-> height);
```

And the direct driven:

```
$bar = Widget-> create( 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 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 cannot be that smart, but if the status bar is an insignificant part of the program, the trade-off of the functionality in favor to the code simplicity might be preferred.

Both methods can be effectively disabled using the paint locking mechanism. The `lock()` and `unlock()` methods can be called several times, stacking the requests. This feature is useful because many properties implicitly call `repaint()`, and if several of these properties activate in a row, the unnecessary redrawing of the widget can be avoided. The drawback is that the last `unlock()` call triggers `repaint()` unconditionally.

Geometry

Basic properties

A widget always has its position and size determined, even if it is not visible on the screen. Prima::Widget provides several properties with overlapping functionality, that govern the geometry of a widget. The base properties are `::origin` and `::size`, and the derived are `::left`, `::bottom`, `::right`, `::top`, `::width`, `::height` and `::rect`. `::origin` and `::size` operate with two integers, `::rect` with four, others with one integer value.

As the Prima toolkit coordinate space begins in the lower bottom corner, the combination of `::left` and `::bottom` is same as `::origin`, and combination of `::left`, `::bottom`, `::right` and `::top` - same as `::rect`.

When a widget is moved or resized, correspondingly two notifications occur: `Move` and `Size`. The parameters to both are old and new position and size. The notifications occur irrespectable to whether the geometry change was issued by the program itself or by the user.

Implicit size regulations

Concerning the size of a widget, two additional two-integer properties exist, `::sizeMin` and `::sizeMax`, that constrain the extension of a widget in their boundaries. The direct call that assigns values to the size properties that lie outside `::sizeMin` and `::sizeMax` boundaries, will fail - the widget extension will be adjusted to the boundary values, not to the specified ones.

Change to widget's position and size can occur not only by an explicit call to one of the geometry properties. The toolkit contains implicit rules, that can move and resize a widget corresponding to the flags, given to the `::growMode` property. The exact meaning of the `gm::XXX` flags is not given here (see description to `::growMode` in API section), but in short, it is possible with simple means to maintain widget's size and position regarding its owner, when the latter is resized. By default, and the default behavior corresponds to `::growMode` 0, widget does not change neither its size nor position when its owner is resized. It stays always in 'the left bottom corner'. 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. When a widget is expected to cover, for example, its owner's lower part and change its width in accord with the owner's, (a horizontal scroll bar in an editor window is the example), the `gm::GrowHiX` value must be used.

When this implicit size change does occur, the `::sizeMin` and `::sizeMax` do take their part as well - they still do not allow the widget's size excess their boundaries. However, this algorithm derives a problem, that is illustrated by the following setup. Imagine a widget with size-dependent `::growMode` (with `gm::GrowHiX` or `gm::GrowHiY` bits set) that must maintain certain relation between the owner's size and its own. If the implicit size change would be dependent on the actual widget size, derived as a result from the previous implicit size action, then its size (and probably position) will be incorrect after an attempt is made to change the widget's size to values outside the size boundaries.

Example: child widget has width 100, growMode set to `gm::GrowHiX` and sizeMin set to (95, 95). Its owner has width 200. If the owner widget changes gradually its width from 200 to 190 and then back, the following width table emerges:

```
                  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
```

That effect would exist if the differential-size algorithm would be implemented, - the owner changes width by 5, and the child does the same. The situation is fixed by introducing the *virtual size* term. The `::size` property is derived from virtual size, and as `::size` cannot exceed the size boundaries, virtual size can. It can even accept the negative values. With this intermediate stage added, the correct picture occurs:

```
                  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
```

Strictly speaking, the *virtual size* must be declared a read-only property, but currently it is implemented as a `get_virtual_size()` function, and it is planned to fix this discrepancy between the document and the implementation in favor of the property syntax.

Geometry managers

The concept of geometry managers is imported from Tk, which in turn is a port of Tcl-Tk. The idea behind it is that a widget size and position is governed by one of the managers, which operate depending on the specific options given to the widget. The selection is operated by `::geometry` property, and is one 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 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 selected by some intrinsic widget knowledge, and the idea is that `::geomSize` it 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 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 `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, `::size` properties are used to initialize `::geomSize`. Such design minimizes the confusion between the two properties, and also minimizes the direct usage of `::geomSize`, limiting it for selecting advisory size in widget internal code.

The geometry request size is useless under `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 the well-known problem, when a dialog window, developed with one font looks garbled on another system with another font. The relative coordinates solve the problem; the solution provides 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 a widget is actually created, are transformed in proportion between the designed and the actual font metrics.

The relative coordinates can be used only when passing the geometry properties values, and only before the creation stage, before a widget is created, because the scaling calculations perform in Prima::Widget::`profile_check_in()` method.

In order to employ the relative coordinates scheme, the owner (or the *dialog*) widget must set its `::designScale` to the font metrics and `::scaleChildren` property to 1. Widgets, created with owner that meets these requirements, 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 default implicit relative geometry schemes are applied.

The `::designScale` property is auto-inherited; its value is copied to the children widgets, unless the explicit `::designScale` was given during the widget's creation. This is used when such a child widget serves as an owner for some other grand-children widgets; the inheritance scheme allows the grand- (grand- etc) children to participate in the relative geometry scheme.

Note: it is advised to test such applications with the Prima::Stress module, which assigns a random font as the default, so the testing phase does not involve tweaking of the system settings.

Z-order

In case when two widgets overlap, one of these is drawn in full, whereas the another only partly. Prima::Widget provides management of the *Z-axis* ordering, but since Z-ordering paradigm can hardly be fit into the properties scheme, the toolkit uses methods instead.

A widget can use four query methods: `first()`, `last()`, `next()`, and `prev()`. These return, correspondingly, the first and the last widgets in Z-order stack, and the direct neighbors of a widget ($widget-> next-> prev always equals to the $widget itself, given that $widget-> next exists).

The last widget is the topmost one, the one that is drawn fully. The first is the most obscured one, given that all the widgets overlap.

Z-order can also be changed at runtime (but not during widget's creation). There are three methods: `bring_to_front()`, that sets the widget last in the order, making it topmost, `send_to_back()`, that does the reverse, and `insert_behind()`, that sets a widget behind the another widget, passed as an argument.

Changes to Z-order trigger `ZOrderChanged` notification.

Parent-child relationship

By default, if a widget is a child to a widget or window, it maintains two features: it is clipped by its owner's boundaries and is moved together as the owner widget moves. It is said also that a *child* is inferior to its *parent*. However, a widget without a parent still does have a valid owner. Instead of implementing *parent* property, the `::clipOwner` property was devised. It is 1 by default, and if it is 1, then owner of a widget is its parent, at the same time. However, when it is 0, many things change. The widget is neither clipped nor moved together with its parent. The widget become parentless, or, more strictly speaking, the screen becomes its parent. Moreover, the widget's origin offset is calculated then not from the owner's coordinates but from the screen, and mouse events in the widget do not transgress implicitly to the owner's top-level window eventual decorations.

The same results are produced if a widget is inserted in the application object, which does not have screen visualization. A widget that belongs to the application object, can not reset its `::clipOwner` value to 1.

The `::clipOwner` property opens a possibility for the toolkit widgets to live inside other programs' windows. If the `::parentHandle` is changed from its default `undef` value to a valid system window handle, the widget becomes child to this window, which can belong to any application residing on the same display. This option is dangerous, however: normally widgets never get destroyed by no reason. A top-level window is never destroyed before its `Close` notification grants the destruction. The case with `::parentHandle` is special, because a widget, inserted into an alien application, must be prepared to be destroyed at any moment. It is recommended to use prior knowledge about such the application, and, even better, use one or another inter-process communication scheme to interact with it.

A widget does not need to undertake anything special to become an 'owner'. Any widget, that was set in `::owner` property on any other widget, becomes owner automatically. Its `get_widgets()` method returns non-empty widget list. `get_widgets()` serves same purpose as Prima::Component::`get_components()`, but returns only Prima::Widget descendants.

A widget can change its owner at any moment. The `::owner` property is both readable and writable, and if a widget is visible during the owner change, it is immediately appeared under different coordinates and different clipping condition after the property change, given that its `::clipOwner` is set to 1.

Visibility

A widget is visible by default. Visible means that it is shown on the screen if it is not shadowed by other widgets or windows. The visibility is governed by the `::visible` property, and its two convenience aliases, `show()` and `hide()`.

When a widget is invisible, its geometry is not discarded; the widget pertains its position and size, and is subject to all previously discussed implicit sizing issues. When change to `::visible` property is made, the screen is not updated immediately, but in the next event loop invocation, because uncovering of the underlying area of a hidden widget, and repainting of a new-shown widget both depend onto the event-driven rendering functionality. If the graphic content must be updated, `update_view()` must be called, but there's a problem. It is obvious that if a widget is shown, the only content to be updated is its own. When a widget becomes hidden, it may uncover more than one widget, depending on the geometry, so it is unclear what widgets must be updated. For the practical reasons, it is enough to get one event loop passed, by calling `yield()` method of 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 over-shadowed by the other windows. These conditions are returned by `showing()` and `exposed()` functions. These return boolean values corresponding to the condition described. So, if a widget is 'exposed', it is 'showing' and 'visible'; `exposed()` returns always 0 if a widget is either not 'showing' or not 'visible'. If a widget is 'showing', then it is always 'visible'. `showing()` returns always 0 if a widget is invisible.

Visibility changes trigger `Hide` and `Show` notifications.

Focus

One of the key points of any GUI is that only one window at a time can possess a *focus*. The widget is *focused*, if the user's keyboard input is directed to it. The toolkit adds another layer in the focusing scheme, as often window managers do, highlighting the decorations of a top-level window over a window with the input focus.

Prima::Widget property `::focused` governs the focused state of a widget. It is sometimes too powerful to be used. Its more often substitutes, `::selected` and `::current` properties provide more respect to widget hierarchy.

`::selected` property sets focus to a widget if it is allowed to be focused, by the usage of the `::selectable` property. With this granted, the focus is passed to the widget or to the one of its (grand-) children. So to say, when 'selecting' a window with a text field by clicking on a window, one does not expect the window to be focused, but the text field. To achieve this goal and reduce unnecessary coding, the `::current` property is introduced. With all equal conditions, a widget that is 'current' gets precedence in getting selected over widgets that are not 'current'.

De-selecting, in its 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.

As within the GUI space, there can be only one 'focused' widget, so within the single widget space, there can be only one 'current' widget. A widget can be marked as a current by calling `::current` (or, identically, `::currentWidget` on the owner widget). The reassignments are performed automatically when a widget is focused. The reverse is also true: if a widget is explicitly marked as 'current', and belongs to the widget tree with the focus in one of its widgets, then the focus passed to the 'current' widget, or down to hierarchy if it is not selectable.

These relations between current widget pointer and focus allow the toolkit easily implement the focusing hierarchy. The focused widget is always on the top of the chain of its owner widgets, each of whose is a 'current' widget. If, for example, a window that contains a widget that contains a focused button, become un-focused, and then user selects the window again, then the button will become focused automatically.

Changes to focus produce `Enter` and `Leave` notifications.

Below discussed mouse- and keyboard- driven focusing schemes. Note that all of these work via `::selected`, and do not focus the widgets with `::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 are the examples. Another kind of behavior is the described above window with the text field - clicking mouse on a window focuses a text field.

Prima::Widget has the `::selectingButtons` property, a combination of mb::XXX (mouse buttons) flags. If the bits corresponding to the buttons are set, then click of this button will automatically call `::selected(1)` (not `::focused(1)`).

Another boolean property, `::firstClick` determines the behavior when the mouse button action is up to focus a widget, but the widget's top-level window is not active. The default value of `::firstClick` is 1, but if set otherwise, the user must click twice to a widget to get it focused. The property does not influence anything if the top-level window was already active when the click event occured.

Due to some vendor-specific GUI designs, it is hardly possibly to force selection of one top-level window when the click was on the another. The window manager or the OS can interfere, although this does not always happen, and produce different results on different platforms. 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 native way to navigate between the toolkit widgets are tab- and arrow- navigation. The tab (and its reverse, shift-tab) key combinations circulate the focus between the widgets in 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 provide the navigations are available and called `next_tab()` and `next_positional()`, correspondingly (see API for the details).

When `next_positional()` operates with the geometry of the widgets, `next_tab()` uses the `::tabStop` and `::tabOrder` properties. `::tabStop`, the boolean property, set to 1 by default, tells if a widget is willing to participate in tab-aided focus circulation. If it doesn't, `next_tab()` never uses it in its iterations. `::tabOrder` value is an integer, unique within the sibling widgets (sharing same owner) list, and is used as simple tag when the next tab-focus candidate is picked up. The default `::tabOrder` value is -1, which changes automatically after widget creation to a unique value.

User input

The toolkit responds to the two basic means of the user input - the keyboard and the mouse. Below described three aspects of the input handling - the event-driven, the polling and the simulated input issues. The event-driven input is the more or less natural way of communicating with the user, so when the user presses the key or moves the mouse, a system event occurs and triggers the notification in one or more widgets. Polling methods provide the immediate state of the input devices; the polling is rarely employed, 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 `notify()` call with specifically crafted parameters. It interacts with the system, so the emulation can gain the higher level of similarity to the user actions. The simulated input functions allow the notifications to be called right away, or *post* it, delaying the notification until the next event loop invocation.

Keyboard

Event-driven

Keyboard input generates several notifications, where the most important are `KeyDown` and `KeyUp`. Both have almost the same list of parameters (see API), that contain the key code, its modifiers (if any) that were pressed and an eventual character code. The algorithms that extract the meaning of the key, for example, discretion between character and functional keys etc are not described here. The reader is advised to look at Prima::KeySelector module, which provides convenience functions for keyboard input values transformations, and to the Prima::Edit and Prima::InputLine modules, the classes that use extensively the keyboard input. But in short, the key code is one of the `kb::XXX` (like, kb::F10, kb::Esc) constants, and the modifier value is a combination of the `km::XXX` (km::Ctrl, km::Shift) constants. The notable exception is kb::None value, which hints that the character code is of value. Some other `kb::XXX`-marked keys have the character code as well, and it is up to a programmer how to treat these combinations. It is advised, however, to look at the key code first, and then to the character code.

`KeyDown` event has also the *repeat* integer parameter, that shows the repetitive count how many times the key was pressed. Usually it is 1, but if a widget was not able to get its portion of events between the key presses, its value can be higher. If a code doesn't check for this parameter, some keyboard input may be lost. If the code will be too much complicated by introducing the repeat-value, one may consider setting the `::briefKeys` property to 0. `::briefKeys`, the boolean property, is 1 by default. If set to 0, it guarantees that the repeat value will always be 1, but with the price of certain under-optimization. If the core `KeyDown` processing code sees 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 needs of the other widgets that are willing to read the user input, even being out of focus. A notable example can be a button with a hot key, that reacts on the key press when the focus is elsewhere within its top-level window. `TranslateAccel` has same parameters as `KeyDown`, except the REPEAT parameter.

Such out-of-focus input is also used with built-in menu keys translations. If a descendant of Prima::AbstractMenu is in the reach of the widget tree hierarchy, then it is checked whether it contains some hot keys that match the user input. See the *Prima::Menu* section for the details. In particular, Prima::Widget has `::accelTable` property, a mere slot for an object that contains a table of hot keys mappings to custom subroutines.

Polling

The polling function for the keyboard is limited to the modifier keys only. `get_shift_state()` method returns the press state of the modifier keys, a combination of `km::XXX` constants.

Simulated input

There are two methods, corresponding to the major notifications - `key_up()` and `key_down()`, that accept the same parameters as the `KeyUp` and `KeyDown` notifications do, plus the POST boolean flag. See the *API* entry for details.

These methods are convenience wrappers for `key_event()` method, which is never used directly.

Mouse

Event-driven

Mouse notifications are send in response when the user moves the mouse, or presses and releases mouse buttons. The notifications are logically grouped in two sets, the first contains `MouseDown`, `MouseUp`, `MouseClick`, and `MouseWheel`, and the second - `MouseMove`, `MouseEnter`, end `MouseLeave`.

The first set deals with button actions. Pressing, de-pressing, clicking (and double-clicking), the turn of mouse wheel correspond to the four notifications. The notifications are sent together with the mouse pointer coordinates, the button that was touched, and the eventual modifier keys that were pressed. In addition, `MouseClick` provides the boolean flag if the click was single or double, and `MouseWheel` - the Z-range of the wheel turn. These notifications occur when the mouse event occurs within the geometrical bounds of a widget, with one notable exception, when a widget is in *capture* mode. If the `::capture` is set to 1, then these events are sent to the widget even if the mouse pointer is outside, and not sent to the widgets and windows that reside under the pointer.

The second set deals with the pointer movements. When the pointer passes over a widget, it receives first `MouseEnter`, then series of `MouseMove`, and finally `MouseLeave`. `MouseMove` and `MouseEnter` notifications provide X,Y-coordinates and modificator keys; `MouseLeave` passes no parameters.

Polling

The mouse input polling procedures are `get_mouse_state()` method, that returns combination of `mb::XXX` constants, and the `::pointerPos` two-integer property that reports the current position of the mouse pointer.

Simulated input

There are five methods, corresponding to the mouse events - `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 `mouse_event()` method, which is never used directly.

Color schemes

Prima::Drawable deals only with such color values, that can be unambiguously decomposed to their red, green and blue components. Prima::Widget extends the range of the values acceptable by its color properties, introducing the color schemes. The color can be set indirectly, without prior knowledge of what is its RGB value. There are several constants defined in `cl::` name space, that correspond to the default values of different color properties of a widget.

Prima::Widget revises the usage of `::color` and `::backColor`, the properties inherited from Prima::Drawable. Their values are widget's 'foreground' and 'background' colors, in addition to their function as template values. Moreover, their dynamic change induces the repainting of a widget, and they can be inherited from the owner. The inheritance is governed by properties `::ownerColor` and `::ownerBackColor`. While these are true, changes to owner `::color` or `::backColor` copied automatically to a widget. Once the widget's `::color` or `::backColor` are explicitly set, the owner link breaks automatically by setting `::ownerColor` or `::ownerBackColor` to 0.

In addition to these two 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 foreground and background when a widget is in the disabled state (see API, `::enabled` property). The 'hilite' values serve as the colors for representation of selection inside a 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. Examples are selections in the text widgets, or in the list boxes. The last pair, `::light3DColor` and `::dark3DColor` is used for drawing 3D-looking outlines of a widget. The purpose of all these properties is the adequate usage of the color settings, selected by the user using system-specific tools, so the program written with the toolkit would look not such different, and more or less conformant to the user's color preferences.

The additional `cl::` constants, mentioned above, represent these eight color properties. These named correspondingly, cl::NormalText, cl::Normal, cl::HiliteText, cl::Hilite, cl::DisabledText, cl::Disabled, cl::Light3DColor and cl::Dark3DColor. cl::NormalText is alias to cl::Fore, and cl::Normal - to cl::Back. Another constant set, `ci::` can be used with the `::colorIndex` property, a multiplexer for all eight color properties. `ci::` constants mimic their non-RGB `cl::` counterparts, so the call `hiliteBackColor(cl::Red)` is equal to `colorIndex(ci::Hilite, cl::Red)`.

Mapping from these constants to the RGB color representation is used with `map_color()` method. These `cl::` constants alone are sufficient for acquiring the default values, but the toolkit provides wider functionality than this. The `cl::` constants can be combined with the `wc::` constants, that represent standard widget class. The widget class is implicitly used when single `cl::` constant is used; its value is read from the `::widgetClass` property, unless one of `wc::` constants is combined with the non-RGB `cl::` value. `wc::` constants are described in the *API* entry; their usage can make call of, for example, `backColor( cl::Back)` on a button and on an input line result in different colors, because the `cl::Back` is translated in the first case into `cl::Back|wc::Button`, and in another - `cl::Back|wc::InputLine`.

Dynamic change of the color properties result in the `ColorChanged` notification.

Fonts

Prima::Widget does not change the handling of fonts - the font selection inside and outside `begin_paint()/end_paint()` is not different at all. A matter of difference is how does Prima::Widget select the default font.

First, if the `::ownerFont` property is set to 1, then font of the owner is copied to the widget, and is maintained all the time while the property is true. If it is not, the default font values read from the system.

The default font metrics for a widget returned by `get_default_font()` method, that often deals with system-dependent and user-selected preferences (see the *Additional resources* entry). Because a widget can host an eventual Prima::Popup object, it contains `get_default_popup_font()` method, that returns the default font for the popup objects. The dynamic popup font settings governed, naturally, by the `::popupFont` property. Prima::Window extends the functionality to `get_default_menu_font()` and the `::menuFont` property.

Dynamic change of the font property results in the `FontChanged` notification.

Additional resources

The resources, operated via Prima::Widget class but not that strictly bound to the widget concept, are gathered in this section. The section includes overview of pointer, cursor, hint, menu objects and user-specified resources.

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, set the pointer icon to a jagged vertical line, or indicate a moving window with a cross-arrowed pointer.

A widget can select either one of the predefined system pointers, mapped by the `cr::XXX` constant set, or supply its own pointer icon of an arbitrary size and color depth.

NB: Not all systems allow the colored pointer icons. System value under sv::ColorPointer index containing a boolean value, whether the colored icons are allowed or not.

In general, the `::pointer` property is enough for these actions. It discerns whether it has an icon or a constant passed, and sets the appropriate 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 their details in the the *API* entry sections.

Another property, which is present only in Prima::Application name space is called `::pointerVisible`, and governs the visibility of the pointer - but for all widget instances at once.

Cursor

The cursor is a blinking rectangular area, indicating the availability of the input focus in a widget. There can be only one active cursor per a GUI space, or none at all. Prima::Widget provides several cursor properties: `::cursorVisible`, `::cursorPos`, and `::cursorSize`. There are also two methods, `show_cursor()` and `hide_cursor()`, which are not the convenience shortcuts but the functions accounting the cursor hide count. If `hide_cursor()` was called three times, then `show_cursor()` must be called three times as well for the cursor to become visible.

Hint

`::hint` is a text string, that usually describes the widget's purpose to the user in a brief manner. If the mouse pointer is hovered over the widget longer than some timeout (see Prima::Application::hintPause), then a label appears with the hint text, until the pointer is drawn away. The hint behavior is governed by Prima::Application, but a widget can do two additional things about hint: it can enable and disable it by calling `::showHint` property, and it can inherit the owner's `::hint` and `::showHint` properties using `::ownerHint` and `::ownerShowHint` properties. If, for example, `::ownerHint` is set to 1, then `::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

The default functionality of Prima::Widget coexists with two kinds of the Prima::AbstractMenu descendants - Prima::AccelTable and Prima::Popup (Prima::Window is also equipped with Prima::Menu reference). The `::items` property of these objects are accessible through `::accelItems` and `::popupItems`, whereas the objects themselves - through `::accelTable` and `::popup`, correspondingly. As mentioned in the *User input* entry, these objects hook the user keyboard input and call the programmer-defined callback subroutine if the key stroke equals to one of their table values. As for `::accelTable`, its function ends here. `::popup` provides access to a context pop-up menu, which can be invoked by either right-clicking or pressing a system-dependent key combination. As a little customization, the `::popupColorIndex` and `::popupFont` properties are introduced. (`::popupColorIndex` is multiplexed to `::popupColor`, `::popupHiliteColor`, `::popupHiliteBackColor`, etc etc properties exactly like the `::colorIndex` property).

The font and color of a menu object might not always be writable. The underlying system capabilities in this area range from total inability for a program to manage the menu fonts and colors in Win32, to a sport in interactive changing menu fonts and colors in OS/2.

The Prima::Window class provides equivalent methods for the menu bar, introducing `::menu`, `::menuItems`, `::menuColorIndex` (with multiplexing) and `::menuFont` properties.

User-specified resources

It is considered a good idea to incorporate the user preferences into the toolkit look-and-feel. Prima::Widget relies to the system-specific code that tries to map these preferences as close as possible to the toolkit paradigm.

Unix version employs XRDB (X resource database), which is the natural way for the user to tell the preferences with fine granularity. Win32 and OS/2 read 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 a 'least common denominator', which is colors and fonts. `fetch_resource()` method is capable of returning any of such settings, provided it's format is font, color or a string. The method is rarely called directly.

The 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 `fetch_resource()` existing functionality, but it is believed that calling up the method for the every property for the every widget created is prohibitively expensive.

API

Properties

accelItems [ITEM_LIST]

Manages items of a Prima::AccelTable object associated with a widget. The ITEM_LIST format is 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 the group-like widgets (ComboBox, SpinEdit etc), that employ their children for visual representation.

Default value: 0

backColor COLOR

In widget paint state, reflects background color in the graphic context. In widget normal state, manages the basic background color. If changed, initiates `ColorChanged` notification and repaints the widget.

See also: `color`, `colorIndex`, `ColorChanged`

bottom INTEGER

Maintains the lower boundary of a widget. If changed, does not affect the widget height; but does so, if called in `set()` together with `::top`.

See also: `left`, `right`, `top`, `origin`, `rect`, `growMode`, `Move`

briefKeys BOOLEAN

If 1, contracts the repetitive key press events into one notification, increasing REPEAT parameter of `KeyDown` callbacks. If 0, REPEAT parameter is always 1.

Default value: 1

See also: `KeyDown`

buffered BOOLEAN

If 1, a widget `Paint` callback draws not on the screen, but on the off-screen memory instead. 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 widget draws in the usual manner.

Default value: 0

See also: `Paint`

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 are redirected to the caller widget. The call for capture might not be always granted due the race conditions between programs.

If CLIP_OBJECT widget is defined in 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, widget is centered by X and Y axis relative to its owner.

See also: `x_centered`, `y_centered`, `growMode`, `origin`, `Move`.

clipOwner BOOLEAN

If 1, a widget is clipped by its owner boundaries. It is the default and expected behavior. If clipOwner is 0, a widget behaves differently: it does not 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 a widget do not transgress to the top-level window decorations. In short, it itself becomes a top-level window, that, contrary to the one created from Prima::Window class, does not have any interference with system-dependent window stacking and positioning (and any other) policy, and is not ornamented by the window manager decorations.

Default value: 1

See the *Parent-child relationship* entry

See also: `Prima::Object` owner section, `parentHandle`

color COLOR

In widget paint state, reflects foreground color in the graphic context. In widget normal state, manages the basic foreground color. If changed, initiates `ColorChanged` notification and repaints the widget.

See also: `backColor`, `colorIndex`, `ColorChanged`

colorIndex INDEX, COLOR

Manages the basic color properties indirectly, by accessing via `ci::XXX` constant. 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, a widget (or one of its children) is marked as the one to be focused (or selected) when the owner widget receives `select()` call. Within children widgets, only one or none at all can be marked as a current.

See also: `currentWidget`, `selectable`, `selected`, `selectedWidget`, `focused`

currentWidget OBJECT

Points to a children widget, that is to be focused (or selected) when the owner widget receives `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 cursor visibility flag. 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 a font, that was used when a widget (usually a dialog or a grouping widget) was designed.

See also: `scaleChildren`, `width`, `height`, `size`, `font`

disabledBackColor COLOR

The color used to substitute `::backColor` when a widget is in its disabled state.

See also: `disabledColor`, `colorIndex`, `ColorChanged`

disabledColor COLOR

The color used to substitute `::color` when a widget is in its disabled state.

See also: `disabledBackColor`, `colorIndex`, `ColorChanged`

enabled BOOLEAN

Specifies if a widget can accept focus, keyboard and mouse events. Default value is 1, however, being 'enabled' does not automatically allow the widget become focused. Only the reverse is true - if enabled is 0, focusing can never happen.

See also: `responsive`, `visible`, `Enable`, `Disable`

font %FONT

Manages font context. Same syntax as in Prima::Drawable. If 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 widget behavior, when its owner is resized or moved. MODE can be 0 (default) or a combination of the following constants:

Basic constants

```
gm::GrowLoX      widget's left side is kept in constant
                 distance from owner's right side
gm::GrowLoY      widget's bottom side is kept in constant
                 distance from owner's top side
gm::GrowHiX      widget's right side is kept in constant
                 distance from owner's right side
gm::GrowHiY      widget's top side is kept in constant
                 distance from owner's top side
gm::XCenter      widget is kept in center on its owner's
                 horizontal axis
gm::YCenter      widget is kept in center on its owner's
                 vertical axis
gm::DontCare     widgets origin is maintained 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, a widget bypasses first mouse click on it, if the top-level window it belongs to was not activated, so selecting such a widget it takes two mouse clicks.

Default value is 1

See also: `MouseDown`, `selectable`, `selected`, `focused`, `selectingButtons`

focused BOOLEAN

Specifies whether a widget possesses the input focus or not. Disregards `::selectable` property on set-call.

See also: `selectable`, `selected`, `selectedWidget`, `KeyDown`

geomWidth, geomHeight, geomSize

Three properties that select geometry request size. Writing and reading to `::geomWidth` and `::geomHeight` is equivalent to `::geomSize`. The properies are run-time only, and behave differently under different circumstances:

- As the properties are run-time only, they can not be set in the profile, and their initial value is fetched from `::size` property. Thus, setting the explicit size is aditionally 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`. When the properties are read, though, the real size properties are not read; the values are kept separately.
- Setting the properties under Tk geometry managers cause widgets size and position changed according to the geometry manager policy.

height

Maintains the height of a widget.

See also: `width`, `growMode`, `Move`, `Size`, `get_virtual_size`, `sizeMax`, `sizeMin`

helpContext STRING

A string that binds a widget, a logical part it plays with the application and an interactive help topic. STRING format is defined as 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 used to draw alternate background areas with high contrast.

See also: `hiliteColor`, `colorIndex`, `ColorChanged`

hiliteColor COLOR

The color used to draw alternate foreground areas with high contrast.

See also: `hiliteBackColor`, `colorIndex`, `ColorChanged`

hint TEXT

A text, shown under mouse pointer if it is hovered over a widget longer than `Prima::Application::hintPause` timeout. The text shows only if the `::showHint` is 1.

See also: `hintVisible`, `showHint`, `ownerHint`, `ownerShowHint`

hintVisible BOOLEAN

If called in get-form, returns whether the hint label is shown or not. If in set-form, immediately turns on or off the hint label, disregarding the timeouts. It does regard the mouse pointer location, however, and does not turn on the hint label if the pointer is away.

See also: `hint`, `showHint`, `ownerHint`, `ownerShowHint`

left INTEGER

Maintains the left boundary of a widget. If changed, does not affect the widget width; but does so, if called in `set()` together with `::right`.

See also: `bottom`, `right`, `top`, `origin`, `rect`, `growMode`, `Move`

light3DColor COLOR

The color used to draw light shades.

See also: `dark3DColor`, `colorIndex`, `ColorChanged`

ownerBackColor BOOLEAN

If 1, the background color is synchronized with the owner's. Automatically set to 0 if `::backColor` property is explicitly set.

See also: `ownerColor`, `backColor`, `colorIndex`

ownerColor BOOLEAN

If 1, the foreground color is synchronized with the owner's. Automatically set to 0 if `::color` property is explicitly set.

See also: `ownerBackColor`, `color`, `colorIndex`

ownerFont BOOLEAN

If 1, the font is synchronized with the owner's. Automatically set to 0 if `::font` property is explicitly set.

See also: `font`, `FontChanged`

ownerHint BOOLEAN

If 1, the hint is synchronized with the owner's. Automatically set to 0 if `::hint` property is explicitly set.

See also: `hint`, `showHint`, `hintVisible`, `ownerShowHint`

ownerShowHint BOOLEAN

If 1, the show hint flag is synchronized with the owner's. Automatically set to 0 if `::showHint` property is explicitly set.

See also: `hint`, `showHint`, `hintVisible`, `ownerHint`

ownerPalette BOOLEAN

If 1, the palette array is synchronized with the owner's. Automatically set to 0 if `::palette` property is explicitly set.

See also: `palette`

origin X Y

Maintains the left and bottom boundaries of a widget relative to its owner (or to the screen if `::clipOwner` is set to 0).

See also: `bottom`, `right`, `top`, `left`, `rect`, `growMode`, `Move`

packInfo %OPTIONS

See the *Prima::Widget::pack* section

palette [@PALETTE]

Specifies array of colors, that are desired to be present into 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 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 can 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; discerns between `cr::XXX` constants and an icon. If an icon contains a hash variable `__pointerHotSpot` with an array of two integers, these integers will be treated as the pointer hot spot. In 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 a pointer icon, associated with a widget.

See also: `pointer`, `pointerIcon`, `pointerType`

pointerIcon ICON

Specifies the pointer icon, associated with a widget.

See also: `pointerHotSpot`, `pointer`, `pointerType`

pointerPos X_OFFSET Y_OFFSET

Specifies the mouse pointer coordinates relative to 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. TYPE can accept one constant of `cr::XXX` set:

```
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::User                    user-defined icon
```

All constants except `cr::User` and `cr::Default` present a system-defined pointers, their icons and hot spot offsets. `cr::User` is a sign that an icon object was specified explicitly via `::pointerIcon` property. `cr::Default` is a way to tell that a widget inherits its owner pointer type, no matter is it a system-defined pointer or a custom icon.

See also: `pointerHotSpot`, `pointerIcon`, `pointer`

popup OBJECT

Manages a Prima::Popup object associated with a widget. The purpose of the popup object is to show a context menu when the user right-clicks or selects the corresponding keyboard combination. Prima::Widget can host many children objects, Prima::Popup as well. But only the one that is set in `::popup` property will be activated automatically.

The popup object can be destroyed safely; its cancellation can be done either via `popup(undef)` or `destroy()` call.

See also: `Prima::Menu`, `Popup`, `Menu`, `popupItems`, `popupColorIndex`, `popupFont`

popupColorIndex INDEX, COLOR

Maintains eight color properties of a pop-up context menu, associated with a widget. INDEX must be one of `ci::XXX` constants (see `::colorIndex` property).

See also: `popupItems`, `popupFont`, `popup`

popupColor COLOR

Basic foreground in a popup context menu color.

See also: `popupItems`, `popupColorIndex`, `popupFont`, `popup`

popupBackColor COLOR

Basic background in a popup context menu color.

See also: `popupItems`, `popupColorIndex`, `popupFont`, `popup`

popupDark3DColor COLOR

Color for drawing dark shadings in a popup context menu.

See also: `popupItems`, `popupColorIndex`, `popupFont`, `popup`

popupDisabledColor COLOR

Foreground color for disabled items in a popup context menu.

See also: `popupItems`, `popupColorIndex`, `popupFont`, `popup`

popupDisabledBackColor COLOR

Background color for disabled items in a popup context menu.

See also: `popupItems`, `popupColorIndex`, `popupFont`, `popup`

popupFont %FONT

Maintains the font of a pop-up context menu, associated with a widget.

See also: `popupItems`, `popupColorIndex`, `popup`

popupHiliteColor COLOR

Foreground color for selected items in a popup context menu.

See also: `popupItems`, `popupColorIndex`, `popupFont`, `popup`

popupHiliteBackColor COLOR

Background color for selected items in a popup context menu.

See also: `popupItems`, `popupColorIndex`, `popupFont`, `popup`

popupItems [ITEM_LIST]

Manages items of a Prima::Popup object associated with a widget. The ITEM_LIST format is same as `Prima::AbstractMenu::items` and is described in the *Prima::Menu* section.

See also: `popup`, `popupColorIndex`, `popupFont`

popupLight3DColor COLOR

Color for drawing light shadings in a 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 a widget relative to its owner (or to the screen if `::clipOwner` is set to 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 a widget. If changed, does not affect the widget width; but does so, if called in `set()` together with `::left`.

See also: `left`, `bottom`, `top`, `origin`, `rect`, `growMode`, `Move`

scaleChildren BOOLEAN

If a widget has `::scaleChildren` set to 1, then the newly-created children widgets inserted in it will be scaled corresponding to the owner's `::designScale`, given that widget's `::designScale` is not `undef` and the owner's is not [0,0].

Default is 1.

See also: `designScale`

selectable BOOLEAN

If 1, a widget can be granted focus implicitly, or by means of the user actions. `select()` regards this property, and does not focus a widget that has `::selectable` set to 0.

Default value is 0

See also: `current`, `currentWidget`, `selected`, `selectedWidget`, `focused`

selected BOOLEAN

If called in get-mode, returns whether a widget or one of its (grand-) children is focused. If in set-mode, either simply turns the system with no-focus state (if 0), or sends input focus to itself or one of the widgets tracked down by `::currentWidget` chain.

See also: `current`, `currentWidget`, `selectable`, `selectedWidget`, `focused`

selectedWidget OBJECT

Points to a child widget, that has property `::selected` set to 1.

See also: `current`, `currentWidget`, `selectable`, `selected`, `focused`

selectingButtons FLAGS

FLAGS is a combination of `mb::XXX` (mouse button) flags. If a widget receives a click with a mouse button, that has the corresponding bit set in `::selectingButtons`, then `select()` is called.

See also: `MouseDown`, `firstClick`, `selectable`, `selected`, `focused`

shape IMAGE

Maintains the non-rectangular shape of a widget. IMAGE is monochrome Prima::Image, with 0 bits treated as transparent pixels, and 1 bits as opaque pixels.

Successive only if `sv::ShapeExtension` value is true.

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. The `::hint` property must contain non-empty string as well, if the hint label must be shown.

Default value is 1.

See also: `hint`, `ownerShowHint`, `hintVisible`, `ownerHint`

size WIDTH HEIGHT

Maintains the width and height of a widget.

See also: `width`, `height` `growMode`, `Move`, `Size`, `get_virtual_size`, `sizeMax`, `sizeMin`

sizeMax WIDTH HEIGHT

Specifies the maximal size for a widget that it is allowed to accept.

See also: `width`, `height`, `size` `growMode`, `Move`, `Size`, `get_virtual_size`, `sizeMin`

sizeMin WIDTH HEIGHT

Specifies the minimal size for a widget that it is allowed to accept.

See also: `width`, `height`, `size` `growMode`, `Move`, `Size`, `get_virtual_size`, `sizeMax`

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 value is 0.

See also: `invalidate_rect`, `repaint`, `validate_rect`, `Paint`

tabOrder INTEGER

Maintains the order in which tab- and shift-tab- key navigation algorithms select the sibling widgets. INTEGER is unique among the sibling widgets. In set mode, if INTEGER value is already taken, the occupier is assigned another unique value, but without destruction of

a queue - widgets with ::tabOrder greater than of the widget, receive their new values too. Special value -1 is accepted as 'the end of list' indicator; the negative value is never returned.

See also: `tabStop`, `next_tab`, `selectable`, `selected`, `focused`

tabStop BOOLEAN

Specifies whether a widget is interested in 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 purpose. Many Prima::Widget descendants use this property heavily - buttons, labels, input lines etc, but Prima::Widget itself does not.

top INTEGER

Maintains the upper boundary of a widget. If changed, does not affect the widget height; but does so, if called in `set()` together with `::bottom`.

See also: `left`, `right`, `bottom`, `origin`, `rect`, `growMode`, `Move`

transparent BOOLEAN

Specifies whether the background of a widget before it starts painting is of any importance. If 1, a widget can gain certain transparency look if it does not clear the background during `Paint` event.

Default value is 0

See also: `Paint`, `buffered`.

visible BOOLEAN

Specifies whether a 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 eight basic color properties. CLASS can be one of `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; any class can use any of these constants in `::widgetClass`.

See also: `map_color`, `colorIndex`

widgets @WIDGETS

In get-mode, returns list of immediate children widgets (identical to `get_widgets`). In set-mode accepts set of widget profiles, as `insert` does, as a list or an array. This way it is possible to create widget hierarchy in a single call.

width WIDTH

Maintains the width of a widget.

See also: `height growMode`, `Move`, `Size`, `get_virtual_size`, `sizeMax`, `sizeMin`

x_centered BOOLEAN

A write-only property. Once set, widget is centered by the horizontal axis relative to its owner.

See also: `centered`, `y_centered`, `growMode`, `origin`, `Move`.

y_centered BOOLEAN

A write-only property. Once set, widget is centered by the vertical axis relative to its owner.

See also: `x_centered`, `centered`, `growMode`, `origin`, `Move`.

Methods

bring_to_front

Sends a widget on top of all other siblings widgets

See also: `insert_behind`, `send_to_back`, `ZOrderChanged` ,`first`, `next`, `prev`, `last`

can_close

Sends `Close` message, and returns its boolean exit state.

See also: `Close`, `close`

client_to_screen @OFFSETS

Maps array of X and Y integer offsets from widget to screen coordinates. Returns the mapped OFFSETS.

See also: `screen_to_client`, `clipOwner`

close

Calls `can_close()`, and if successful, destroys a widget. Returns the `can_close()` result.

See also: `can_close`, `Close`

defocus

Alias for `focused(0)` call

See also: `focus`, `focused`, `Enter`, `Leave`

deselect

Alias for `selected(0)` call

See also: `select`, `selected`, `Enter`, `Leave`

exposed

Returns a boolean value, indicating whether a widget is at least partly visible on the screen. Never returns 1 if a widget has `::visible` set to 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 resource, defined by the widget hierarchy, its class, name and owner. RESOURCE_TYPE can be one of type qualificators:

```
fr::Color  - color resource
fr::Font   - font resource
fs::String - text string resource
```

Such a number of the parameters is used because the method can be called before a widget is created. CLASS_NAME is widget class string, NAME is widget name. CLASS_RESOURCE is class of 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. As many times `hide_cursor()` was called, as many time its counterpart `show_cursor()` must be called to reach the cursor's initial state.

See also: `show_cursor`, `cursorVisible`

help

Starts an interactive help viewer opened on `::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 an 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 a trailing slash, and its children widgets are assigned `::helpContext` to the topics without the manpage but the leading slash instead. If the grouping widget has an empty string " " as `::helpContext` then the help is forced to be unavailable for all the children widgets.

See also: `helpContext`

insert CLASS, %PROFILE [[CLASS, %PROFILE], ...]

Creates one or more widgets with `owner` property set to the caller widget, and returns the list of 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 a 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 a widget as 'invalid', so re-painting of the area happens. See the *Graphic content* entry.

See also: `validate_rect`, `get_invalid_rect`, `repaint`, `Paint`, `syncPaint`, `update_view`

key_down CODE, KEY = kb::NoKey, MOD = 0, REPEAT = 1, POST = 0

The method sends or posts (POST flag) 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) simulated keyboard event to the system. CODE, KEY, MOD and REPEAT are the parameters to be passed to an eventual `KeyDown` or `KeyUp` notifications. 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) 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 a widget to re-paint itself. As many times `lock()` was called, as may times its counterpart, `unlock()` must be called to enable re-painting again. Returns a boolean success flag.

See also: `unlock`, `repaint`, `Paint`, `get_locked`

map_color COLOR

Transforms `cl::XXX` and `ci::XXX` combinations into RGB color representation and returns the result. If COLOR is already in RGB format, no changes are made.

See also: `colorIndex`

mouse_click BUTTON = mb::Left, MOD = 0, X = 0, Y = 0, DBL_CLICK = 0, POST = 0

The method sends or posts (POST flag) simulated `MouseClick` event to the system. BUTTON, MOD, X, Y, and DBL_CLICK 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) 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) 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, DBL_CLICK = 0, POST = 0

The method sends or posts (POST flag) simulated mouse event to the system. BUTTON, MOD, X, Y and DBL_CLICK are the parameters to be passed to an eventual mouse notifications. COMMAND is allowed to be one of `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) 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) 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) 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, Z = 0, POST = 0

The method sends or posts (POST flag) simulated `MouseUp` event to the system. MOD, X, Y and Z 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 Z-order. If none found, undef is returned.

See also: `first`, `last`, `prev`

next_tab FORWARD = 1

Returns the next widget in the sorted by `::tabOrder` list of sibling widgets. FORWARD is a boolean lookup direction flag. If none found, the first (or the last, depending on FORWARD flag) widget is returned. Only widgets with `::tabStop` set to 1 participate.

Also used by the internal keyboard navigation code.

See also: `next_positional`, `tabOrder`, `tabStop`, `selectable`

next_positional DELTA_X DELTA_Y

Returns a sibling, (grand-)child of a sibling or (grand-)child widget, that matched best the direction specified by DELTA_X and DELTA_Y. At one time, only one of these parameters can 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 Z-order. If none found, undef is returned.

See also: `first`, `last`, `next`

repaint

Marks the whole widget area as 'invalid', so re-painting of the area happens. See the *Graphic content* entry.

See also: `validate_rect`, `get_invalid_rect`, `invalidate_rect`, `Paint`, `update_view`, `syncPaint`

responsive

Returns a boolean flag, indicating whether a widget and its owners have all `::enabled` 1 or not. Useful for fast check if a widget should respond to the user 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. OPTIONS is hash, that contains optional parameters to the scrolling procedure:

clipRect [X1, Y1, X2, Y2]

The clipping area is confined by X1, Y1, X2, Y2 rectangular area. If not specified, the clipping area covers the whole widget. Only the bits, covered by clipRect are affected. Bits scrolled from the outside of the rectangle to the inside are painted; bits scrolled from the inside of the rectangle to the outside are not painted.

confineRect [X1, Y1, X2, Y2]

The scrolling area is confined by X1, Y1, X2, Y2 rectangular area. If not specified, the scrolling area covers the whole widget.

withChildren BOOLEAN

If 1, the scrolling performs with the eventual children widgets change their positions to DELTA_X and DELTA_Y as well.

Cannot be used inside paint state.

See also: `Paint`, `get_invalid_rect`

select

Alias for `selected(1)` call

See also: `deselect`, `selected`, `Enter`, `Leave`

send_to_back

Sends a widget at bottom of all other siblings widgets

See also: `insert_behind`, `bring_to_front`, `ZOrderChanged` ,`first`, `next`, `prev`, `last`

show

Sets widget `::visible` to 1.

See also: `hide`, `visible`, `Show`, `Hide`, `showing`, `exposed`

show_cursor

Shows the cursor. As many times `hide_cursor()` was called, as many time its counterpart `show_cursor()` must be called to reach the cursor's initial state.

See also: `hide_cursor`, `cursorVisible`

showing

Returns a boolean value, indicating whether the widget and its owners have all `::visible` 1 or not.

unlock

Turns on the ability of a widget to re-paint itself. As many times `lock()` was called, as may times its counterpart, `unlock()` must be called to enable re-painting again. When last `unlock()` is called, an implicit `repaint()` call is made. Returns a boolean success flag.

See also: `lock`, `repaint`, `Paint`, `get_locked`

update_view

If any parts of a widget were marked as 'invalid' by either `invalidate_rect()` or `repaint()` calls or the exposure caused by window movements (or any other), then `Paint` notification is immediately called. If no parts are invalid, no action is performed. If a widget has `::syncPaint` set to 1, `update_view()` is always a no-operation 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 widget area covered by the rectangular area passed. If a 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 a Prima::Widget class.

See also: `font`

get_default_popup_font

Returns the default font for a Prima::Popup class.

See also: `font`

get_invalid_rect

Returns the result of successive calls `invalidate_rect()`, `validate_rect()` and `repaint()`, as a rectangular area (four integers) that cover all invalid regions in a widget. If none found, (0,0,0,0) is returned.

See also: `validate_rect`, `invalidate_rect`, `repaint`, `Paint`, `syncPaint`, `update_view`

get_handle

Returns a system handle for a widget

See also: `get_parent_handle`

get_locked

Returns 1 if a widget is in `lock()` - initiated repaint-blocked state.

See also: `lock`, `unlock`

get_mouse_state

Returns a combination of `mb::XXX` constants, reflecting the currently pressed mouse buttons.

See also: `pointerPos`, `get_shift_state`

get_parent

Returns the owner widget that clips the widget boundaries, or application object if a widget is top-level.

See also: `clipOwner`

get_parent_handle

Returns a system handle for a parent of a widget, a 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, width and height of a icon, that the system accepts as valid for a pointer. If the icon is supplied that is more or less than these values, it is truncated or padded with transparency bits, but is not stretched. Can be called with class syntax.

get_shift_state

Returns a combination of `km::XXX` constants, reflecting the currently pressed keyboard modificator buttons.

See also: `get_shift_state`

get_virtual_size

Returns virtual width and height of a widget. See the *Geometry* entry, Implicit size regulations.

See also: `width`, `height`, `size growMode`, `Move`, `Size`, `sizeMax`, `sizeMin`

get_widgets

Returns list of children widgets.

Events

Change

Generic notification, used for Prima::Widget descendants; Prima::Widget itself neither calls not uses the event. Designed to be called when an arbitrary major state of a widget is changed.

Click

Generic notification, used for Prima::Widget descendants; Prima::Widget itself neither calls not uses the event. Designed to be called when an arbitrary major action for a widget is called.

Close

Triggered by `can_close()` and `close()` functions. If the event flag is cleared during execution, these functions fail.

See also: `close`, `can_close`

ColorChanged INDEX

Called when one of widget's color properties is changed, either by direct property change or by the system. INDEX is one of `ci::XXX` constants.

See also: `colorIndex`

Disable

Triggered by a successive `enabled(0)` call

See also: `Enable`, `enabled`, `responsive`

DragDrop X Y

Design in progress. Supposed to be triggered when a drag-and-drop session started by the widget. X and Y are mouse pointer coordinates on the session start.

See also: `DragOver`, `EndDrag`

DragOver X Y STATE

Design in progress. Supposed to be called when a mouse pointer is passed over a widget during a drag-and-drop session. X and Y are mouse pointer coordinates, identical to `MouseMove` X Y parameters. STATE value is undefined.

See also: `DragDrop`, `EndDrag`

Enable

Triggered by a successive `enabled(1)` call

See also: `Disable`, `enabled`, `responsive`

EndDrag X Y

Design in progress. Supposed to be called when a drag-and-drop session is finished successfully over a widget. X and Y are mouse pointer coordinates on the session end.

See also: `DragDrop`, `DragOver`

Enter

Called when a widget receives the input focus.

See also: `Leave`, `focused`, `selected`

FontChanged

Called when a widget font is changed either by direct property change or by the system.

See also: `font`, `ColorChanged`

Hide

Triggered by a successive `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 SHOW_FLAG. The hint show or hide action fails, if the event flag is cleared during execution.

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 `kb::XXX` constants, MOD is a combination of the modificator keys pressed when the event occurred (`km::XXX`). REPEAT is 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
```

The valid `kb::` constants are grouped in several sets. Some codes are aliased, like, `kb::PgDn` and `kb::PageDown`.

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
```

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 `kb::XXX` constants, MOD is a combination of the modificator keys pressed when the event occurred (`km::XXX`).

See also: `KeyDown`, `key_up`

Leave

Called when the input focus is removed from a 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 Prima::AbstractMenu descendant, that children to a widget, and VAR_NAME is the name of the menu item that is about to be shown.

Used for making changes in the menu structures dynamically.

See also: `popupItems`

MouseClick BUTTON, MOD, X, Y, DOUBLE_CLICK

Called when a mouse click (button is pressed, and then released within system-defined interval of time) is happened in the widget area. BUTTON is one of `mb::XXX` constants, MOD is a combination of `km::XXX` constants, reflecting pressed modificator keys during the event, X and Y are the mouse pointer coordinates. DOUBLE_CLICK is a boolean flag, set to 1 if it was a double click, 0 if a single.

`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 mouse button on a widget. BUTTON is one of `mb::XXX` constants, MOD is a combination of `km::XXX` constants, reflecting the pressed modificator keys during the event, 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 is entered the area occupied by a widget (without mouse button pressed). MOD is a combination of `km::XXX` constants, reflecting the pressed modificator keys during the event, X and Y are the mouse pointer coordinates.

See also: `MouseDown`, `MouseUp`, `MouseClick`, `MouseWheel`, `MouseMove`, `MouseLeave`

MouseLeave

Occurs when the mouse pointer is driven off the area occupied by a widget (without mouse button pressed).

See also: `MouseDown`, `MouseUp`, `MouseClick`, `MouseWheel`, `MouseMove`, `MouseEnter`

MouseMove MOD, X, Y

Occurs when the mouse pointer is transported over a widget. MOD is a combination of `km::XXX` constants, reflecting the pressed modificator keys during the event, 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 mouse button on a widget. BUTTON is one of `mb::XXX` constants, MOD is a combination of `km::XXX` constants, reflecting the pressed modificator keys during the event, X and Y are the mouse pointer coordinates.

See also: `MouseDown`, `MouseClick`, `MouseWheel`, `MouseMove`, `MouseEnter`, `MouseLeave`

MouseWheel MOD, X, Y, Z

Occurs when the user rotates mouse wheel on a widget. MOD is a combination of `km::XXX` constants, reflecting the pressed modificator keys during the event, X and Y are the mouse pointer coordinates. Z is the virtual coordinate of a wheel. Typical (2001 A.D.) mouse produces Z 120-fold values.

See also: `MouseDown`, `MouseUp`, `MouseClick`, `MouseMove`, `MouseEnter`, `MouseLeave`

Move OLD_X, OLD_Y, NEW_X, NEW_Y

Triggered when widget changes its position relative to its parent, either by Prima::Widget methods or by the user. OLD_X and OLD_Y are the old coordinates of a 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 a graphic context, associated with a widget. CANVAS is the widget itself, however its usage instead of widget is recommended (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 a key or mouse combination defined for a context pop-up menu execution. By default executes the associated Prima::Popup object, if it is present. If the event flag is cleared during the execution of callbacks, the pop-up menu is not shown.

See also: `popup`

Setup

This message is posted right after `Create` notification, and comes first from the event loop. Prima::Widget does not use it.

Show

Triggered by a successive `visible(1)` call

See also: `Show`, `visible`, `showing`, `exposed`

Size OLD_WIDTH, OLD_HEIGHT, NEW_WIDTH, NEW_HEIGHT

Triggered when widget changes its size, either by Prima::Widget methods or by the user. OLD_WIDTH and OLD_HEIGHT are the old extensions of a widget, NEW_WIDTH and NEW_HEIGHT are the new ones.

See also: `Move`, `origin`, `size`, `growMode`, `sizeMax`, `sizeMin`, `rect`, `clipOwner`

TranslateAccel CODE, KEY, MOD

A distributed `KeyDown` event. Traverses all the object tree that the widget which received original `KeyDown` event belongs to. Once the event flag is cleared, the iteration stops.

Used for tracking keyboard events by out-of-focus widgets.

See also: `KeyDown`

ZOrderChanged

Triggered when a widget changes its stacking order, or Z-order among its siblings, either by Prima::Widget methods or by the user.

See also: `bring_to_front`, `insert_behind`, `send_to_back`

3.8 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.

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.

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 **geometryPropagate** 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 of the packer. 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.

3.9 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).

3.10 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

Prima::Window is a descendant of Prima::Widget class. It deals with top-level windows, the windows that are specially treated by the system. Its major difference from Prima::Widget is that instances of Prima::Window can only be inferior by the screen, not the other windows, and that the system or window manager add decorations to these - usually menus, buttons and title bars. Prima::Window provides methods that communicate with the system and hint these decorations.

Usage

A typical program communicates with the user with aid of widgets, collected upon one or more top-level windows. Prima::Widget already has all functionality required for these child-parent operations, so Prima::Window is not special in respect of widget grouping and relationship. Its usage therefore is straightforward:

```
my $w = Prima::Window-> create(
    size => [300,300],
    text => 'Startup window',
);
```

There are more about Prima::Window in areas, that it is specifically designed to - the system window management and the dialog execution.

System window management

As noted before, top-level windows are special for the system, not only in their 'look', but also in 'feel': the system adds specific functions to the windows, aiding the user to navigate through the desktop. The system ofter dictates the size and position for windows, and some times these rules are hard or even impossible to circumvent. This document will be long if it would venture to describe the features of different window management systems, and the task would be never accomplished - brand new window managers emerge every month, and the old change their behavior in an unpredictable way. The only golden rule is to never rely on the behavior of one window manager, and test programs with at least two.

The Prima toolkit provides simple access to buttons, title bar and borders of a window. Buttons and title bar are managed by the `::borderIcons` property, and borders by the `::borderStyle` property. These operate with set of predefined constants, `bi::XXX` and `bs::XXX`, correspondingly.

The button constants can be combined with each other, but not all combinations may be granted by the system. The same is valid also for the border constant, except that they can not be combined - the value of `::borderStyle` is one of the integer constants.

There are other hints that the toolkit can set for a window manager. The system can be supplied with an icon that a window is bound to; the icon dimensions are much different, and although can be requested via `sv::XIcon` and `sv::YIcon` system values, the `::icon` property scales the image automatically to the closest system-recognizable dimension. The window icon is not shown by the toolkit, it is usually resides in the window decorations and sometimes on a task bar, along with the window's name. The system can be hinted to not reflect the window on the task bar, by setting the `::taskListed` property to 0.

Another issue is the window positioning. Usually, if no explicit position was given, the window is positioned automatically by the system. The same is valid for the size. But some window managers bend it to the extreme - for example, default CDE setup force the user to set newly created windows' positions explicitly. However, there is at least one point of certainty. Typically, when the initial size and/or position of a top-level window are expected to be set by the system, the `::originDontCare` and `::sizeDontCare` properties can be set to 1 during window creation. If these set, the system is asked to size/position a window regarding its own windowing policy. The reverse is not always true, unfortunately. Either if these properties set to 0, or explicit size or positions are given, the system is hinted to use these values instead, but this does not always happen. Actually, this behavior is expected by the user and often does not get even noticed as something special. Therefore it is a good practice to test a top-level windowing code with several window managers.

There are different policies about window positioning and sizing; some window managers behave best when the position is given to the window with the system-dependent decorations. It is hardly can be called a good policy, since it is not possible to calculate the derived window coordinates with certainty. This problem results in that it is impossible to be sure about window position and size before these are set explicitly. The only, not much efficient help the toolkit can provide is the property pair `::frameOrigin` and `::frameSize`, which along with `::origin` and `::size` reflect the position and size of a window, but taking into account the system-dependent decorations.

Dialog execution

Method of Prima::Window, `execute()` brings a window in a modal state on top of other toolkit windows, and returns after the window is dismissed in one or another way. This method is special as it is an implicit event loop, similar to

```
run Prima;
```

code. The event flow is not disrupted, but the windows and widgets that do not belong to the currently executed, the 'modal' window group can not be activated. There can be many modal windows on top of each other, but only one is accessible. As an example a message box can be depicted, a window that prevents the user to work with the application windows until dismissed. There can be other message boxes on top of each other, preventing the windows below from operation as well. This scheme is 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 is distinct and contains its own windows. An example analogy is when several independent applications run with modal message boxes being activated. This scheme, however, can not be achieved with single execute()-like call without creating interlocking conditions. The shared model call, `execute_shared()`, inserts the window into the shared modal stack, activates the window and returns immediately.

The both kinds of modal windows can coexist, but the exclusive windows prevents the shared from operation; while there are exclusive windows, the shared have same rights as the usual windows.

The stacking order for these two models is slightly different. A window after execute() call is set on 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; a window after execute_shared() call is added to a shared window stack, to the one the window's owner belongs to. The shared window stacks are rooted in so-called modal horizons, windows with boolean property `::modalHorizon` set to `true`. The default horizon is `::application`.

A window in modal state can return to the normal (non-modal) state by calling `end_modal()` method. The window is then hidden and disabled, and the windows below are accessible to the user. If the window was in the exclusive modal state, the execute() call is finished and returns the exit code, the value of `::modalResult` property. There two shortuct methods that end modal state, setting `::modalResult` to the basic 'ok' and 'not ok' code, correspondingly `ok()` and `cancel()` methods. Behavior of `cancel()` is identical to when the user closes the modal window by clicking the system close button, pressing Escape key, or otherwise cancelling the dialog execution. `ok()` 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, `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, that manages the menu items of a `::menu` object of the *Prima::Menu* section class, arrange 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 case of Prima::Window is being displayed as menu bar.

NB: A window can be an owner for several menu objects and still do not have a menu bar displayed, if no menu objects are marked as selected.

Prima::Dialog

Prima::Dialog, a descendant from Prima::Window, introduces no new functionality. It has its default values adjusted so the colors use more appropriate system colors, and hints the system that the outlook of a 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 Prima::Window, which overloads `on_destroy` notification and calls `$application->close` inside it. The purpose of declaration of a separate class for such a trifle difference is that many programs are designed under a paradigm where these is a main window, which is most 'important' to the user. As such consruct is used more often than any other, it is considered an optimization to write

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

rather than

```
Prima::Window-> create( ..., onDestroy => sub { $::application-> close })
```

, although these lines are equivalent.

API

Properties

borderIcons INTEGER

Hints the system about window's decorations, by selecting the combination of `bi::XXX` constants. The constants are:

```
bi::SystemMenu  - system menu button and/or close button
                  ( usually with icon ) is shown
bi::Minimize    - minimize button
bi::Maximize    - maximize ( and eventual restore )
bi::TitleBar    - window title
bi::All         - all of the above
```

Not all systems respect these hints, and many systems provide more navigating decoration controls than these.

borderStyle STYLE

Hints the system about window's border style, by selecting one of `bs::XXX` constants. The constants are:

```
bs::None      - no border
bs::Single    - thin border
bs::Dialog    - thick border
bs::Sizeable  - thick border with interactive resize capabilities
```

`bs::Sizeable` is an 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 disallow interactive resizing.

Not all systems recognize all these hints, although many recognize interactive resizing flag.

frameHeight HEIGHT

Maintains the height of a window, including the window decorations.

frameOrigin X_OFFSET, Y_OFFSET

Maintains the left X and bottom Y boundaries of a window's decorations relative to the screen.

frameSize WIDTH, HEIGHT

Maintains the width and height of a window, including the window decorations.

frameWidth WIDTH

Maintains the width of a window, including the window decorations.

icon OBJECT

Hints the system about an icon, associated with a window. If OBJECT is `undef`, the system-default icon is assumed.

See also: `ownerIcon`

menu OBJECT

Manages a Prima::Menu object associated with a window. Prima::Window can host many Prima::Menu objects, but only the one that is set in `::menu` property will be seen as a menu bar.

See also: `Prima::Menu`, `menuItems`

menuColorIndex INDEX, COLOR

Maintains eight color properties of a menu, associated with a window. INDEX must be one of `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

Color for drawing dark shadings in menus.

See also: `menuItems`, `menuColorIndex`, `menuFont`, `menu`

menuDisabledColor COLOR

Foreground color for disabled items in menus.

See also: `menuItems`, `menuColorIndex`, `menuFont`, `menu`

menuDisabledBackColor COLOR

Background color for disabled items in menus.

See also: `menuItems`, `menuColorIndex`, `menuFont`, `menu`

menuFont %FONT

Maintains the font of a menu, associated with a window.

See also: `menuItems`, `menuColorIndex`, `menu`

menuHiliteColor COLOR

Foreground color for selected items in menus.

See also: `menuItems`, `menuColorIndex`, `menuFont`, `menu`

menuHiliteBackColor COLOR

Background color for selected items in menus.

See also: `menuItems`, `menuColorIndex`, `menuFont`, `menu`

menuItems [ITEM_LIST]

Manages items of a Prima::Menu object associated with a window. The ITEM_LIST format is same as `Prima::AbstractMenu::items` and is described in the *Prima::Menu* section.

See also: `menu`, `menuColorIndex`, `menuFont`

menuLight3DColor COLOR

Color for drawing light shadings in menus.

See also: `menuItems`, `menuColorIndex`, `menuFont`, `menu`

modalHorizon BOOLEAN

Reflects if a window serves as root to the shared modal window stack. A window with `::modalHorizon` set to 1 in shared modal state 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 a single window stack.

modalResult INTEGER

Maintains a custom integer value, returned by `execute()`. Historically it is one of `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
```

NB: These constants are defined so they can be bitwise-or'ed, and *Prima::MsgBox* package uses this feature, where one of its functions parameters is a combination of `mb::` constants.

onTop BOOLEAN

If set, the window is hinted to stay on top of all other windows.

Default value: 0

ownerIcon BOOLEAN

If 1, the icon is synchronized with the owner's. Automatically set to 0 if `::icon` property is explicitly set. Default value is 1, so assigning an icon to $::application spawns the icon to all windows.

taskListed BOOLEAN

If set to 0, hints the system against reflecting existence of a window into a system task bar, or a top-level window list, or otherwise lower the window's value before the other windows. If 1, does not hint anything.

Default value: 1

windowState STATE

A three-state property, that governs the state of a window. STATE can be one of three `ws::XXX` constants:

```
ws::Normal
ws::Minimized
ws::Maximized
```

There can be more or less, or other window states provided by the system, but these three were chosen as a 'least common denominator'. The property can be changed either by 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()`.

See also: `WindowState`

Methods

cancel

A standard method to dismiss a modal window with `mb::Cancel` result. The effect of calling this method is equal to when the user selects a 'close window' action with system-provided menu, button or other tool.

See also: `ok`, `modalResult`, `execute`, `execute_shared`

end_modal

If a window is in modal state, the `EndModal` notification is activated. Then the window is returned from the modal state, gets hidden and disabled. If the window was on top in the exclusive modal state, the last called `execute()` function 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

A window is turned to the exclusive modal state and is put on top of 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 queue before the INSERT_BEFORE window. The window is showed and enabled, if necessary, and `Execute` notification is triggered.

The function is returned when a window is dismissed, or if the system-dependent 'exit'-event is triggered by the user (the latter case falls through all execute() calls and terminates `run Prima;` call, exiting gracefully).

execute_shared INSERT_BEFORE = undef

A window is turned to the shared modal state and is put on top of non-modal windows in the stack of its `::modalHorizon`. A window with `::modalHorizon` set to 1 starts its own stack, independent of all other window stacks.

By default, if INSERT_BEFORE object is undef, the window is also put on top of other shared-modal windows in its stack. If INSERT_BEFORE is one of the shared-modal windows in its stack, the window is placed in queue before the INSERT_BEFORE window.

The window is showed and enabled, if necessary, and `Execute` notification is triggered.

The function is returned immediately.

get_default_menu_font

Returns the default font for a Prima::Menu class.

get_modal

Returns one of three constants, reflecting the modal state of a window:

```
mt::None
mt::Shared
mt::Exclusive
```

Value of `mt::None` is 0, so result of get_modal() can be also treated as a boolean value, if only the fact of modality is needed to check.

get_modal_window MODALITY_TYPE = mt::Exclusive, NEXT = 1

Returns a 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 a boolean flag, selecting the lookup direction; if it is 1, the 'upper' window is returned, if

0, the 'lower' one (in a simple case when window A is made modal (executed) after modal window B, the A window is the 'upper' one).

If a window has no immediate modal relations, `undef` is returned.

maximize

Maximizes window. A shortcut for `windowState(ws::Maximized)`.

minimize

Minimizes window. A shortcut for `windowState(ws::Minimized)`.

ok

A standard method to dismiss a modal window with `mb::OK` result. Typically the effect of calling this method is equal to when the user presses the enter key of a modal window, signaling that the default action is to be taken.

See also: `cancel`, `modalResult`, `execute`, `execute_shared`

restore

Restores window to normal state from minimized or maximized state. A shortcut for `windowState(ws::Normal)`.

Events

Activate

Triggered when a window is activated by the user. Activation mark is usually resides on a window that contains keyboard focus, and is usually reflected by highlighted system decorations.

The toolkit does not provide standalone activation functions; `select()` call is used instead.

Deactivate

Triggered when a window is deactivated by the user. Window is usually marked inactive, when it contains no keyboard focus.

The toolkit does not provide standalone de-activation functions; `deselect()` call is used instead.

EndModal

Called before a window leaves modal state.

Execute

Called after a window enters modal state.

WindowState STATE

Triggered when 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.

3.11 Prima::Clipboard

GUI interprocess data exchange

Description

Prima::Clipboard class is a descendant of Prima::Component. It serves as an interface to the specific data storage, called clipboard, visible to all clients of one GUI space. The system clipboard is intended for the exchange of information of an arbitrary type between graphic applications.

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 the easier communication, the system clipboard has one 'format' field, that 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 possibility that the receiving application recognizes the data.

Different systems provide spectrum of predefined data types, but the toolkit uses only three of these - 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, and the three predefined formats are represented by `'Text'`, `'UTF8'`, and `'Image'` string constants.

The most frequent usage of Prima::Clipboard is to preform 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);
```

This simplistic code hides other aspects of Prima::Clipboard class.

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 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, code for fetching and storing multi-format data is somewhat different. Clipboard is viewed as a shared system resource, and have 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 the clipboard locking is advisory, and any process can grab clipboard at any time) .

`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 said:

```
 # 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 a program:

```
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;
}
```

Custom formats

Once registered, all processes in a GUI space can access the data by this format. The registration must take place also if a Prima-driven program needs to read data in a format, defined by an another program. In either case, the duplicate registration is a valid event. When no longer

needed, a format can be de-registered. It is not a mandatory action, however - the toolkit cleans up before exit. Moreover, the system maintains a reference counter on the custom-registered formats; de-registering does not mean deletion, thus. If two processes use a custom format, and one exits and re-starts, it still can access the data in the same format, registered by its previous incarnation.

Unicode

In real life, application often interchange text in both ascii and utf8, leaving the 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, widgets implement their own pasting scheme. To avoid hacking widget code, usage of `text` property is advised instead of indicating `'Text'` and `'UTF8'` constants. This method is used in standard widgets, and is implemented so the programmer can reprogram its default action by overloading `PasteText` notification of `Prima::Application` (see the **PasteText** entry in the *Prima::Application* section).

The default action of `PasteText` is to query first if `'Text'` format is available, and if so, return the ascii text scalar. If `Prima::Application::wantUnicodeInput` is set, `'UTF8'` format is checked before resorting to `'Text'`. It is clear that this scheme is not the only possibly needed, for example, an application may want to ignore ASCII text, or, ignore UTF8 text but have `Prima::Application::wantUnicodeInput` set, etc.

API

Properties

image OBJECT

Provides access to an image, stored in the system clipboard. In get-mode call, return `undef` if no image is stored.

text STRING

Provides access to text stored in the system clipboard. In get-mode call, return `undef` if no text information is present.

Methods

clear

Deletes all data from clipboard.

close

Closes the open/close brackets. open() and close() can be called recursively; only the last close() removes the actual clipboard locking, so other processes can use it as well.

deregister_format FORMAT_STRING

De-registers a previously registered data format. Called implicitly for all not de-registered format before a clipboard object is destroyed.

fetch FORMAT_STRING

Returns the data of FORMAT_STRING data format, if present in the clipboard. Depending on FORMAT_STRING, data is either text string for `'Text'` format, Prima::Image object for `'Image'` format and a binary scalar value for all custom formats.

format_exists FORMAT_STRING

Returns a boolean flag, showing whether FORMAT_STRING format data is present in the clipboard or not.

get_handle

Returns a system handle for a clipboard object.

get_formats

Returns 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. There is no way to see if a format, not registered before, is present.

get_registered_formats

Returns array of strings, each representing a registered format. `Text` and `Image` are returned also.

get_standard_clipboards

Returns 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.

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.

register_format FORMAT_STRING

Registers a data format under FORMAT_STRING string ID, 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 data format. Depending of FORMAT_STRING, SCALAR 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
```

NB. All custom formats treated as a 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 in 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 *CPAN* collection provides the system-independent conversion routines.

3.12 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]) }],
        ]],
        [],
        [ '~Help' => [
           [ 'Show help' => sub { $::application-> open_help($0); }],
        ]],
     ],
);

sub open_file
{
    # enable 'save' menu item
    $window-> menu-> save_file-> enable;
}

$window-> popupItems( $window-> menuItems);
```

Description

The document describes interfaces of Prima::AbstractMenu class, and its three descendants - Prima::Menu, Prima::Popup, and Prima::AccelTable, all aimed at different targets. Prima::AbstractMenu is a descendant of Prima::Component class, and its specialization is 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 as 'menu classes' and 'menu objects'), is handling of a complex structure, contained in `::items` property. This property is special in that its structure is a tree-like array of scalars, each of whose 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 '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 shortcuts, that imply default or special cases. These base six fields are:

Menu item name

A string identifier. Menu items can be accessed individually by their names, and the following fields can be managed by calling elemental properties, that require an item name. If not given, or empty, item name is assigned a string in a form '#ID' where ID is the unique integer value within the menu object.

IDs are set for each menu item, disregarding whether they have names or not. Any menu item can be uniquely identifed by its ID value, by supplying the '#ID' string, in the same fashion as named menu items. When creating or copying menu items, names in format '#ID' are not accepted, 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 explicitly set names are preserved. Since the anonymous menu items do not have name, their auto-generated names change also.

If the name is prepended by '-' or '*' characters, or both, these are not treated as part of the name but as indicator that the item is disabled ('-' character) or checked ('*' character). This syntax is valid only for `::items` and `insert()` functions, not for `set_variable()` method.

Menu text / menu image

A non-separator menu item can be visualized either as a text string or an image. These options are exclusive to each other, and therefore occupy same field. Menu text is an arbitrary string, with with `~` (tilde) quoting for a shortcut character, that the system uses as a hot key during menu navigation. Menu image is a the *Prima::Image* section object of no particular color space and dimensions.

Menu text in menu item is accessible via the `::text` property, and menu image via the `::image` property. These can not accept or return sensible arguments simultaneously.

Accelerator text

An alternate text string, appearing together with a menu item or a menu image, usually serving as a description to the hot key, associated with a menu item. For example, if a hot key to a menu item is combination of 'enter' and 'control' keys, then usually accelerator text is 'Ctrl+Enter' string.

Accelerator text in menu item is accessible via `::accel` property.

NB: There is `Prima::KeySelector::describe` function, that converts a key value to a string in human-readable format.

Hot key

An integer value, combined from either `kb::XXX` constant or a character index with modificator key values (`km::XXX` constant). This representation format is not that informative as 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 is translated to INTEGER format by `translate_key()` method. The reverse operation is not needed for `Prima::AbstractMenu` functionality and is performed by `Prima::KeySelector::translate_codes` method.

The integer value can be given in a some more readable format when submitting to `::items`. Character and F-keys (from F1 to F16) can be used literally, without `kb::` prepending, and the modificator keys can be hinted as prefix characters: km::Shift as '#', km::Ctrl as '^' and km::Alt as '@'. This way, combination of 'control' and 'G' keys can be expressed as `'^G'` literal, and 'control'+'shift'+'F10' - as `'^#F10'`.

Hot key in menu item is accessible via `::key` property. The property does accept literal key format, described above.

A literal key string can be converted to an integer value by `translate_shortcut` method.

When the user presses the key combination, that matches to hot key entry in a menu item, its action is triggered.

Action

Every non-separator and non-submenu item is destined to perform an action. The action can be set either as an anonymous sub, or as string with name of a method on the owner of a menu object. Both have their niche of usage, and both are supplied with three parameters, when called - the owner of a menu object, the menu object itself and the name of a menu item, that triggered the action.

Action scalar in menu item is accessible via `::action` property.

User data

At last, a non-separator and non-submenu menu item can hold an arbitrary scalar value, the 'user data' field. The toolkit does not use this field, leaving that to the programmer.

User data scalar in menu item is accessible via `::data` property.

Syntax of `::items` does not provide 'disabled' and 'checked' states for a menu item as separate fields. These states can be set by using '-' and '*' prefix characters, as described above, in the *Menu item name* entry. They also can be assigned on per-item basis via `::enabled` and `::checked` properties.

All these fields qualify a most common menu item, that has text, shortcut key and an action - a 'text item'. However, there are also two other types of menu items - a sub-menu and separator. The type of a menu items can not be changed except by full menu item tree change functions (`::items`, `remove()`, `insert()`.

Sub-menu item can hold same references as text menu item does, except 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, syntax of `::items` can be more elaborated and shown as

```
$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 ],
      ...
   ] ],
   ...
]);
```

Separator items do not hold any fields, except name. Their purpose is to hint a logical division of menu items by the system, which visualizes them usually as non-selectable horizontal lines.

In menu bars, the first separator item met by parser is treated differently. It serves as a hint, that the following items must be shown in the right corner of a 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 bars, or ignored.

With these menu items types and fields, it is possible to construct the described above menu description arrays. An item description array can hold from 0 to 6 scalars, and each combination is treated differently.

six - [NAME, TEXT/IMAGE, ACCEL, KEY, ACTION/SUBMENU, DATA]

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 six-scalar syntax, but without DATA field. If DATA is skipped it is `undef` by default.

four [TEXT/IMAGE, ACCEL, KEY, ACTION/SUBMENU]

Same as five-scalar syntax, but without NAME field. When NAME is skipped it is assigned to an unique string within menu object.

three [NAME, TEXT/IMAGE, ACTION/SUBMENU]

Same as five-scalar syntax, but without ACCEL and KEY fields. KEY is `kb::NoKey` by default, so no keyboard combination is bound to the item. Default ACCEL value is an empty string.

two [TEXT/IMAGE, ACTION/SUBMENU]

Same as three-scalar syntax, but without NAME field.

one and zero [ ]

Both empty and 1-scalar arrays indicate a separator menu item. In case of 1-scalar syntax, the scalar value is ignored.

As an example of all above said, a real-life piece of code is exemplified:

```
$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 main menu opens
    [ "~Clusters" => [              # right-adjacent part
      [ "*".checker =>  "Checking Item"   => "Check"     ],
      [],
      [ "-".slave   =>  "Disabled state"   => "PrintText"],
      ...
    ]]
] );
```

The code is stripped from 'menu.pl' from 'examples' directory in the toolkit installation. The reader is advised to run the example and learn the menu mechanics.

Prima::MenuItem

As described above, text and sub-menu items can be managed by elemental properties - `::accel`, `::text`, `::image`, `::checked`, `::enabled`, `::action`, `::data`. All these, plus some other methods can be called in an alternative way, resembling name-based component calls of the *Prima::Object* section. A code

```
$menu-> checked('CheckerMenuItem', 1);
```

can be re-written as

```
$menu-> CheckerMenuItem-> checked(1);
```

Name-based call substitutes Prima::MenuItem object, created on the fly. Prima::MenuItem class shares same functions of Prima::AbstractMenu, that handle individual menu items.

Prima::Menu

Objects, derived from Prima::Menu class are used to tandem Prima::Window objects, and their items to be shown as menu bar on top of the window.

Prima::Menu is special in that its top-level items visualized horizontally, and in behavior of the top-level separator items (see above, the *Menu items* entry).

If `::selected` is set to 1, then a menu object is visualized in a window, otherwise it is not. This behavior allows window to host multiple menu objects without clashing. When a Prima::Menu object gets 'selected', it displaces the previous 'selected' menu Prima::Menu object, and its items are installed into the visible menu bar. Prima::Window property `::menu` then points to the menu object, and `::menuItems` is an alias for `::items` menu class property. Prima::Window's properties `::menuFont` and `::menuColorIndex` are used as visualization hints.

Prima::Menu provide no new methods or properties.

Prima::Popup

Objects, derived from Prima::Popup class are used together with Prima::Widget objects. Menu items are visualized when the user pressed the pop-up key or mouse buttons combination, in response to Prima::Widget's `Popup` notification.

If `::selected` is set to 1, then a menu object is visualized in the system pop-up menu, otherwise it is not. This behavior allows widget to host multiple menu objects without clashing. When a Prima::Popup object gets 'selected', it displaces the previous 'selected' menu Prima::Popup object. Prima::Widget property `::popup` then points to the menu object, and `::popupItems` is an alias for `::items` menu class property. Prima::Widget's properties `::popupFont` and `::popupColorIndex` are used as visualization hints.

A Prima::Popup object can be visualized explicitly, by means of `popup` method. The implicit visualization by the user is happened only if the `::autoPopup` property is set to 1.

Prima::Popup provides new `popup` method and new `::autoPopup` property.

Prima::AccelTable

This class is destined for a more limited functionality than Prima::Menu and Prima::Popup, primarily for mapping key strokes to predefined actions. Prima::AccelTable objects are never visualized, and consume no system resources, although full menu item management syntax is supported.

If `::selected` is set to 1, then it displaces the previous 'selected' menu Prima::AccelTable object. Prima::Widget property `::accelTable` then points to the menu object, and `::accelItems` is an alias for `::items` menu class property.

Prima::AccelTable provide no new methods or properties.

API

Properties

accel NAME, STRING / Prima::MenuItem::accel STRING

Manages accelerator text for a menu item. NAME is name of the menu item.

action NAME, SCALAR / Prima::MenuItem::action SCALAR.

Manages action for a menu item. NAME is name of the menu item. SCALAR can be either an anonymous sub or a method name, defined in the menu object owner's name space. Both called with three parameters - the owner of a menu object, the menu object itself and the name of the menu item.

autoPopup BOOLEAN

Only in Prima::Popup

If set to 1 in selected state, calls `popup()` action in response to `Popup` notification, when the user presses the default key or mouse button combination.

If 0, the pop-up menu can not be executed implicitly.

Default value: 1

checked NAME, BOOLEAN / Prima::MenuItem::checked BOOLEAN

Manages 'checked' state of a menu item. If 'checked', a menu item visualized with a distinct check-mark near the menu item text or image. Its usage with sub-menu items is possible, although discouraged.

NAME is name of the menu item.

data NAME, SCALAR / Prima::MenuItem::data SCALAR

Manages the user data scalar.

NAME is name of the menu item. SCALAR can be any scalar value, the toolkit does not use this property internally.

enabled NAME, BOOLEAN / Prima::MenuItem::enabled BOOLEAN

Manages 'enabled' state of a menu item. If 'enabled' is 0, a menu item visualized with grayed or otherwise dimmed color palette. If a sub-menu item is disabled, whole sub-menu is inaccessible.

NAME is name of the menu item.

image NAME, OBJECT / Prima::MenuItem::image OBJECT

Manages the image, bound with a menu item. 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).

`::image` and `::text` are mutually exclusive menu item properties, and can not be set together, but a menu item can change between image and text representation at run time by calling these properties.

NAME is name of the menu item.

items SCALAR

Manages the whole menu items tree. SCALAR is a multi-level anonymous array structure, with syntax described in the *Menu items* entry.

`::items` is an ultimate tool for reading and writing the menu items tree, but often it is too powerful, so there are elemental properties `::accel`, `::text`, `::image`, `::checked`, `::enabled`, `::action`, `::data` declared, that handle menu items individually.

key NAME, KEY / Prima::MenuItem::key KEY

Manages the hot key combination, bound with a menu item. Internally KEY is kept as an integer value, and get-mode call returns integers only, but set-mode accepts the literal key format - like, '^C', 'F5' strings.

NAME is name of the menu item, KEY is an integer value.

selected BOOLEAN

If set to 1, menu object is granted extra functionality from a window or widget owner object. Different Prima::AbstractMenu descendant provided with different extra functionalities. In *Usage* section, see the *Prima::Menu* section, the *Prima::Popup* section and the *Prima::AccelTable* section.

Within each menu class, only one menu object can be selected for its owner.

If set to 0, the only actions performed are implicit hot-key lookup when on `KeyDown` event.

Default value: 1

text NAME, STRING / Prima::MenuItem::text STRING

Manages the text, bound with a menu item. STRING is an arbitrary string, with '~' (tilde) quotation of a hot key character. The hot key character is only used when keyboard navigation of a pop-up or a pull-down menu is performed; it has no influence outside menu sessions.

`::text` and `::image` are mutually exclusive menu item properties, and can not be set together, but a menu item can change between image and text representation at run time by calling these properties.

Methods

check NAME / Prima::MenuItem::check

Alias for `checked(1)`. Sets menu item in checked state.

disable NAME / Prima::MenuItem::disable

Alias for `enabled(0)`. Sets menu item in disabled state.

enabled NAME / Prima::MenuItem::enabled

Alias for `enabled(1)`. Sets menu item in enabled state.

get_handle

Returns a system-dependent menu handle.

NB: Prima::AccelTable use no system resources, and this method returns its object handle instead.

has_item NAME

Returns boolean value, whether the menu object has a menu item with name NAME.

insert ITEMS, ROOT_NAME, INDEX

Inserts menu item inside existing item tree. ITEMS has same syntax as `::items`. ROOT_NAME is the name of a 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, which the newly inserted items would possess after the insertion. INDEX 0 indicates the beginning, thus.

Returns no value.

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 location near (X_OFFSET,Y_OFFSET) pixel on the screen, with items from `::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.

remove NAME / Prima::MenuItem::remove

Deletes a menu item from the items tree, and its sub-menus if the item is a sub-menu item.

select

Alias for `selected(1)`. Sets menu object in selected state.

set_command KEY, ENABLED

Disables or enables menu items, associated with key combinations KEY.

set_variable NAME, NEW_NAME

Changes the name of a menu item with NAME to NEW_NAME. NEW_NAME must not be an empty string and must not be in a '#integer' form.

toggle NAME / Prima::MenuItem::toggle

Toggles the checked state of a menu item and returns the new state.

translate_accel TEXT

Locates a '~' (tilde) - escaped character in a 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 three-integer key representation into the one-integer format and returns the integer value. The three-integer format is used in `KeyDown` and `KeyUp` notifications for Prima::Widget.

See the *Prima::Widget* section

The method can be called with no object.

translate_shortcut KEY

Converts literal-represented 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 menu item in unchecked state.

3.13 Prima::Timer

Programmable periodical events

Synopsis

```
my $timer = Prima::Timer-> create(
    timeout => 1000, # milliseconds
    onTick  => sub {
       print "tick!\n";
    },
);

$timer-> start;
```

Description

Prima::Timer arranges periodical notifications to be delivered in certain time intervals. The notifications are triggered by the system, and are seen as `Tick` events. There can be many active Timer objects at one time, spawning events simultaneously.

Usage

Prima::Timer is a descendant of Prima::Component. Objects of Prima::Timer class are created in 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 inactive state; no events are spawned. To start spawning events, <start()> method must be explicitly called. Time interval value is assigned using the <::timeout> property in milliseconds.

When the system generates timer event, no callback is called immediately, - an event is pushed into stack instead, to be delivered during next event loop. Therefore, timeout value is not held accurately, and events may take longer time to pass. More accurate timing scheme, as well as timing with precision less than a millisecond, is not supported by the toolkit.

API

Properties

timeout MILLISECONDS

Manages time interval between `Tick` events. In set-mode call, if the timer is in active state (see `get_active()`, the new timeout value is applied immediately.

Methods

get_active

Returns a boolean flag, whether object is in active state or not. In the active state `Tick` events are spawned after `::timeout` time intervals.

get_handle

Returns a system-dependent handle of object

start

Sets object in active state. If succeed, or if the object is already in active state, returns 1. If the system was unable to create a system timer instance, 0 is returned.

stop

Sets object in inactive state.

Events

Tick

A system generated event, spawned every `::timeout` milliseconds if object is in active state.

3.14 Prima::Application

Root of widget objects hierarchy

Description

Prima::Application class serves as a hierarchy root for all objects with child-owner relationship. All toolkit objects, existing with non-null owner property, belong by their top-level parental relationship to Prima::Application object. There can be only one instance of Prima::Application class at a time.

Synopsis

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

or

```
use Prima qw(Application);

Prima::MainWindow-> create();

run Prima;
```

Usage

Prima::Application class, and its only instance are treated specially throughout the toolkit. The object instance is contained in

```
$::application
```

scalar, defined in *Prima.pm* module. The application instance must be created whenever widget and window, or event loop functionality is desired. Usually

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

code is enough, but *$::application* can also be assigned explicitly. The 'use' syntax has advantage as more resistant to eventual changes in the toolkit design. It can also be used in conjunction with custom parameters hash, alike the general create() 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 object classes. This functionality is generally explained in the *API* entry.

Inherited functionality

Prima::Application is a descendant of Prima::Widget, but it is designed so because their functional outliers are closest to each other. Prima::Application does not strictly conform (in OO sense) to any of the built-in classes. It has methods copied from both Prima::Widget and Prima::Window at one time, and the inherited Prima::Widget methods and properties function differently. For example, `::origin`, a property from Prima::Widget, is also implemented in Prima::Application, but returns always (0,0), an expected but not much usable result. `::size`, on the contrary, returns the extent of the screen in pixels. There are few properties, inherited from Prima::Widget, which return actual, but uninformative results, - `::origin` is one of those, but

same are `::buffered`, `::clipOwner`, `::enabled`, `::growMode`, `::owner` and owner-inheritance properties, `::selectable`, `::shape`, `::syncPaint`, `::tabOrder`, `::tabStop`, `::transparent`, `::visible`. To this group also belongs `::modalHorizon`, Prima::Window class property, but defined for consistency and returning always 1. Other methods and properties, like `::size`, that provide different functionality are described in the *API* entry.

Global functionality

Prima::Application is a wrapper to functionality, that is not related to one or another class clearly. A notable example, paint mode, which is derived from Prima::Drawable class, allows painting on the screen, overwriting the graphic information created by the other programs. Although being subject to begin_paint()/end_paint() brackets, this functionality can not be attached to a class-shared API, an therefore is considered global. All such functionality is gathered in the Prima::Application class.

These topics enumerated below, related to the global scope, but occupying more than one method or property - such functions described in the *API* entry.

Painting

As stated above, Prima::Application provides interface to the on-screen painting. This mode is triggered by begin_paint()/end_paint() methods pair, and 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.

Hint

$::application hosts a special Prima::HintWidget class object, accessible via `get_hint_widget()`, but with color and font functions aliased (`::hintColor`, `::hintBackColor`, `::hintFont`).

This widget serves as a hint label, floating over widgets if the mouse pointer hovers longer than `::hintPause` milliseconds.

Prima::Application internally manages all hint functionality. The hint widget itself, however, can be replaced before application object is created, using `::hintClass` create-only property.

Printer

Result of the *get_printer* entry method points to an automatically created printer object, responsible for the system-driven 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 document interface.

See the *Prima::Printer* section for details.

Clipboard

$::application hosts set of Prima::Clipboard objects, created automatically to reflect the system-provided clipboard IPC functionality. Their number depends on the system, - under X11 environment there is three clipboard objects, and only one under Win32 and OS/2.

These are no methods to access these clipboard objects, except fetch() (or, the indirect name calling) - the clipboard objects are named after the system clipboard names, which are returned by Prima::Clipboard::get_standard_clipboards.

The default clipboard is named *Clipboard*, and is accessible via

```
my $clipboard = $::application-> Clipboard;
```

code.

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 part of the toolkit, and resides in `Prima::HelpViewer` module, any custom help viewing module can be assigned. 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. Or, knowing the toolkit boundaries in some platforms, the program changes its behavior accordingly. Both these topics are facilitated by extra system information, returned by Prima::Application methods. `get_system_value` returns a system value for one of `sv::XXX` constants, so the program can read the system-specific information. As well as `get_system_info` method, that returns the short description of the system, it is the portable call. To the contrary, `sys_action` method is a wrapper to system-dependent functionality, called in non-portable way. This method is never used within the toolkit, and its usage is discouraged, primarily because its options do not serve the toolkit design, are subject to changes and cannot be relied upon.

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 general 'one main window' application layouts.

Default value: 0

icon OBJECT

Holds the icon object, associated with the application. If `undef`, a system-provided default icon is assumed. Prima::Window object instances inherit the application icon by default.

insertMode BOOLEAN

A system boolean flag, showing 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 a class of object, used as a help viewing package. The default value is Prima::HelpViewer.

Run-time changes to the property do not affect the help subsystem until `close_help` call is made.

helpModule STRING

Specifies a perl module, loaded indirectly when a help viewing call is made via `open_help`. Used when `::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 `close_help` call is made.

hintClass STRING

Create-only property.

Specifies a class of widget, used as the hint label.

Default value: Prima::HintWidget

hintColor COLOR

An alias to foreground color property for the hint label widget.

hintBackColor COLOR

An alias to background color property for the hint label widget.

hintFont %FONT

An alias to font property for the hint label widget.

hintPause TIMEOUT

Selects the timeout in milliseconds before the hint label is shown when the mouse pointer hovers over a widget.

modalHorizon BOOLEAN

A read-only property. Used as a landmark for the lowest-level modal horizon. Always returns 1.

palette [@PALETTE]

Used only within paint and information modes. Selects solid colors in a system palette, as many as possible. PALETTE is an array of integer triplets, where each is red, green, and blue component, with intensity range from 0 to 255.

printerClass STRING

Create-only property.

Specifies a class of object, used as a printer. The default value is system-dependent, but is either `Prima::Printer` or `Prima::PS::Printer`.

printerModule STRING

Create-only property.

Specifies a perl module, loaded indirectly before a printer object of `::printerClass` class is created. Used when `::printerClass` property is overridden and the new class is contained in a third-party module.

pointerVisible BOOLEAN

Governs 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, 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 functionality of `::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 widgets.

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 operating system do not support UTF8.

If 1, `Prima::Clipboard::text` property may return UTF8 text from system clipboards is available.

Default value: 0

Events

PasteText $CLIPBOARD, $$TEXT_REF

The notification queries `$CLIPBOARD` for text content and stores in `$$TEXT_REF`. Default action is that `'Text'` format is queried if `wantUnicodeInput` is unset. Otherwise, `'UTF8'` format is queried beforehand.

The `PasteText` mechanism is devised to ease defining text unicode/ascii conversion between clipboard and standard widgets, in a standard way.

Methods

add_startup_notification @CALLBACK

CALLBACK is an array of anonymous subs, which is executed when Prima::Application object is created. If the application object is already created during the call, CALLBACKs called immediately.

Useful for add-on packages initialization.

begin_paint

Enters the enabled (active paint) state, returns success flag. Once the object is in enabled state, painting and drawing methods can perform write operations on the whole screen.

begin_paint_info

Enters the information state, returns success flag. The object information state is same as 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 `close` for all top-level windows. The call can be interrupted by these, and thus canceled. If not canceled, stops the application event loop.

close_help

Closes the help viewer window.

end_paint

Quits the enabled state and returns application object to the normal state.

end_paint_info

Quits the information state and returns application object to the normal state.

font_encodings

Returns 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 hash of font hashes (see the **Fonts** entry in the *Prima::Drawable* section) describing fonts of NAME font family and of ENCODING. If NAME is '' or `undef`, returns one fonts 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 `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 element `encodings`, that points to array of the font encodings, available for the fonts of NAME font family.

get_active_window

Returns object reference to a 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 keyboard focus on one of its children widgets.

get_caption_font

Returns a 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 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_default_window_borders BORDER_STYLE = bs::Sizeable

Returns width and height of standard system window border decorations for one of `bs::XXX` constants. The method can be called with a class string instead of an object instance.

get_focused_widget

Returns object reference to a currently focused widget, if any, that belongs to the program. If no such widget exists, `undef` is returned.

get_hint_widget

Returns the hint label widget, attached automatically to Prima::Application object during startup. The widget is of `::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 screen boundaries, or WIDTH and HEIGHT are zero or negative, `undef` is returned.

get_indents

Returns 4 integers that corresponds 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 scraan that windows taskbar may occupies, if any.

get_printer

Returns the printer object, attached automatically to Prima::Application object. The object is of `::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 a 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 'lowest' one (in a simple case when window A is made modal (executed) after modal window B, the A window is the 'topmost' one).

If a chain is empty `undef` is returned. In case when a chain consists of just one window, TOPMOST value is apparently irrelevant.

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 is expired. The second 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 information about the system. The hash result contains the following keys:

apc

One of `apc::XXX` constants, reflecting the platform. Currently, the list of the supported platforms is:

```
apc::Os2
apc::Win32
apc::Unix
```

gui

One of `gui::XXX` constants, reflecting the graphic user interface used in the system:

```
gui::Default
gui::PM
gui::Windows
gui::XLib
gui::GTK2
```

guiDescription

Description of graphic user interface, returned as an arbitrary string.

system

An arbitrary string, representing the operating system software.

release

An arbitrary string, reflecting 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 `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 system accepts color pointer icons.
```

```
sv::CanUTF8_Input    - 1 if system can generate key codes in unicode
sv::CanUTF8_Output   - 1 if system can output utf8 text
```

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. It is usually passed to the program by other IPC means, so it returns 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 screen area under (X_OFFSET,Y_OFFSET) coordinates. If no toolkit widget are found, `undef` is returned.

go

The main event loop. Called by

run Prima;

standard code. Returns when the program is about to terminate, or if the exception was signaled. In the latter case, the loop can be safely re-started.

lock

Effectively blocks the graphic output for all widgets. The output can be restored with `unlock()`.

open_help TOPIC

Opens the help viewer window with TOPIC string in 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.

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 subject to change. The actual services can be looked in the toolkit source code under *apc_system_action* tag.

unlock

Unblocks the graphic output for all widgets, previously locked with `lock()`.

yield

An event dispatcher, called from within the event loop. If the event loop can be schematized, then in

```
while ( application not closed ) {
        yield
}
```

draft yield() is the only function, called repeatedly within the event loop. yield() cannot be used to organize event loops, but it can be employed to process stacked system events explicitly, to increase responsiveness of a program, for example, inside a long calculation cycle.

The method can be called with a class string instead of an object instance; however, the $::application object must be initialized.

3.15 Prima::Printer

System 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;
   $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

Prima::Printer is a descendant of *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*. *Prima::Printer* object is created only when the system provides graphic printing capabilities - drawing and painting procedures on a graphic device. If there are no such API, *Prima::Application* creates an instance of *Prima::PS::Printer* instead, which emulates a graphic device, producing PostScript output. The discretion between *Prima::Printer* and *Prima::PS::Printer* is transparent for both the user and the programmer, unless printer device specific adjustments desired.

A printing session is started by `begin_doc()`, which switches the object into the painting state. If finished by `end_doc()`, the document is delivered to a printer device. Alternative finishing method, `abort_doc()`, terminates the printing session with no information printed, unless the document is multi-paged and pages were sent to the printer via `new_page()`.

A printer object (that means, both *Prima::Printer* and *Prima::PS::Printer*) provides selection of the printer mechanism. `printers()` method returns array of hashes, each describing a printer device; `get_default_printer()` returns a default printer string identifier. A printer device can be selected via the `::printer` property.

The capabilities of the selected printer can be adjusted via `setup_dialog()` method, that invokes a system-provided (or, in case of *Prima::PS::Printer*, toolkit-provided) printer setup dialog, so the user can adjust settings of a printer device. It depends on the system, whether the setup changes only the instance settings, or the default behavior of a printer driver is affected for all programs.

Some printer capabilities can be queried by the `::size()` property, that reports the dimension of the page, the `::resolution()` property, that reports the DPI resolution selected by a printer driver and font list (by `fonts()` method), available for usage.

Typical code that prints the document looks like

```
my $p = $::application-> get_printer;
if ( $p-> begin_doc) {
   ... draw ...
   $p-> end_doc;
}
```

In addition, a standard package *Prima::PrintDialog* can be recommended so the user can select a printer device and adjust its setup interactively.

API

Properties

printer STRING

Selects a printer device, specified by its STRING identifier. Can not select a device if a printing session is started.

resolution X, Y

A read-only property; returns a DPI horizontal and vertical resolution, currently selected for a printer device. The user can change this, if the printer device supports several resolutions, inside `setup_dialog()`.

size WIDTH, HEIGHT

A read-only property; returns dimensions of a printer device page. The user can change this, if the printer device supports several resolutions or page formats, inside `setup_dialog()`.

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 been printed.

begin_doc DOCUMENT_NAME = ""

Initiates the printing session, and triggers the object into the enabled painting state. The document is assigned DOCUMENT_NAME string identifier.

begin_paint

Identical to `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 `new_page()` and `abort_doc()` methods work. The information mode is exited via `end_paint_info()` method.

end_doc

Quits the printing session and delivers the document to a printer device. Does not report eventual errors, occurred during the spooling process - the system is expected to take care about 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 into the disabled painting state.

font_encodings

Returns array of 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 hash of font hashes (see the *Prima::Drawable* section, Fonts section) describing fonts of NAME font family and of ENCODING. If NAME is '' or `undef`, returns one fonts 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 `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 element `encodings`, that points to array of the font encodings, available for the fonts of NAME font family.

new_page

Finalizes the current page and starts a new blank page.

options [OPTION, [VALUE, [...]]]

Queries and sets printer-specific setup options, such as orientation, paper size, etc. If called without parameters, returns list of options the printer supports. If called with one parameter, treats is as the option name and return the corresponsing value. Otherwise, treats parameters as a list of key-value pairs, and sets the printer options. Returns number of options that were successfully set.

The compatibility between options and values used by different OSes is low here. 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-dependant. For win32, consult Microsoft manual on DEVMODE structure the *http:msdn.microsoft.comlibraryen-usgdiprntspol_8nle.asp* entry; for Prima's own PostScript printer, consult the *Prima::PS::Printer* section.

printers

Returns array of hashes, where each entry describes a printer device. The hash consists of the following entries:

name

A printer device name

device

A physical device name, that the printer is connected to

defaultPrinter

A boolean flag, 1 if the printer is default, 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 a string, identifying a default printer device.

get_handle

Returns a system handle for a printer object.

3.16 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 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

Prima::File provides access to the I/O stream events, that are called when a file handle becomes readable, writable or if an exception occurred. Registering file handles to Prima::File objects makes possible the stream operations coexist with the event loop.

Usage

Prima::File is a descendant of Prima::Component. Objects of Prima::File class must be binded to a valid file handle object, before the associated events can occur:

```
my $f = Prima::File-> create();
$f-> file( *STDIN);
```

When a file handle, binded via the `::file` property becomes readable, writable or when an exception signaled, one of three correspondent events called - `Read`, `Write` or `Exception`. When a handle is always readable, or always writable, or, 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);
```

NB. Due to different system implementations, the only handles, currently supported on all systems, are socket handle and disk file handles. Pipes only work on unix platforms. The example file *socket.pl* elucidates the use of sockets together with Prima::File.

When a file handle is not needed anymore, it is expected to be detached from an object explicitly:

```
$f-> file( undef);
```

However, if the system detects that a 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 are basically the same I/O callbacks, provided by a system `select()` call. See documentation of your system's select() for the implementation details.

API

Properties

file HANDLE

Selects a file handle, that is to be monitored for stream I/O events. If HANDLE is `undef`, object is returned to a passive state, and the previously binded file handle is de-selected.

mask EVENT_MASK

Selects a event mask, that is a combination of `fe::XXX` integer constants, each representing an event:

```
fe::Read
fe::Write
fe::Exception
```

The omitted 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 fileno() result.

is_active AUTODETACH = 0

Returns a boolean flag, indicating if a file handle is valid. If AUTODETACH is 1, and the file handle is not valid, `file(undef)` is called.

Events

Read

Called when a file handle becomes readable. The callback procedure is expected to call a non-blocking read() on the file handle.

Write

Called when a 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 a file handle. The exceptions are specific to handle type and the operating system. For example, a unix socket signals `Exception` when a control status data for a pseudo terminal or an out-of-band data arrives.

4 Widget library

4.1 Prima::Buttons

Button widgets and grouping widgets.

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 two separate sets of classes: the button widgets and the grouping widgets. The button widgets include push buttons, check-boxes and radio buttons. The grouping widgets are designed for usage as containers for the check-boxes and radio buttons, however, any widget can be inserted in a grouping widget.

The module provides the following classes:

```
*Prima::AbstractButton ( derived from Prima::Widget and Prima::MouseScroller )
        Prima::Button
                Prima::SpeedButton
        *Prima::Cluster
                Prima::CheckBox
                Prima::Radio
Prima::GroupBox ( derived from Prima::Widget )
        Prima::RadioGroup       ( obsolete )
        Prima::CheckBoxGroup    ( obsolete )
```

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 common functionality of buttons. It provides reaction on mouse and keyboard events, and calls the *Click* entry notification when the user activates the button. The mouse activation is performed either by mouse double click or successive mouse down and mouse up events within the button boundaries. The keyboard activation is performed on the following conditions:

- The spacebar key is pressed
- {`default`} (see the *default* entry property) boolean variable is set and enter key is pressed. This condition holds even if the button is out of focus.
- {`accel`} character variable is assigned and the corresponding character key is pressed. {`accel`} variable is extracted automatically from the text string passed to the *text* entry property. This condition holds even if the button is out of focus.

Events

Check

Abstract callback event.

Click

Called whenever the user presses the button.

Properties

pressed BOOLEAN

Represents the state of button widget, whether it is pressed or not.

Default value: 0

text STRING

The text that is drawn in the button. If STRING contains ~ (tilde) character, the following character is treated as a hot key, and the character is underlined. If the user presses the corresponding character key then the *Click* entry event is called. This is true even when the button is out of focus.

Methods

draw_veil CANVAS, X1, Y1, X2, Y2

Draws a rectangular veil shape over CANVAS in given boundaries. This is the default method of drawing the button in the disabled state.

draw_caption CANVAS, X, Y

Draws single line of text, stored in the *text* entry property on CANVAS at X, Y coordinates. Performs underlining of eventual tilde-escaped character, and draws the text with dimmed colors if the button is disabled. If the button is focused, draws a dotted line around the text.

caption_box [CANVAS = self]

Calculates geometrical extensions of text string, stored in the *text* entry property in pixels. Returns two integers, the width and the height of the string for the font selected on CANVAS. If CANVAS is undefined, the widget itself is used as a graphic device.

Prima::Button

A push button class, that extends Prima::AbstractButton functionality by allowing an image to be drawn together with the text.

Properties

autoHeight BOOLEAN

If 1, the button height is automatically changed as text extensions change.

Default value: 1

autoRepeat BOOLEAN

If set, the button behaves like a keyboard button - after the first the *Click* entry event, a timeout is set, after which is expired and the button still pressed, the *Click* entry event is repeatedly called until the button is released. Useful for emulating the marginal scroll-bar buttons.

Default value: 0

autoWidth BOOLEAN

If 1, the button width is automatically changed as text extensions change.

Default value: 1

borderWidth INTEGER

Width of 3d-shade border around the button.

Default value: 2

checkable BOOLEAN

Selects if the button toggles 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 *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 the black border, indicating that it executes the 'default' action. Useful for OK-buttons in dialogs.

Default value: 0

defaultGlyph INTEGER

Selects index of the default sub-image.

Default value: 0

disabledGlyph INTEGER

Selects index of the sub-image for the disabled button state. If `image` does not contain such sub-image, the `defaultGlyph` sub-image is drawn, and is dimmed over with 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 the 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 button resource to contain all button states into one image file. The `glyphs` property assigns how many such sub-images the image object contains.

The sub-image indices can be assigned for rendition of the different states. These indices are selected by the following integer properties: the *defaultGlyph* entry, the *hiliteGlyph* entry, the *disabledGlyph* entry, the *pressedGlyph* entry, the *holdGlyph* entry.

Default value: 1

hiliteGlyph INTEGER

Selects index of the sub-image for the state when the mouse pointer is over the button. This image is used only when the *flat* entry property is set. If `image` does not contain such sub-image, the `defaultGlyph` sub-image is drawn.

Default value: 0

holdGlyph INTEGER

Selects index of the sub-image for the state when the button is the *checked* entry. This image is used only when the *checkable* entry property is set. If `image` does not contain such 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, over or left to it (see the *vertical* entry property). If OBJECT contains several sub-images, then the corresponding sub-image is drawn for each button state. See the *glyphs* entry property.

Default value: undef

imageFile FILENAME

Alternative to image selection by loading an image from the file. During the creation state, if set together with the *image* entry property, is superseded by the latter.

To allow easy multiframe image access, FILENAME string is checked if it contains a number after a colon in the string end. Such, `imageFile('image.gif:3')` call would load the fourth frame in `image.gif` file.

imageScale SCALE

Contains 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 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 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

pressedGlyph INTEGER

Selects index of the sub-image for the pressed state of the button. If `image` does not contain such 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 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, image is centered.

Prima::SpeedButton

A convenience class, same as the *Prima::Button* section but with default square 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 arrow key walking by the radio buttons without pressing spacebar key. It is also has a drawback, that if a radio button gets focused without user intervention, or indirectly, it also gets checked, so that behavior might cause confusion. The said can be exemplified when an unchecked radio button in a notebook widget gets active by turning the notebook page.

Although this property is present on the the *Prima::CheckBox* section, 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 a standard radio button, that can be either in checked, or in unchecked state. When checked, delivers the *RadioClick* entry event to the owner (if the latter provides one).

The button uses the standard toolkit images with `sbmp::RadioXXX` indices. If the images can not be loaded, the button is drawn with the graphic primitives.

Events

Check

Called when a button is checked.

Prima::CheckBox

Represents a standard check box button, that can be either in checked, or in unchecked state.

The button uses the standard toolkit images with `sbmp::CheckBoxXXX` indices. If the images can not be loaded, the button is drawn with graphic primitives.

Prima::GroupBox

The class to be used as a container of radio and check-box buttons. It can, however, contain any other widgets.

The widget draws a 3d-shaded box on its boundaries and a text string in its upper left corner. Uses `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 the half of the font height.

Events

RadioClick BUTTON

Called whenever one of 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 `RadioClick` event is `nt::PrivateFirst`, `on_radioclick` method must be directly overloaded to disable this functionality.

Properties

index INTEGER

Checks the child radio button with `index`. The indexing is based on the index in the widget list, returned by `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 `Prima::Widget::widgets` method.

Prima::RadioGroup

This class is obsolete and is same as `Prima::GroupBox`.

Prima::CheckBoxGroup

This class is obsolete and is same as `Prima::GroupBox`.

Bugs

The push button is not capable of drawing anything other than single line of text and single image. If an extended functionality is needed, instead of fully rewriting the painting procedure, it might be reasonable to overload `put_image_indirect` method of `Prima::Button`, and perform custom output there.

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.

`Prima::GroupBox::value` parameter is an integer, which size is architecture-dependent. Shift towards a vector is considered a good idea.

4.2 Prima::Calendar

Standard calendar widget

Synopsis

```
use Prima::Calendar;
my $cal = Prima::Calendar-> create(
        useLocale => 1,
        onChange  => sub {
                print $_[0]-> date_as_string, "\n";
        },
);
$cal-> date_from_time( localtime );
$cal-> month( 5);
```

Description

Provides interactive selection of date between 1900 and 2099 years. The main property, the *date* entry, is a three-integer array, day, month, and year, in the format of perl localtime (see localtime in *perlfunc*) - day can be in range from 1 to 31,month from 0 to 11, year 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 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 month.

firstDayOfWeek INTEGER

Selects the first day of week, an integer between 0 and 6, where 0 is Sunday is 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 invocation of 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 boolean value, whether the locale information can be retrieved by calling `strftime`.

month2str MONTH

Returns MONTH name according to the *useLocale* entry value.

make_months

Returns array of 12 month names according to the *useLocale* entry value.

day_of_week DAY, MONTH, YEAR, [USE_FIRST_DAY_OF_WEEK = 1]

Returns integer value, from 0 to 6, of the day of week on DAY, MONTH, YEAR date. If boolean USE_FIRST_DAY_OF_WEEK is set, the value of `firstDayOfWeek` property is taken into the account, so 0 is a Sunday shifted forward by `firstDayOfWeek` days.

The switch from Julian to Gregorian calendar is ignored.

date_as_string [DAY, MONTH, YEAR]

Returns string representation of date on DAY, MONTH, YEAR according to the *useLocale* entry property value.

date_from_time SEC, MIN, HOUR, M_DAY, MONTH, YEAR, ...

Copies the *date* entry from `localtime` or `gmtime` result. This helper method allows the following syntax:

```
$calendar-> date_from_time( localtime( time));
```

4.3 Prima::ComboBox

Standard combo box widget

Synopsis

```
use Prima::ComboBox;

my $combo = Prima::ComboBox-> create( style => cs::DropDown, items => [ 1 .. 10 ]);
$combo-> style( cs::DropDownList );
print $combo-> text;
```

Description

Provides a combo box widget which consists of an input line, list box of possible selections and eventual drop-down button. The combo box can be either in form with a drop-down selection list, that is shown by the command of the user, or in form when the selection list is always visible.

The combo box is a grouping widget, and contains neither painting nor user-input code. All such functionality is delegated into the children widgets: input line, list box and button. `Prima::ComboBox` exports a fixed list of methods and properties from namespaces of the *Prima::InputLine* section and the *Prima::ListBox* section. Since, however, 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 would be other classes, it is not necessarily that all default functionality would work. The list of exported names is stored in package variables %listProps, %editProps and %listDynas. These also described in the *Exported names* entry section.

The module defines `cs::` package for the constants used by the *style* entry property.

API

Properties

buttonClass STRING

Assigns a drop-down button class.

Create-only property.

Default value: `Prima::Widget`

buttonDelegations ARRAY

Assigns a drop-down button list of delegated notifications.

Create-only property.

buttonProfile HASH

Assigns 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 an input line class.

Create-only property.

Default value: `Prima::InputLine`

editProfile HASH

Assigns hash of properties, passed to the input line during the creation.

Create-only property.

editDelegations ARRAY

Assigns an input line list of delegated notifications.

Create-only property.

editHeight INTEGER

Selects height of an input line.

items ARRAY

Mapped onto the list widget's `items` property. See the *Prima::Lists* section for details.

listClass STRING

Assigns a listbox class.

Create-only property.

Default value: `Prima::ListBox`

listHeight INTEGER

Selects height of the listbox widget.

Default value: 100

listVisible BOOLEAN

Sets whether the listbox is visible or not. Not writable when the *style* entry is `cs::Simple`.

listProfile HASH

Assigns hash of properties, passed to the listbox during the creation.

Create-only property.

listDelegations ARRAY

Assigns a selection listbox list of delegated notifications.

Create-only property.

literal BOOLEAN

Selects whether the combo box user input routine assume that the listbox contains literal strings, that can be fetched via `get_item_text` (see the *Prima::Lists* section). As an example when this property is set to 0 is `Prima::ColorComboBox` from the *Prima::ComboBox* section package.

Default value: 1

style INTEGER

Selected one of three styles:

cs::Simple

The listbox is always visible, and the drop-down button is not.

cs::DropDown

The listbox is not visible, but the drop-down button is. When the use presses the drop-down button, the listbox is shown; when the list-box is defocused, it gets hidden.

cs::DropDownList

Same as `cs::DropDown`, but the user is restricted in the selection: the input line can only accept user input that is contained in listbox. If the *literal* entry set to 1, the auto completion feature is provided.

text STRING

Mapped onto the edit widget's `text` property.

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

```
autoHeight     focusedItem    hScroll
integralHeight items          itemHeight
topItem        vScroll        gridColor
multiColumn    offset
```

%listDynas

```
onDrawItem
onSelectItem
```

4.4 Prima::DetailedList

A multi-column list viewer with controlling header widget.

Synopsis

use Prima::DetailedList;

```
my $l = $w-> insert( 'Prima::DetailedList',
        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);
```

Description

Prima::DetailedList is a descendant of Prima::ListViewer, and as such provides a certain level of abstraction. It overloads format of the *items* entry in order to support multi-column (2D) cell span. It also inserts the *Prima::Header* section widget on top of the list, so the user can interactively move, resize and sort the content of the list. The sorting mechanism is realized inside the package; it is activated by the mouse click on a header tab.

Since the class inherits Prima::ListViewer, some functionality, like 'item search by key', or `get_item_text` method can not operate on 2D lists. Therefore, the *mainColumn* entry property is introduced, that selects the column representing all the data.

API

Events

Sort COLUMN, DIRECTION

Called inside the *sort* entry method, to facilitate custom algorithms of sorting. If the callback procedure is willing to sort by COLUMN index, then it must call `clear_event`, to signal the event flow stop. The DIRECTION is a boolean flag, specifying whether the sorting must be performed is 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 precedence is arbitrated by `cmp` operator (see Equality Operators in *perlop*) .

Properties

columns INTEGER

Governs 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 a header class.

Create-only property.

Default value: `Prima::Header`

headerProfile HASH

Assigns hash of properties, passed to the header widget during the creation.

Create-only property.

headerDelegations ARRAY

Assigns a header widget list of delegated notifications.

Create-only property.

headers ARRAY

Array of strings, passed to the header widget as column titles.

items ARRAY

Array of arrays of scalars, of arbitrary 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 representation of all the data. As the user clicks the 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 COLUMN was specified, and the last specified column equals to COLUMN, the sort direction is reversed.

The method does not perform sorting itself, but invokes the *Sort* entry notification, so the sorting algorithms can be overloaded, or be applied differently to the columns.

4.5 Prima::DetailedOutline

A multi-column outline viewer with controlling header widget.

Synopsis

```
use Prima::DetailedOutline;

my $l = $w-> insert( 'Prima::DetailedList',
      columns => 2,
      headers => [ 'Column 1', 'Column 2' ],
      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);
```

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 is introduced, and one property is different enough to warrant 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 child items.

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 within 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.

`Prima::DockManager` class is an application-oriented class in a way that (mostly) the only instance of it is needed in the program. It is derived from `Prima::Component` and therefore is never visualized. The class instance is stored in `instance` property of all module classes to serve as a docking hierarchy root. Through the document, *instance* term is referred to `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 *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 dock and accept docking widgets simultaneously. In the scope of `Prima::DockManager`, the toolbar hosts command widget, 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 a toolbar when all of its command widgets are undocked. If 1, the toolbar is automatically destroyed. If 0 it calls `visible(0)`.

childDocker WIDGET

Pointer to `Prima::DockManager::ToolbarDocker` instance.

See also `Prima::DockManager::ToolbarDocker::parentDocker`.

instance INSTANCE

`Prima::DockManager` instance, the docking hierarchy root.

Prima::DockManager::ToolbarDocker

Internal class, implements a dock widget for command widgets, while serves as a client in a dockable toolbar, which is a `Prima::LinearDockerShuttle` descendant. When its size is changed due an eventual rearrange of its docked widgets, also resizes the toolbar.

Properties

instance INSTANCE

`Prima::DockManager` instance, the docking hierarchy root.

parentDocker WIDGET

Pointer to `Prima::DockManager::Toolbar` instance. When in the docked state, `parentDocker` value is always equals 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 size is changed to resizes the owner widget. If it is in the docked state, the size change might result in change of position or docking state.

Prima::DockManager::Panelbar

The class is derived from `Prima::LinearDockerShuttle`, and is different only in that `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 close button on the external shuttle.

Properties

instance INSTANCE

`Prima::DockManager` instance, the docking hierarchy root.

Prima::DockManager

A binder class, contains set of functions that groups toolbars, panels, and command widgets together under the docking hierarchy.

The manager servers several purposes. First, it is a command state holder: the command widgets, mostly buttons, usually are in enabled or disabled state in different life stages of a program. The manager maintains the enabled/disabled state by assigning each command an unique scalar value (farther and in the code referred as *CLSID*). The toolbars can be created with set of command widgets, referred via 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 CLSID, the commands can be grouped by sections. Both CLSID and group descriptor scalars are defined by the programmer.

Second, `create_manager` method presents a standard configuration widget, that allows 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 maintaining the toolbars and panels visibility when the main window is hidden or restored. `windowState` method hides or shows the toolbars and panels effectively.

Fourth, it serves as a docking hierarchy root. All docking sessions begin from `Prima::DockManager` object, which although does not provide docking capabilities itself (it is `Prima::Component` descendant), redirects the docking requests to the lower-level dock widgets.

Fifth, it provides number of helper methods and notifications, and enforces use or `fingerprint` property by all dockable widgets. This property has default value of `0xFFFF` (defined in `Prima::Docks`). The module contains the fingerprint `dmfp::XXX` constants with value greater than this, so the toolbars and panels are not docked to a dock widget with the default configuration. The base constant set is:

```
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. If value is 1, the command is available. If 0, the command is disabled. Changes to this property are reflected in the visible command widgets, which are enabled or disabled immediately. Also, `CommandChange` notification is triggered.

fingerprint INTEGER

The property is read-only, and always returns `0xFFFFFFFF`, to allow landing for all dockable widgets. In case when 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 `create_manager` 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 an unique name for an automatically created toolbar, like `Toolbar1`, `Toolbar2` etc.

commands_enable BOOLEAN, @CLSIDs

Enabled or disables commands from CLSIDs array. The changes are reflected in the visible command widgets, which are enabled or disabled immediately. Also, `CommandChange` notification is triggered.

create_manager OWNER, %PROFILE

Inserts two widgets into OWNER with PROFILE parameters: a listbox 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 are interactively selected by the listbox navigation.

The icons, dragged from the notebook, behave as dockable widgets: they can be landed upon 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 two widgets, the listbox and the notebook.

PROFILE recognizes the following keys:

origin X, Y

Position where the widgets are to be inserted. Default value is 0,0.

size X, Y

Size of the widget insertion area. By default the widgets occupy all OWNER interior.

listboxProfile PROFILE

Custom parameters, passed to the listbox.

dockerProfile PROFILE

Custom parameteres, passed to the notebook.

create_panel CLSID, %PROFILE

Creates a dockable panel of a previously registered CLSID by `register_panel`. PROFILE recognizes the following keys:

profile HASH

Hash of parameters, passed to `create()` of the panel widget class. Before passing it is merged with the set of parameters, registered by `register_panel`. The `profile` hash takes the precedence.

dockerProfile HASH

Constains extra options, passed to `Prima::DockManager::Panelbar` widget. Before the usage it is merged with the set of parameters, registered by `register_panel`.

NB: The `dock` key here contains a reference to a desired dock widget. If `dock` 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 CLSID by `register_tool`, into OWNER widget with X1 - Y2 coordinates. For automatic maintenance of enable/disable state of command widgets OWNER is expected to be a toolbar. If it is not, the maintenance must be performed separately, for example, by `CommandChange` event.

create_toolbar %PROFILE

Creates a new toolbar of `Prima::DockManager::Toolbar` class. The following PROFILE options are recognized:

autoClose BOOLEAN

Sets `autoClose` property of the toolbar.

Default value is 1 if `name` options is set, 0 otherwise.

dock DOCK

Contain a reference to a 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.

NB: The `dock` key here contains a reference to a desired dock widget. If `dock` set to `undef`, the panel is created in the non-docked state.

rect X1, Y1, X2, Y2

Selects rectangle 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 creation properties.

vertical BOOLEAN

Sets `vertical` property of the toolbar.

visible BOOLEAN

Selects visibility state of the toolbar.

get_class CLSID

Returns class record hash, registered under 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 `create` when the widget is created.

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

Return reference to a panel widget represented by CLSID scalar, or `undef` if none found.

panel_menuitems CALLBACK

A helper function; maps all panel names into a structure, ready to feed into `Prima::AbstractMenu::items` property (see the *Prima::Menu* section). The action member of the menu item record is set to CALLBACK scalar.

panel_visible CLSID, BOOLEAN

Sets the visibility of a panel, referred by CLSID scalar. If VISIBLE is 0, a panel is destroyed; if 1, 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 second is the default dock widget. Checks for panel visibility, and creates the panels that are not visible.

The method is useful in program startup, when some panels have to be visible from the beginning.

predefined_toolbars @PROFILES

Accepts array of hashes, where each array item describes a toolbar and a default set of command widgets. Checks for 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

Array of CLSIDs corresponding to the command widgets to be inserted into the toolbar.

name STRING

Selects 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 CLSID scalar. PROFILE must contain the following keys:

class STRING

Client widget class

text STRING

String, displayed in the panel title bar

dockerProfile HASH

Hash of parameters, passed to `Prima::DockManager::Panelbar`.

profile

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 CLSID scalar. PROFILE must be set the following keys:

class STRING

Client widget class

profile HASH

Hash of parameters, passed to the control widget.

toolbar_by_name NAME

Returns a pointer to a toolbar of NAME, or `undef` if none found.

toolbar_menuitems CALLBACK

A helper function; maps all toolbar names into a structure, ready to feed into `Prima::AbstractMenu::items` property (see the *Prima::Menu* section). The action member of the menu item record is set to 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 `commands` property changes or `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, hidden, or destroyed by the user.

Prima::DockManager::S::SpeedButton

The package simplifies creation of `Prima::SpeedButton` command widgets.

Methods

class IMAGE, CLSID, %PROFILE

Builds a hash with parameters, ready to feed `Prima::DockManager::register_tool` for registering a `Prima::SpeedButton` class instance with PROFILE parameters.

IMAGE is a path to a image file, loaded and stored in the registration hash. IMAGE 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 `:` character prefix.

CLSID scalar is not used; it is returned so the method result can directly be passed into `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 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 `Prima::AbstractDocker::Interface` package.

Usage

A dockable widget is required to take particular steps before it can dock to a dock widget. It needs to talk to the dock and find out if it is allowed to land, or if the dock contains lower-level dock widgets that might suit better for 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 mouse and the arbitration is performed on geometrical distance basis.

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 `dockingRoot` property for this purpose.

The conversation between parties starts when a dockable widget calls `open_session` method of the dock. The dockable widget passes set of parameters signaling if the widget is ready to change its size in case the dock widget requires so, and how. `open_session` method can either refuse or accept the widget. In case of the positive answer from `open_session`, the dockable widget calls `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 `next_docker` method. The session is closed by `close_session` call; after that, the widget is allowed to dock by setting its `owner` to the dock widget, the `rect` property to the negotiated position and size, and calling `dock` method.

`open_session`/`close_session` brackets are used to cache all necessary calculations once, making `query` call as light as possible. This design allows a dockable widget, when dragged, 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 `find_docking` method which performs the arbitration automatically and returns the appropriate dock widget.

`Prima::InternalDockerShuttle` is a class that implements dockable widget functionality. It also employs a top-level window-like wrapper widget for the dockable widget when it is not docked. By default, `Prima::ExternalDockerShuttle` is used as the wrapper widget class.

It is not required, however, to use neither `Prima::InternalDockerShuttle` nor `Prima::AbstractDocker::Interface` to implement a dockable widget; the only requirements to one is to respect `open_session/close_session` protocol.

`Prima::InternalDockerShuttle` initiates a class hierarchy of dockable widgets. Its descendants are `Prima::LinearWidgetDocker` and, in turn, `Prima::SingleLinearWidgetDocker`. `Prima::SimpleWidgetDocker` and `Prima::LinearWidgetDocker`, derived from `Prima::AbstractDocker::Interface`, begin hierarchy of dock widgets. The full hierarchy is as follows:

```
Prima::AbstractDocker::Interface
        Prima::SimpleWidgetDocker
        Prima::ClientWidgetDocker
        Prima::LinearWidgetDocker
        Prima::FourPartDocker

Prima::InternalDockerShuttle
        Prima::LinearDockerShuttle
        Prima::SingleLinearWidgetDocker

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. The most complicated scheme is implemented in `Prima::LinearWidgetDocker`; it does not allow children overlapping and is able to rearrange with 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 `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 dock widget classes.

Properties

Since the class is not `Prima::Object` descendant, it provides only run-time implementation of its properties. It is up to the descendant object whether the properties are recognized on the creation stage or not.

fingerprint INTEGER

A custom bit mask, to be used by docking widgets to reject inappropriate dock widgets on early stage. The `fingerprint` property is not part of the protocol, and is not required to be present in a dockable widget implementation.

Default value: `0x0000FFFF`

dockup DOCK_WIDGET

Selects the upper link in 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 lower-level docker widgets. The items of the list are returned by `next_docker` method.

check_session SESSION

Debugging procedure; checks SESSION hash, 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 is successfully finished negotiation with the dock widget and changed its `owner` property. The method adapts the dock widget layout and lists WIDGET into list of docked widgets. The method does not change `owner` property of 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 array of docked widgets.

next_docker SESSION, [X, Y]

Enumerates lower-level docker widgets within SESSION; returns one docker widget at a time. After the last widget returns `undef`.

The enumeration pointer is reset by `query` call.

X and Y are coordinates of the point of interest.

open_session PROFILE

Opens docking session with parameters stored in PROFILE and returns 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 screen coordinate system.

self WIDGET

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 used as last resort if `sizes` key does not contain the desired size.

sizeMin ARRAY

Two integers; minimal size that the widget can accept.

sizes ARRAY

Contains arrays of points in (X,Y) format; each point represents an acceptable size of the widget. If `sizeable` flags are set to 0, and none of `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 into 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 `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 change its owner to 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 lower-lever dock widgets, returns the first lower-lever widget. The caller can use subsequent calls to `next_docker` to enumerate all lower-level dock widgets. A call to `query` resets the internal enumeration pointer.

If the widget cannot 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 `redock` method in its namespace, it is called instead.

remove_subdocker SUBDOCK

Removes SUBDOCK from the list of lower-level docker widgets. See also the *add_subdocker* entry.

replace FROM, TO

Assigns widget TO same owner and rectangle as FROM. The FROM widget must be a docked widget.

undock WIDGET

Removes WIDGET from list of docked widgets. The layout of the dock widget can be changed after execution of this method. The method does not change `owner` property of WIDGET.

The method must not be called directly.

Prima::SimpleWidgetDocker

A simple dock widget; accepts any widget that geometrically fits into. Allows overlapping of the docked widgets.

Prima::ClientWidgetDocker

A simple dock widget; accepts any widget that can be fit to cover all dock's interior.

Prima::LinearWidgetDocker

A toolbar-like docking widget class. The implementation does not allow tiling, and can reshape the dock widget and rearrange the docked widgets if necessary.

`Prima::LinearWidgetDocker` is orientation-dependent; its main axis, governed by `vertical` property, is used to align 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 `grow::XXX` constants, that describes how the dock widget can be resized. The constants are divided in 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 be grown to 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
```

`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 wigdet 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 `dock` is successfully finished.

DockError WIDGET

Called when `dock` 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 line (or row, depending on `vertical` property value) of docked widgets.

Prima::FourPartDocker

Implementation of a docking widget, with its four sides acting as 'rubber' docking areas.

Properties

indents ARRAY

Contains four integers, specifying the breadth of offset for each side. The first integer is width of the left side, the second - height of the bottom side, the third - width of the right side, the fourth - height of the top side.

dockerClassLeft STRING

Assigns class of left-side dock window.

Default value: `Prima::LinearWidgetDocker`. Create-only property.

dockerClassRight STRING

Assigns class of right-side dock window.

Default value: `Prima::LinearWidgetDocker`. Create-only property.

dockerClassTop STRING

Assigns class of top-side dock window.

Default value: `Prima::LinearWidgetDocker`. Create-only property.

dockerClassBottom STRING

Assigns class of bottom-side dock window.

Default value: `Prima::LinearWidgetDocker`. Create-only property.

dockerClassClient STRING

Assigns class of center dock window.

Default value: `Prima::ClientWidgetDocker`. Create-only property.

dockerProfileLeft HASH

Assigns hash of properties, passed to the left-side dock widget during the creation.

Create-only property.

dockerProfileRight HASH

Assigns hash of properties, passed to the right-side dock widget during the creation.

Create-only property.

dockerProfileTop HASH

Assigns hash of properties, passed to the top-side dock widget during the creation.

Create-only property.

dockerProfileBottom HASH

Assigns hash of properties, passed to the bottom-side dock widget during the creation.

Create-only property.

dockerProfileClient HASH

Assigns hash of properties, passed to the center dock widget during the creation.

Create-only property.

dockerDelegationsLeft ARRAY

Assigns the left-side dock list of delegated notifications.

Create-only property.

dockerDelegationsRight ARRAY

Assigns the right-side dock list of delegated notifications.

Create-only property.

dockerDelegationsTop ARRAY

Assigns the top-side dock list of delegated notifications.

Create-only property.

dockerDelegationsBottom ARRAY

Assigns the bottom-side dock list of delegated notifications.

Create-only property.

dockerDelegationsClient ARRAY

Assigns the center dock list of delegated notifications.

Create-only property.

dockerCommonProfile HASH

Assigns hash of properties, passed to all 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 an `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 non-docked state. The latter is implemented by reparenting of the client widget to an external shuttle widget, selected by `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`, while itself it is derived from `Prima::Widget` only.

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 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 class of external shuttle widget.

Default value: `Prima::ExternalDockerShuttle`

externalDockerModule STRING

Assigns module that contains the external shuttle widget class.

Default value: `Prima::MDI` (`Prima::ExternalDockerShuttle` is derived from `Prima::MDI`).

externalDockerProfile HASH

Assigns hash of properties, passed to the external shuttle widget during the creation.

fingerprint INTEGER

A custom bit mask, used to reject inappropriate dock widgets on 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 a rectangle that the shuttle would occupy if its client rectangle is assigned to X1, Y1, X2, Y2 rectangle.

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 STATE boolean flag.

If it is 1, RECT is an array with the coordinates of the shuttle rectangle before the drag has started; ANCHOR_X and ANCHOR_Y are coordinates of the aperture point where the mouse event occurred that has initiated the drag. 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 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 particular 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, `GetCaps` request notification is triggered, which fills the parameters. The default action sets `sizeable` options according to `x_sizeable` and `y_sizeable` properties.

In case an appropriate landing area is found, `Landing` notification is triggered with the proposed dock widget and the target rectangle. The area can be rejected on this stage if `Landing` returns negative answer.

On success, returns a dock widget found and the target rectangle; the widget is never docked though. On failure returns an empty array.

This method is used by the dragging routine to provide a 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 a rectangle that the client would occupy if the shuttle rectangle is assigned to X1, Y1, X2, Y2 rectangle.

redock

If docked, undocks form the dock widget and docks back. If not docked, does not perform anything.

Events

Dock

Called when shuttle is docked.

EDSClose

Triggered when the user presses close button or otherwise activates the `close` function of the EDS (external docker shuttle). To cancel the closing, `clear_event` must be called inside the event handler.

FailDock X, Y

Called after the dragging session in the non-docked stage is 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 DOCK widget. PROFILE is a hash reference, which is to be filled inside the event handler. After that PROFILE is passed to `open_session` call.

The default action sets `sizeable` options according to `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 shuttle is switched to the non-docked state.

Prima::ExternalDockerShuttle

A shuttle class, used to host a client of `Prima::InternalDockerShuttle` widget when it is in the non-docked state. The class represents an emulation of a top-level window, which can be moved, resized (this feature is not on by default though), and closed.

`Prima::ExternalDockerShuttle` is inherited from `Prima::MDI` class, and its window emulating functionality is a subset of its ascendant. See also the *Prima::MDI* section.

Properties

shuttle WIDGET

Contains 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. All its properties concern only the way the shuttle draws itself.

Properties

headerBreadth INTEGER

Breadth of the header in pixels.

Default value: 8

indent INTEGER

Provides a wrapper to `indents` property; besides the space for the header, all indents are assigned to `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 editing widget

Synopsis

```
use Prima::Edit;
my $e = Prima::Edit-> create(
        text         => 'Hello $world',
        syntaxHilite => 1,
);
$e-> selection( 1, 1, 1, 2);
```

Description

The class provides text editing capabilities, three types of selection, text wrapping, syntax highlighting, auto indenting, undo and redo function, search and replace methods.

The module declares `bt::` package, that contains integer constants for selection block type, used by the *blockType* entry property.

Usage

The class addresses the text space by (X,Y)-coordinates, where X is character offset and Y is line number. The addressing can be 'physical' and 'logical', - in logical case Y is number of line of text. The difference can be observed if the *wordWrap* entry property is set to 1, when a single text string can be shown as several sub-strings, called *chunks*.

The text is stored line-wise in {`lines`} array; to access it use the *get_line* entry method. To access the text chunk-wise, use the *get_chunk* entry method.

All keyboard events, except the character input and tab key handling, 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.

API

Events

ParseSyntax TEXT, RESULT_ARRAY_REF

Called when syntax highlighting is requires - 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

Selects if the auto indenting feature is on.

Default value: 1

blockType INTEGER

Defines 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 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 physical position of the cursor

cursorX X

Selects physical horizontal position of the cursor

cursorY Y

Selects physical 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

Governs 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 default 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

Array of scalar pairs, that define words to be highlighted. The first item in the pair is an array of words; the second item is a color value.

hiliteChars ARRAY

Array of scalar pairs, that define characters to be highlighted. The first item in the pair is a string of characters; the second item is a color value.

hiliteREs ARRAY

Array of scalar pairs, that define character patterns to be highlighted. The first item in the pair is a perl regular expression; the second item is a color value.

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

Array of arrays with integer pairs, X and Y, where each represents a physical 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.

offset INTEGER

Horizontal offset of text lines in pixels.

persistentBlock BOOLEAN

Selects whether the selection is cancelled 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.

selection X1, Y1, X2, Y2

Accepts two pair of coordinates, (X1,Y1) the beginning and (X2,Y2) the end of new selection, and sets the block according to 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

Governs the syntax highlighting. Is not implemented for word wrapping mode.

tabIndent INTEGER

Maps tab (\t) key to `tabIndent` amount 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.

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 text string.

The property is more efficient than the *text* entry with the large text, because the copying of the text scalar to the stack stage is eliminated.

See also: the *text* entry.

topLine INTEGER

Selects the first line of the text drawn.

undoLimit INTEGER

Sets limit on 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 by the Tab key; however, this disallows the toolkit widget tab-driven navigation. If 0, the tab character can be entered by pressing 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 by the Enter key; however, this disallows the toolkit default button activation. If 0, the new line character can be entered by pressing Ctrl+Enter key combination.

Default value: 1

wordDelimiters STRING

Contains string of character that are used for locating a word break. Default STRING value consists of punctuation marks, space and tab characters, and \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 inferior borders. If 1, the *syntaxHilite* entry is not used. A line of text can be represented by more than one line of screen text (chunk) . To access the text chunk-wise, use the *get_chunk* entry method.

Methods

add_marker X, Y

Adds physical coordinated X,Y to 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 lines.

Default key: Backspace

begin_undo_group

Opens bracket for group of actions, undone as single operation. The bracket is closed by calling `end_undo_group`.

cancel_block

Removes the selection block

Default key: Alt+U

change_locked

Returns 1 if the logical locking is on, 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 bottom line

Default key: Ctrl+End

cursor_chome

Moves cursor to the top 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]

Delete 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 *markers* entry list.

delete_to_end

Removes 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, TEXT, LINE_ID, X, Y, COLOR

Paints the syntax-highlighted chunk of TEXT, taken from LINE_ID line index, at X, Y. COLOR is used if the syntax highlighting information contains `cl::Fore` as color reference.

end_block

Stops the block selection session.

end_undo_group

Closes bracket for group of actions, opened by `begin_undo_group`.

find SEARCH_STRING, [X = 0, Y = 0, REPLACE_LINE = '', OPTIONS]

Tries to find (and, if REPLACE_LINE is defined, to replace with it) SEARCH_STRING from (X,Y) physical coordinates. OPTIONS is an integer that consists of the `fdo::` constants; the same constants are used in the *Prima::EditDialog* section, which provides graphic interface to the find and replace facilities of the *Prima::Edit* section.

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 backwards.

fdo::ReplacePrompt

Not used in the class, however, used in the *Prima::EditDialog* section.

get_chunk CHUNK_ID

Returns chunk of text, located at CHUNK_ID. Returns empty string if chunk is nonexistent.

get_chunk_end CHUNK_ID

Returns the index of chunk at CHUNK_ID, corresponding to the last chunk of same line.

get_chunk_org CHUNK_ID

Returns the index of chunk at CHUNK_ID, corresponding to the first chunk of same line.

get_chunk_width TEXT, FROM, LENGTH, [RETURN_TEXT_PTR]

Returns the width in pixels of `substr( TEXT, FROM, LENGTH)`. If FROM is larger than length of TEXT, TEXT is padded with space characters. Tab character in TEXT replaced to the *tabIndent* entry times space character. If RETURN_TEXT_PTR pointer is specified, the converted TEXT is stored there.

get_line INDEX

Returns line of text, located at INDEX. Returns empty string if line is nonexistent.

get_line_dimension INDEX

Returns two integers, representing the line at INDEX in the *wordWrap* entry mode: the first value is the corresponding chunk index, the second is how many chunks represent the line.

See also: the *make_logical* entry.

get_line_ext CHUNK_ID

Returns the line, corresponding to the chunk index.

has_selection

Returns 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 cancelled 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 internal lock count hits zero, `Change` notification is called.

make_logical X, Y

Maps logical X,Y coordinates to the physical and returns the integer pair. Returns same values when the *wordWrap* entry is 0.

make_physical X, Y

Maps physical X,Y coordinates to the logical and returns the integer pair.

Returns same values when the *wordWrap* entry is 0.

mark_horizontal

Starts block marking session with `bt::Horizontal` block type.

Default key: Alt+L

mark_vertical

Starts block marking session with `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 in 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.

redo

Re-applies changes, formerly rolled back by `undo`.

set_line LINE_ID, TEXT, [OPERATION, FROM, LENGTH]

Changes line at LINE_ID to new TEXT. Hint scalars OPERATION, FROM and LENGTH 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 a character offset and LENGTH is a length of 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 if BLOCK_TYPE, if it is specified, or the *blockType* entry property value otherwise.

undo

Rolls back changes into internal array, which size cannot extend `undoLimit` value. In case `undoLimit` is 0, no undo actions can be made.

update_block

Adjusts the selection inside the block session, extending of shrinking it to the current cursor position.

word_left [REPEAT = 1]

Moves cursor REPEAT words to the left.

word_right [REPEAT = 1]

Moves cursor REPEAT words to the right.

4.9 Prima::ExtLists

Extended functionality for list boxes

Synopsis

```
use Prima::ExtLists;

my $vec = '';
vec( $vec, 0, 8) = 0x55;
Prima::CheckList-> new(
        items  => [1..10],
        vector => $vec,
);
```

Description

The module is intended to be a collection of list boxes with particular enhancements. Currently the only package is contained is `Prima::CheckList` class.

Prima::CheckList

Provides a list box, 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 by click and double click.

Properties

button INDEX, STATE

Runtime only. Sets INDEXth button STATE to 0 or 1. If 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 check 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

Standard frameset widget

Synopsis

```
use Prima::FrameSet;

my $frame = Prima::FrameSet->create(
        frameSizes => [qw(211 20% 123 10% * 45% *)],
        opaqueResize => 0,
        frameProfiles => [ 0,0, { minFrameWidth => 123, maxFrameWidth => 123 }],
);
$frame->insert_to_frame(
        0,
        Button =>
        text => '~Ok',
);
```

Description

Provides standard means of framesets manipulations. It includes sharing of common workspace among several widget groups; redistribution of space, occupied by frames; isolation of different frames from each other.

This module defines `fra::` and `frr::` packages for constants, used by the *arrangement* entry and the *resizeMethod* entry properties, respectively.

Two additional auxiliary packages are defined within this module: the *Prima::FrameSet::Frame* section and the *Prima::FrameSet::Slider* section.

4.11 Prima::Grids

Grid widgets

Synopsis

```
use Prima::Grids;

$grid = Prima::Grid-> create(
        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";
        }
);
```

Description

The module provides classes for several abstraction layers of grid representation. The classes hierarchy is as follows:

```
AbstractGridViewer
        AbstractGrid
        GridViewer
                Grid
```

The root class, `Prima::AbstractGridViewer`, provides common interface, while by itself it is not directly usable. The main differences between classes are centered around 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 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 in grid margins, and their cell are drawn with distinguished colors. An example use 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 with 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 interface to generic grid browsing functionality, plus functionality for text-oriented grids. The class is not usable directly.

`Prima::AbstractGridViewer` is a descendant of `Prima::GroupScroller`, and some properties are not described here. See the **Prima::GroupScroller** entry in the *Prima::IntUtils* section.

Properties

allowChangeCellHeight BOOLEAN

If 1, the user is allowed to change vertical extents of cells by dragging the horizontal grid lines. Prerequisites to the options are: the lines must be set visible via `drawHGrid` property, `constantCellHeight` property set to 0, and the changes to the vertical extents can be recorded via `SetExtent` notification.

Default value: 0

allowChangeCellWidth BOOLEAN

If 1, the user is allowed to change horizontal extents of cells by dragging the horizontal grid lines. Prerequisites to the options are: the lines must be set visible via `drawVGrid` property, `constantCellWidth` property set to 0, and the changes to the horizontal extents can be recorded via `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), cannot be selected and scrolled. X1 and X2 correspond to amount of indent columns, and Y1 and Y2, - to the indent rows.

`leftCell` and `topCell` do not count the indent cells as the leftmost or topmost visible cell; 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 governs the way clipping is applied when cells are drawn. Depending on 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 proceed column-wise. If the value is 2, the clipping as 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 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 width of a column. To acquire or set the width, `Measure` and `SetExtent` notifications can be invoked. Result of `Measure` may be cached internally using `cache_geometry_requests` method.

The width does not include 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 grid lines.

Default value: `cl::Black` .

gridGravity INTEGER

The property selects the breadth of area around the grid lines, that reacts on grid-dragging mouse events. The minimal value, 0, marks only grid lines as the drag area, but makes the dragging 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 indent cells.

Default value: `cl::Gray` .

indentCellColor

Selects the foreground color of indent cells.

Default value: `cl::Gray` .

leftCell INTEGER

Selects 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 number of rows, including the indent rows. The number of rows must be larger than the number of indent rows.

Default value: 0.

topCell

Selects index of the topmost visible normal cell.

rowHeight INTEGER

A run-time property, selects height of a row. To acquire or set the height, `Measure` and `SetExtent` notifications can be invoked. Result of `Measure` may be cached internally using `cache_geometry_requests` method.

The height does not include widths 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, governs the extents of a rectangular area, that contains selected cells. If no such area is present, selection is (-1,-1,-1,-1), and `has_selection` returns 0 .

If `multiSelect` is 0, in get-mode returns the focused cell, and discards the parameters in the set-mode.

Methods

cache_geometry_requests CACHE

If CACHE is 1, starts caching results of `Measure` notification, thus lighting the subsequent `columnWidth` and `rowHeight` calls; if CACHE is 0, flushes the cache.

If a significant geometry change was during the caching, the cache is not updated, so it is the caller's responsibility to flush the cache.

deselect_all

Nullifies the selection, if `multiSelect` is 1.

draw_cells CANVAS, COLUMNS, ROWS, AREA

A bulk draw routine, called from `onPaint` to draw cells. AREA is an array of four integers with inclusive-inclusive coordinates of the widget inferior without borders and scrollbars (result of `get_active_area(2)` call; see the **get_active_area** entry in the *Prima::IntUtils* 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 inclusive-inclusive coordinate system, and do not include eventual grid space, nor gaps between indent and normal cells. By default, internal arrays {`colsDraw`} and {`rowsDraw`} are passed as COLUMNS and ROWS parameters.

In `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, called from `std_draw_text_cells` .

SCREEN_RECTANGLES and CELL_RECTANGLES are arrays, where each item is a rectangle with exterior of a cell. SCREEN_RECTANGLES contains rectangles that cover the cell visible area; CELL_RECTANGLES contains rectangles that span the cell extents 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 pair where item 0 is column, and item 1 is row of the cell.

FONT_HEIGHT is a current font height value, cached since `draw_text_cells` is often used for text operations and may require vertical text justification.

get_cell_area [WIDTH, HEIGHT]

Returns screen area in inclusive-inclusive pixel coordinates, that 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_text COLUMN, ROW

Returns text string assigned to cell in COLUMN and ROW. Since the class does not assume the item storage organization, the text is queried via `Stringify` notification.

get_range AXIS, INDEX

Returns a pair of integers, minimal and maximal breadth of INDEXth column or row in pixels. If AXIS is 1, the rows are queried; if 0, the columns.

The method calls `GetRange` notification.

get_screen_cell_info COLUMN, ROW

Returns information about a cell in COLUMN and ROW, if it is currently visible. The returned parameters are indexed by `gsci::XXX` constants, and explained below:

```
gsci::COL_INDEX - visual column number where the cell displayed
gsci::ROW_INDEX - visual row number where the cell displayed
gsci::V_FULL    - cell is fully visible

gsci::V_LEFT    - 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      - 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 these in gsci::V_RECT.
```

If the cell is not visible, returns 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]

Return information about point X, Y in widget coordinates. The method returns two integers, CX and CY, with cell coordinates, and eventual HINTS hash, with more information about pixe localtion. If OMIT_GRID is set to 1 and the pixel belongs to a grid, the pixels is treated a part of adjacent cell. The call syntax:

```
( $CX, $CY, %HINTS) = $self->point2cell( $X, $Y);
```

If the pixel lies within cell boundaries by either coordinate, CX and/or CY are correspondingly set to cell column and/or row. When the pixel is outside 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 boundaries of a cell.

If -1, the coordinate is on the left/top to the cell body.

If +1, the coordinate is on the right/bottom to the cell body, but within the widget.

If +2, the coordinate is on the right/bottom to 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 left/top indent cell.

If +1, the cell is right/bottom indent cell.

`x_grid` **and** `y_grid`

If 1, the point is over 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 area.

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`. Select indices of cells adjacent to the grid line.

`x_gap` **and** `y_gap`

If 1, the point is within 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 over a grid line.

`exterior`

If 1, the point is in inoperable area or outside the widget boundaries.

redraw_cell X, Y

Repaints 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 assumption that each cell is drawn with two colors only, so the color switching can be reduced.

The routine itself paints the cells background, and calls `draw_text_cells` to draw text and/or otherwise draw the cell content.

For explanation of COLUMNS, ROWS, and AREA parameters see the *draw_cells* entry .

Events

DrawCell CANVAS, COLUMN, ROW, INDENT, @SCREEN_RECT, @CELL_RECT, SELECTED, FOCUSED

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 same as SCREEN_RECT, but calculated as if the cell is fully visible.

SELECTED and FOCUSED are boolean flags, if the cell must be drawn correspondingly in selected and focused states.

GetRange AXIS, INDEX, MIN, MAX

Puts minimal and maximal breadth of INDEXth column (AXIS = 0) or row (AXIS = 1) in corresponding MIN and MAX scalar references.

Measure AXIS, INDEX, BREADTH

Puts breadth in pixels of INDEXth column (AXIS = 0) or row (AXIS = 1) into BREADTH scalar reference.

This notification by default may be called from within `begin_paint_info/end_paint_info` brackets. To disable this feature set internal flag {`NoBulkPaintInfo`} to 1.

SelectCell COLUMN, ROW

Called when a cell with COLUMN and ROW coordinates is focused.

SetExtent AXIS, INDEX, BREADTH

Reports breadth in pixels of INDEXth column (AXIS = 0) or row (AXIS = 1), as a response to `columnWidth` and `rowHeight` calls.

Stringify COLUMN, ROW, TEXT_REF

Puts text string, assigned to cell with COLUMN and ROW coordinates, into TEXT_REF scalar reference.

Prima::AbstractGrid

Exactly the same as its ascendant, `Prima::AbstractGridViewer`, except that it does not propagate `DrawItem` message, assuming that the items must be drawn as text.

Prima::GridViewer

The class implements cells data and geometry storage mechanism, but leaves the cell data format to the programmer. The cells are accessible via `cells` property and several other helper routines.

The cell data are stored in an array, where each item corresponds to a row, and contains array of scalars, where each corresponds to a column. All data managing routines, that accept two-dimensional arrays, assume that the columns arrays are of the same widths.

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 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 number of columns.

rows INDEX

A read-only property; returns number of rows.

Methods

add_column CELLS

Inserts one-dimensional array of scalars to the end of columns.

add_columns CELLS

Inserts two-dimensional array of scalars to the end of columns.

add_row CELLS

Inserts one-dimensional array of scalars to the end of rows.

add_rows CELLS

Inserts 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 one-dimensional array of scalars as column OFFSET. Negative values are accepted.

insert_columns OFFSET, CELLS

Inserts two-dimensional array of scalars in column OFFSET. Negative values are accepted.

insert_row

Inserts one-dimensional array of scalars as row OFFSET. Negative values are accepted.

insert_rows

Inserts two-dimensional array of scalars in row OFFSET. Negative values are accepted.

Prima::Grid

Descendant of `Prima::GridViewer`, declares format of cells as a single text string. Incorporating all functionality of its ascendants, provides a standard text grid widget.

Methods

get_cell_text COLUMN, ROW

Returns text string assigned to cell in COLUMN and ROW. Since the item storage organization is implemented, does so without calling `Stringify` notification.

4.12 Prima::Header

A multi-tabbed 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 with a view 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 used 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 *vertical* entry property value) of the 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 call, all internal structures are updated.

SizeItem INDEX, OLD_EXTENT, NEW_EXTENT

Called when the user resizes a tab in INDEX position. OLD_EXTENT and NEW_EXTENT are either width or height of the tab, depending on the *vertical* entry property value.

SizeItems

Called when more than one tab has changed its extent. This might happen as a result of user action, as well as an effect of set-calling to some properties.

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 of the tabs.

Default value: 1

items ARRAY

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 *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 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 offset of the INDEXth tab (without regard to the *offset* entry property value).

tab2rect INDEX

Returns four integers, representing the rectangle area, occupied by the INDEXth tab (without regard to the *offset* entry property value).

4.13 Prima::HelpViewer

The built-in pod file browser

Usage

The module presents two packages, `Prima::HelpViewer` and `Prima::PodViewWindow`. Their sole purpose is to serve as a mediator between `Prima::PodView` package, the toolkit help interface and the user. `Prima::PodViewWindow` includes all the user functionality, including (but not limited to :) text search, color and font setup, printing etc. `Prima::HelpViewer` provides two methods - `open` and `close`, used by `Prima::Application` for help viewer invocation.

Help

The browser can be used to view and print POD (plain old documentation) files. See the command overview below for more detailed description:

File

Open

Presents a file selection dialog, when the user can select a file to browse in the viewer. The file must contain POD content, otherwise a warning is displayed.

Goto

Asks for a manpage, that is searched in PATH and the installation directories.

New window

Opens the new viewer window with the same context.

Run

Commands in this group call external processes

p-class

p-class is Prima utility for displaying the widget class hierachies. The command asks for Prima class to display the hierachy information for.

Print

Provides a dialog, when the user can select the appropriate printer device and its options.

Prints the current topic to the selected printer.

If 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 topic, and only one topic is displayed.

Find

Presents a find dialog, where the user can select the text to search and the search options - the search direction, scope, and others.

Find again

Starts search for the text, entered in the last 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 *Forward* entry option, except that the serach direction is backwards.

Repeat forward

Repeat the search in the same direction as the initial search was being invoked.

Repeat backward

Repeat the search in the reverse direction as the initial search was being invoked.

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 *Back* entry command.

Up

Displays the upper level topic within a manpage.

Previous

Moves to the previous topic within a manpage.

Next

Moves to the next topic within a manpage.

Help

About

Displays the information about the help viewer.

Help

Displays the information about the usage of the help viewer

4.14 Prima::Image::TransparencyControl

Standard dialog for transparent color index selection.

Description

The module contains two classes - `Prima::Image::BasicTransparencyDialog` and `Prima::Image::TransparencyControl`. The former provides a 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 a widget class that displays the image palette and allow color rather than index selection.

Prima::Image::TransparencyControl

Properties

index INTEGER

Selects the palette index.

image IMAGE

Selects image which palette is displayed, and the color index can be selected from.

Events

Change

Triggered when the user changes `index` property.

Prima::Image::BasicTransparencyDialog

Methods

transparent BOOLEAN

If 1, the transparent selection widgets are enabled, and the user can select the palette index. If 0, the widgets are disabled; the image file is saved with no transparent color index.

The property can be toggled interactively by a checkbox.

4.15 Prima::ImageViewer

Standard image, icon, and bitmap viewer class.

Description

The module contains `Prima::ImageViewer` class, which provides image displaying functionality, including different zoom levels.

`Prima::ImageViewer` is a descendant of `Prima::ScrollWidget` and inherits its document scrolling behavior and programming interface. See the *Prima::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`

image OBJECT

Selects the image object to be displayed. OBJECT can be an instance of `Prima::Image`, `Prima::Icon`, or `Prima::DeviceBitmap` class.

imageFile FILE

Set the image FILE to be loaded and displayed. Is rarely used since does not return a loading success flag.

quality BOOLEAN

A boolean flag, selecting 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.

NB: `ta::Middle` value is not equal to `ta::Center`'s, however the both constants produce equal effect here.

Default value: `ta::Bottom`

zoom FLOAT

Selects zoom level for image display. 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, is `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 `zoom` property. Minimal acceptable value is 10, where zoom 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 image of arbitrary zoom factor is requested to be displayed, the image sometimes must begin to be drawn from partial pixel - for example, 10x zoomed image shifted 3 pixels left, must be displayed so the first image pixel from the left occupies 3 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 ot offset -7 screen pixels. In case of a large image, such negative offsets become large, and the system will behave ineffectively trying to access all image pixels in system memory, slowing the drawing significantly, or in the worst case, failing the request. A workaround is to pre-calculate the zoom factor so that whatever image offset is requested, the negative screen offset will be fixed, and will impose fixed penalty on the system image scaling routine. For example, the default `zoomPrecision` value 100 means that for any given image offset, the screen offset will not exceed 100 pixels, and thus whatever the zoom factor is, the system will internally scale max. screen size / zoom factor + 100 pixels.

These considerations make sense for zoom factors greater than one only, but are applied also to those less than one for the consistency sake.

Default value: 100

Methods

screen2point X, Y, [X, Y, ...]

Performs translation of integer pairs integers as (X,Y)-points from widget coordinates to pixel offset in image coordinates. Takes in account zoom level, image alignments, and offsets. Returns array of same length as the input.

Useful for determining correspondence, for example, of a mouse event to a 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 in account zoom level, image alignments, and offsets. Returns array of same length as the input.

Useful for determining a screen location of an image point.

The reverse function is `screen2point`.

watch_load_progress IMAGE

When called, image viewer watches as the IMAGE is loaded (see the **load** entry in the *Prima::Image* section) and displays the progress. As soon IMAGE begins to load, it replaces the existing `image` property. Example:

```
$i = Prima::Image-> new;
$viewer-> watch_load_progress( $i);
$i-> load('huge.jpg');
$viewer-> unwatch_load_progress( $i);
```

Similar functionality is present in the *Prima::ImageDialog* section.

unwatch_load_progress CLEAR_IMAGE=1

Stops monitoring of image loading progress. If CLEAR_IMAGE is 0, the leftovers of the incremental loading stay intact in `image` propery. 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 `zoom` property.

4.16 Prima::InputLine

Standard input line widget

Description

The class provides basic functionality of an input line, including hidden input, read-only state, selection, and clipboard operations. The input line text data is contained in 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 direct call.

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, the keyboard `kb::Left` and `kb::Right` commands, if received when the cursor is at the beginning or at the end of text, and cannot be mover farther, not processed. 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 3d-shade border around the widget.

Default value: 2

charOffset INTEGER

Selects the position of the cursor in characters starting from the beginning of text.

firstChar

Selects the first visible character of text

insertMode BOOLEAN

Governs 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 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. A case with no selection is when START equals END.

selStart INTEGER

Selects the start of text selection.

selEnd INTEGER

Selects the end of text selection.

textRef SCALAR_REF

If not undef, contains reference to the scalar that holds the text of the input line. All changes to ::text property are reflected there. The direct write access to the scalar is not recommended because it leaves internal structures inconsistent, and the only way to synchronize structures is to set-call either ::textRef or ::text after every such change.

If undef, the internal text container is used.

Default value: undef

wordDelimiters STRING

Contains string of character that are used for locating a word break. Default STRING value consists of punctuation marks, space and tab characters, 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

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

4.17 Prima::KeySelector

Key combination widget and routines

Description

The module provides a standard widget for selecting a user-defined key combination. The widget class allows import, export, and modification of key combinations.

The module provides a set of routines, useful for conversion of a key combination between 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 **Hot key** entry in the *Prima::Menu* section, and is a combination of `km::XXX` key modifiers and either a `kb::XXX` virtual key, or a character code value.

The property allows almost, but not all possible combinations of key constants. Only `km::Ctrl`, `km::Alt`, and `km::Shift` modifiers are allowed.

Methods

All methods here can (and must) be called without the object syntax; - the first parameter must not be neither package nor widget.

describe KEY

Accepts KEY in integer format, and returns string description of the key combination in human readable format. Useful for supplying an accelerator text to a menu.

```
print describe( km::Shift|km::Ctrl|km::F10);
Ctrl+Shift+F10
```

export KEY

Accepts KEY in integer format, and returns string with a perl-evaluable expression, which after evaluation resolves to the original KEY value. Useful for storing a key into text config files, where value must be both human readable and easily passed to a program.

```
print 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 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 **KeyDown** entry in the *Prima::Widget* section event: code, key, and modifier. USE_CTRL is only relevant when KEY first byte (`KEY & 0xFF`) is between 1 and 26, what means that the key is a combination of an alpha key with the control key. If USE_CTRL is 1, code result is unaltered, and is in range 1 - 26. Otherwise, code result is converted to the character code (1 to ord('A'), 2 to ord('B') etc).

4.18 Prima::Label

Static text widget

Description

The class is designed for display of text, and assumes no user interaction. The text output capabilities include wrapping, horizontal and vertical alignment, and automatic widget resizing to match text extension. If text contains a tilde-escaped (hot) character, the label can explicitly focus the specified widget upon press of the character key, what feature is useful for dialog design.

Synopsis

```
my $label = Prima::Label-> create(
        text      => 'Enter ~name:',
        focusLink => $name_inputline,
        alignment => ta::Center,
);
```

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, which is explicitly focused when the user presses the combination of a hot key with the `Alt` key.

Prima::Label does not provide a separate property to access the hot key value, however it can be read from the {`accel`} variable.

Default value: `undef`.

showAccelChar BOOLEAN

If 0, the tilde (`~`) character is collapsed from the text, and the hot character is underlined. When the user presses combination of the escaped character with the `Alt` key, the `focusLink` widget is explicitly focused.

If 1, the text is showed as is, and no hot character is underlined. Key combinations with `Alt` key are not recognized.

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

wordWrap BOOLEAN

If 1, the text is wrapped if it can not be horizontally fit in the widget interior.

If 0, the text is not wrapped unless new line characters are present in the text.

New line characters signal line wrapping with no respect to `wordWrap` property value.

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.

NB: `ta::Middle` value is not equal to `ta::Center`'s, however the both constants produce equal effect here.

Default value: `ta::Top`

4.19 Prima::Lists

User-selectable item list widgets

Description

The module provides classes for several abstraction layers of item representation. The hierarchy of classes is as follows:

```
AbstractListViewer
        AbstractListBox
        ListViewer
                ProtectedListBox
                ListBox
```

The root class, `Prima::AbstractListViewer`, provides common interface, while by itself it is not directly usable. The main differences between classes are centered around the way the item list is 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 own mechanisms. For example, for a list of items that contain text strings and icons see the **Prima::DirectoryListBox** entry in the *Prima::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::GroupScroller`, and some properties are not described here. See the **Prima::GroupScroller** entry in the *Prima::IntUtils* section.

The class provides 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, item height is not changed; this is useful for non-text items.

Default value: 1

count INTEGER

An integer property, destined to reflect number of items in the list. Since it is tied to the item storage organization, and hence, to 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 control key together with left mouse button. If 0, item dragging is disabled.

Default value: 1

drawGrid BOOLEAN

If 1, vertical grid lines between columns are drawn with `gridColor`. Actual only in multi-column mode.

Default value: 1

extendedSelect BOOLEAN

Regards the way the user selects multiple items and is only actual when `multiSelect` is 1. If 0, the user must click each item in order to mark as selected. If 1, the user can drag mouse or use `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 on 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 widget does, via the `colorIndex` property. To achieve this, `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 items that are partially visible 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 items that are partially visible are drawn also. Actual only in 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 different dimensions, 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 different dimensions, 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, `focusedItem`'th item is not necessarily selected. To access selected item list, use `selectedItems` property.

Default value: 0

multiColumn BOOLEAN

If 0, the items are arrayed vertically in one column, and the main scroll bar is vertical. If 1, the items are arrayed in several columns, `itemWidth` pixels wide each. 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 number of selected items.

selectedItems ARRAY

ARRAY is an array of integer indices of selected items.

vertical BOOLEAN

Sets seneral 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 FLAG value, correspondingly 1 or 0.

Only for multi-select mode.

deselect_all

Removes selection from all items.

Only for multi-select mode.

draw_items CANVAS, ITEM_DRAW_DATA

Called from within `Paint` notification to draw items. The default behavior is to call `DrawItem` notification for every item in ITEM_DRAW_DATA array. ITEM_DRAW_DATA is an array or arrays, where each array consists of parameters, passed to `DrawItem` notification.

This method is overridden in some descendant classes, to increase the speed of drawing routine. For example, `std_draw_text_items` is the optimized routine for drawing unified text-based items. It is used in `Prima::ListBox` class.

See the *DrawItem* entry for parameters description.

draw_text_items CANVAS, FIRST, LAST, STEP, X, Y, OFFSET, CLIP_RECT

Called by `std_draw_text_items` to draw 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 array of four integers with inclusive-inclusive coordinates of the active clipping rectangle.

get_item_text INDEX

Returns text string assigned to INDEXth item. Since the class does not assume the item storage organization, the text is queried via `Stringify` notification.

get_item_width INDEX

Returns width in pixels of INDEXth item. Since the class does not assume the item storage organization, the value is queried via `MeasureItem` notification.

is_selected INDEX

Returns 1 if INDEXth item is selected, 0 if it is not.

item2rect INDEX, [WIDTH, HEIGHT]

Calculates and returns four integers with rectangle coordinates of INDEXth item within the widget. WIDTH and HEIGHT are optional parameters with pre-fetched dimension of the widget; if not set, the dimensions are queried by calling `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 point (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 INDICES array.

select_all

Selects all items.

Only for multi-select mode.

set_item_selected INDEX, FLAG

Sets selection flag of INDEXth item. If FLAG is 1, the item is selected. If 0, it is deselected.

Only for multi-select mode.

select_item INDEX

Selects INDEXth item.

Only for multi-select mode.

std_draw_text_items CANVAS, ITEM_DRAW_DATA

An optimized method, draws unified text-based items. It is fully compatible to `draw_items` interface, and is used in `Prima::ListBox` class.

The optimization is derived from the assumption that items maintain common background and foreground colors, that differ in selected and non-selected states only. The routine groups drawing requests for selected and non-selected items, and draws items with reduced number of calls to `color` property. While the background is drawn by the routine itself, the foreground (usually text) is delegated to the `draw_text_items` method, so the text positioning and eventual decorations would not require full rewrite of code.

ITEM_DRAW_DATA is an array of arrays of scalars, where each array contains parameters of `DrawItem` notification. See the *DrawItem* entry for parameters description.

toggle_item INDEX

Toggles selection of INDEXth item.

Only for multi-select mode.

unselect_item INDEX

Deselects INDEXth item.

Only for multi-select mode.

Events

Click

Called when the user presses 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 in accord with the dragging action.

DrawItem CANVAS, INDEX, X1, Y1, X2, Y2, SELECTED, FOCUSED

Called when an INDEXth item is to be drawn on CANVAS. X1, Y1, X2, Y2 designate the item rectangle in widget coordinates, where the item is to be drawn. SELECTED and FOCUSED are boolean flags, if the item must be drawn correspondingly in selected and focused states.

MeasureItem INDEX, REF

Puts width in pixels of INDEXth item into REF scalar reference. This notification must be called from within `begin_paint_info/end_paint_info` block.

SelectItem INDEX, FLAG

Called when the item changed 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

Puts text string, assigned to INDEXth item into TEXT_REF scalar reference.

Prima::AbstractListBox

Exactly the same as its ascendant, `Prima::AbstractListViewer`, except that it does not propagate `DrawItem` message, assuming that the items must be drawn as text.

Prima::ListViewer

The class implements items storage mechanism, but leaves the items format to the programmer. The items are accessible via `items` property and several other helper routines.

The class also defines the user navigation, by accepting character keyboard input and jumping to the items that have text assigned with the first letter that match the input.

`Prima::ListViewer` is derived from `Prima::AbstractListViewer`.

Properties

autoWidth BOOLEAN

Selects if the gross item width must be recalculated automatically when either the font changes or item list is changed.

Default value: 1

count INTEGER

A read-only property; returns number of items.

items ARRAY

Accesses the storage array of items. The format of items is not defined, it is merely treated as one scalar per index.

Methods

add_items ITEMS

Appends array of ITEMS to the end of the list.

calibrate

Recalculates all item widths and 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 width in pixels of INDEXth item from internal cache.

get_items INDICES

Returns 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 array context the item list is returned; in scalar - only the first item from the list. The semantic of the last call is naturally usable only for single item retrieval.

insert_items OFFSET, ITEMS

Inserts array of items at OFFSET index in the list. Offset must be a valid index; to insert items at the end of the list use `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 OFFSET index in the list. 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`, that applies certain protection for every item drawing session. Assuming that several item drawing routines can be assembled in one widget, `Prima::ProtectedListBox` provides a safety layer between these, so, for example, one drawing routine that selects a font or a color and does not care to restore the old value back, does not affect the outlook of the other items.

This functionality is implementing by overloading `draw_items` method and also all graphic properties.

Prima::ListBox

Descendant of `Prima::ListViewer`, declares format of items as a single text string. Incorporating all of functionality of its predecessors, provides a standard 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 text string assigned to INDEXth item. Since the item storage organization is implemented, does so without calling `Stringify` notification.

4.20 Prima::MDI

Top-level windows emulation classes

Description

MDI stands for Multiple Document Interface, and is a Microsoft Windows user interface, that consists of multiple non-toplevel windows belonging to an application window. The module contains classes that provide similar functionality; sub-window widgets realize a set of operations, close 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 exactly the same as `Prima::MDIOwner` but is a `Prima::Window` descendant: the both owner classes are different only in their first ascendants. Their second ascendant is `Prima::MDIMethods` package, that contains all the owner class functionality.

Usage of `Prima::MDI` class extends beyond the multi-document paradigm. `Prima::DockManager` module uses the class as a base of a dockable toolbar window class (see the *Prima::DockManager* section.

Synopsis

```
use Prima::MDI;

my $owner = Prima::MDIWindowOwner-> create();
my $mdi   = $owner-> insert( 'Prima::MDI');
$mdi-> client-> insert( 'Prima::Button' => centered => 1 );
```

Prima::MDI

Implements MDI window functionality. A subwindow widget consists of a titlebar, titlebar 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 toolbar-anchored popup menu. The default set of commands is as follows:

```
Close window              - Ctrl+F4
Restore window            - Ctrl+F5 or a double click on the titlebar
Maximize window           - Ctrl+F10 or a double click on the titlebar
Go to next MDI window     - Ctrl+Tab
Go to previous MDI window - Ctrl+Shift+Tab
Invoke popup menu         - Ctrl+Space
```

The class mimics API of `Prima::Window` class, and in some extent the *Prima::Window* section and this page share the same information.

Properties

borderIcons INTEGER

Selects window decorations, which are buttons on titlebar and the titlebar itself. Can be 0 or a combination of the following `mbi::XXX` constants, which are supreset of `bi::XXX` constants (see the **borderIcons** entry in the *Prima::Window* section) and are interchangeable.

```
mbi::SystemMenu  - system menu button with icon is shown
mbi::Minimize    - minimize button
```

```
mbi::Maximize    - maximize ( and eventual restore )
mbi::TitleBar    - window title
mbi::Close       - close button
mbi::All         - all of the above
```

Default value: `mbi::All`

borderStyle INTEGER

One of `bs::XXX` constants, selecting the window border style. The constants are:

```
bs::None      - no border
bs::Single    - thin border
bs::Dialog    - thick border
bs::Sizeable  - thick border with interactive resize capabilities
```

`bs::Sizeable` is an 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 runtime. When changing the client, the old client children are not reparented to the new client. The property cannot be used to set the client during the MDI window creation; use `clientClass` and `clientProfile` properties instead.

When setting new client object, note that is has to be named `MDIClient` and the window is automatically destroyed after the client is destroyed.

clientClass STRING

Assigns client widget class.

Create-only property.

Default value: `Prima::Widget`

clientProfile HASH

Assigns hash of properties, passed to the client during the creation.

Create-only property.

dragMode SCALAR

A three-state variable, which governs the visual feedback style when the user moves or resizes a window. If 1, the window is moved or resized simultaneously with the user 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 actually 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 Esc key or calling `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 minimized button image in normal state.

iconMax HANDLE

Selects maximized button image in normal state.

iconClose HANDLE

Selects close button image in normal state.

iconRestore HANDLE

Selects restore button image in normal state.

iconMinPressed HANDLE

Selects minimize button image in pressed state.

iconMaxPressed HANDLE

Selects maximize button image in pressed state.

iconClosePressed HANDLE

Selects close button image in pressed state.

iconRestorePressed HANDLE

Selects restore button image in pressed state.

tileable BOOLEAN

Selects whether the window is allowed to participate in cascading and tiling auto-arrangements, performed correspondingly by `cascade` and `tile` methods. If 0, the window is never positioned automatically.

Default value: 1

titleHeight INTEGER

Selects 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 governs 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 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()`.

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, window next to the lowest occupies the inferior rectangle of the lowest window, etc.

Only windows with `tileable` property set to 1 are processed.

client2frame X1, Y1, X2, Y2

Returns a rectangle that the window would occupy if its client rectangle is assigned to 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 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 array of sibling MDI windows.

maximize

Maximizes window. A shortcut for `windowState(ws::Maximized)`.

minimize

Minimizes window. A shortcut for `windowState(ws::Minimized)`.

post_action STRING

Posts an action to the windows; 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 following string commands: `min`, `max`, `restore`, `close`.

repaint_title [**STRING** = `title`]

Invalidates rectangle on the title bar, corresponding to STRING, which can be one of the following:

```
left    - redraw the menu button
right   - redraw minimize, maximize, and close buttons
title   - redraw the title
```

restore

Restores window to normal state from minimized or maximized state. A shortcut for `windowState(ws::Normal)`.

sizemove_cancel

Cancels active window moving or resizing session and returns the window to the state before the session.

tile

Arranges sibling MDI windows so they form a grid-like structure, where all windows occupy equal space, if possible.

Only windows with `tileable` property set to 1 are processed.

xy2part X, Y

Maps a point in (X,Y) coordinates into a string, corresponding to a part of the window: titlebar, button, or 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  - titlebar; the window is not moveable
title    - titlebar; 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 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 a window. Activation mark is usually resides on a window that contains keyboard focus.

The module does not provide the activation function; `select()` call is used instead.

Deactivate

Triggered when the user deactivates a window. Window is usually marked inactive, when it contains no keyboard focus.

The module does not provide the de-activation function; `deselect()` call is used instead.

WindowState STATE

Triggered when 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 several methods for a class that is to be used as a MDI windows owner. It is enough to add class inheritance to `Prima::MDIMethods` to use the functionality. This step, however, is not required for a widget to become a MDI windows owner; the package contains helper functions only, which mostly mirror the arrangement functions of `Prima::MDI` class.

mdi_activate

Repaints window title in all children MDI windows.

mdis

Returns array of children MDI windows.

arrange_icons

Same as `Prima::MDI::arrange_icons`.

cascade

Same as `Prima::MDI::cascade`.

tile

Same as `Prima::MDI::tile`.

Prima::MDIOwner

A predeclared descendant class of `Prima::Widget` and `Prima::MDIMethods`.

Prima::MDIWindowOwner

A pre-declared descendant class of `Prima::Window` and `Prima::MDIMethods`.

4.21 Prima::Notebooks

Multipage widgets

Description

The module contains several widgets useful for organizing multipage (notebook) containers. `Prima::Notebook` provides basic functionality of 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

```
my $nb = Prima::TabbedNotebook-> create(
        tabs => [ 'First page', 'Second page', 'Second page' ],
);
$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 );
```

Prima::Notebook

Properties

Provides basic widget container functionality. Acts as merely a grouping widget, hiding and showing the children widgets when `pageIndex` property is changed.

defaultInsertPage INTEGER

Selects the page where widgets, attached by `insert` call are assigned to. If set to `undef`, the default page is the current page.

Default value: `undef`.

pageCount INTEGER

Selects 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 list of WIDGETS to INDEXth page. The widgets are not necessarily must be children of the notebook widget. If the 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 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 `owner` property set to the caller widget, and returns the list of references to the newly created widgets.

See the **insert** entry in the *Prima::Widget* section for details.

insert_page [INDEX = -1]

Inserts a new empty page at INDEX. 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 ore more widgets to INDEXth page. The semantics of setting CLASS and PROFILE, as well as the return values are fully equivalent to `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 `insert` method.

See the **insert** entry in the *Prima::Widget* section for details.

move_widget WIDGET, INDEX

Moves WIDGET from its old page to 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 INDEXth page.

Events

Change OLD_PAGE_INDEX, NEW_PAGE_INDEX

Called when `pageIndex` value is changed from OLD_PAGE_INDEX to NEW_PAGE_INDEX. Current implementation invokes this notification while the notebook widget is in locked state, so no redraw requests are honored during the notification execution.

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 execution of the following code

```
$notebook-> pageIndex(1);
my $widget = $notebook-> insert_to_page( 0, ... );
$widget-> visible(0);
$notebook-> pageIndex(0);
```

`$widget` will still be visible. As a workaround, `widget_set` method can be suggested, to be called together with the explicit state calls. Changing

```
$widget-> visible(0);
```

code to

```
$notebook-> widget_set( $widget, visible => 0);
```

solves the problem, but introduces an inconsistency in API.

Prima::TabSet

`Prima::TabSet` class implements 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

firstTab INTEGER

Selects the first (leftmost) visible tab.

focusedTab INTEGER

Selects the currently focused tab. This property value is almost always equals to `tabIndex`, except when the widget is navigated by arrow keys, and 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 bottom.

Default value: 1

tabIndex INDEX

Selects the INDEXth tab. When changed, `Change` notification is triggered.

tabs ARRAY

Anonymous 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 INDEXth tab.

tab2firstTab INDEX

Returns the index of a tab, that will be drawn leftmost if INDEXth tab is to be displayed.

Events

Change

Triggered when `tabIndex` property is changed.

DrawTab CANVAS, INDEX, COLOR_SET, POLYGON1, POLYGON2

Called when INDEXth tab is to be drawn on CANVAS. COLOR_SET is an array reference, and consists of the four cached color values: foreground, background, dark 3d color, and light 3d color. POLYGON1 and POLYGON2 are array references, and contain four points as integer pairs in (X,Y)-coordinates. POLYGON1 keeps coordinates of the larger polygon of a tab, while POLYGON2 of the smaller. Text is displayed inside the larger polygon:

```
POLYGON1

     [2,3]         [4,5]
       o..........o
      .             .
[0,1].   TAB_TEXT   . [6,7]
    o................o

POLYGON2

 [0,1]                [2,3]
    o................o
[6,7]o..............o[4,5]
```

Depending on `topMost` property value, POLYGON1 and POLYGON2 change their mutual vertical orientation.

The notification is always called from within `begin_paint/end_paint` block.

MeasureTab INDEX, REF

Puts width of INDEXth tab in pixels into REF scalar value. This notification must be called from within `begin_paint_info/end_paint_info` block.

Prima::TabbedNotebook

The class combines functionality of `Prima::TabSet` and `Prima::Notebook`, providing the interactive multipage widget functionality. The page indexing scheme is two-leveled: the first level is equivalent to the `Prima::TabSet` - provided tab scheme. Each first-level tab, in turn, contains one or more second-level pages, which can be switched using native `Prima::TabbedNotebook` controls.

First-level tab is often referred as *tab*, and second-level as *page*.

Properties

defaultInsertPage INTEGER

Selects the page where widgets, attached by `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 hash of properties, passed to the notebook widget during the creation.

Create-only property.

notebookDelegations ARRAY

Assigns 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 `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 a tabset widget using the first-level tab numeration.

tabs ARRAY

Governs 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, with the following exception. To define second-level tabs, the same text string must be repeated as many times as many second-level tabs are desired. For example, the code

```
$nb-> tabs('1st', ('2nd') x 3);
```

results in 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 hash of properties, passed to the tab set widget during the creation.

Create-only property.

tabsetDelegations ARRAY

Assigns 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 second-level tab index, that corresponds to the INDEXth first-level tab.

page2tab INDEX

Returns first-level tab index, that corresponds to the INDEXth second-level tab.

Events

Change OLD_PAGE_INDEX, NEW_PAGE_INDEX

Triggered when `pageIndex` property is changes it s value from OLD_PAGE_INDEX to NEW_PAGE_INDEX.

4.22 Prima::Outlines

Tree view widgets

Synopsis

```
use Prima::Outlines;

my $outline = Prima::StringOutline-> create(
        items => [
                [  'Simple item' ],
                [[ 'Embedded item ']],
                [[ 'More embedded items', [ '#1', '#2' ]]],
        ],
);
$outline-> expand_all;
```

Description

The module provides a set of widget classes, designed to display a tree-like hierarchy of items. `Prima::OutlineViewer` presents a generic class that contains basic functionality and defines the interface for the descendants, which are `Prima::StringOutline`, `Prima::Outline`, and `Prima::DirectoryOutline`.

Prima::OutlineViewer

Presents a generic interface for browsing the tree-like lists. A node in a linked list represents each item. The format of node is predefined, and is an anonymous array with the following definitions of indices:

1. Item id with non-defined 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 `items` article of a concrete class for the format of a node record.

2. Reference to a child node. `undef` if there is none.

3. A boolean flag, which selects if the node shown as expanded, e.g. all its immediate children are visible.

4. 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. It should be enough item 0 to store any value.

To support a custom format of node, it is sufficient to overload the following notifications: `DrawItem`, `MeasureItem`, `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 addressing methods, 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 internal node list due to the speed reasons.

`Prima::OutlineViewer` is a descendant of `Prima::GroupScroller` and `Prima::MouseScroller`, so some properties and methods are not described here. See the *Prima::IntUtils* section for these.

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 control key together with left mouse button. If 0, item dragging is disabled.

Default value: 1

extendedSelect BOOLEAN

Regards the way the user selects multiple items and is only actual when `multiSelect` is 1. If 0, the user must click each item in order to mark as selected. If 1, the user can drag mouse or use `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 on this stage as well.

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 different vertical dimensions, 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 items is described in the opening article (see the *Prima::OutlineViewer* section).

Default value: []

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, `focusedItem`'th item is not necessarily selected. To access selected item list, use `selectedItems` property.

Default value: 0

offset INTEGER

Horizontal offset of an item list in pixels.

selectedItems ARRAY

ARRAY is an array of integer indices of 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`, `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 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 multi-select mode.

adjust INDEX, EXPAND

Performs expansion (1) or collapse (0) of INDEXth item, depending on EXPAND boolean flag value.

calibrate

Recalculates the node tree and the 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 `Paint` notification to draw items. The default behavior is to call `DrawItem` notification for every visible item. PAINT_DATA is an array of arrays, where each array consists of parameters, passed to `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 item index and the second is item depth level. If it is not visible, `-1, undef` is returned.

get_index_text INDEX

Returns text string assigned to INDEXth item. Since the class does not assume the item storage organization, the text is queried via `Stringify` notification.

get_index_width INDEX

Returns width in pixels of INDEXth item, which is a cached result of `MeasureItem` notification, stored under index #3 in node.

get_item INDEX

Returns two scalars corresponding to INDEXth item: node reference and its depth level. If INDEX is outside the list boundaries, empty array is returned.

get_item_parent NODE

Returns two scalars, corresponding to NODE: its parent node reference and offset of NODE in the parent's immediate children list.

get_item_text NODE

Returns text string assigned to NODE. Since the class does not assume the item storage organization, the text is queried via `Stringify` notification.

get_item_width NODE

Returns width in pixels of INDEXth item, which is a cached result of `MeasureItem` notification, stored under index #3 in 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 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 ACTION subroutine for each node. If 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:

1. Node reference
2. Parent node reference; if `undef`, the node is the root.
3. Node offset in parent item list.
4. Node index.
5. Node depth level. 0 means the root node.
6. A boolean flag, set to 1 if the node is the last child in parent node list, set to 0 otherwise.
7. Visibility index. When `iterate` is called with `FULL = 1`, the index is the item index as seen of 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 upon INDEXth item. If a boolean flag SHOW is set to 1, and `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 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 index of an item that contains 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 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 in the screen position. Only for multi-select mode.

select_item INDEX, ITEM

Selects an item.

The method can address the item either directly (ITEM) or by its INDEX in the screen position. Only for 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 in the screen position. Only for multi-select mode.

unselect_item INDEX, ITEM

Deselects an item.

The method can address the item either directly (ITEM) or by its INDEX in the screen position. Only for multi-select mode.

validate_items ITEMS

Traverses the array of ITEMS and puts every node to the common format: cuts scalars above index #3, if there are any, or 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 in accord with the dragging action.

DrawItem CANVAS, NODE, X1, Y1, X2, Y2, INDEX, SELECTED, FOCUSED

Called when INDEXth item, contained in NODE is to be drawn on CANVAS. X1, Y1, X2, Y2 coordinated define the exterior rectangle of the item in widget coordinates. SELECTED and FOCUSED boolean flags are set to 1 if the item is selected or focused, respectively; 0 otherwise.

MeasureItem NODE, REF

Puts width of NODE item in pixels into REF scalar reference. This notification must be called from within `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 set of arrays for each items, where the item can be defined either as integer INDEX, or directly as ITEM, or both. In case INDEX is undef, the item is invisible; if ITEM is undef, then the caller didn't bother to call `get_item` for the speed reasons, and the received should call this function. The SELECTED flag contains the new value of the item.

Stringify NODE, TEXT_REF

Puts text string, assigned to NODE item into TEXT_REF scalar reference.

Prima::StringOutline

Descendant of `Prima::OutlineViewer` class, provides standard single-text items widget. The items can be set by merely supplying a text as the first scalar in 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 neither format nor the amount 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 each item is mapped to a directory. Depending on the type of the host OS, there is either single root directory (unix), or one or more disk drive root items (win32, os2).

The format of a node is defined as follows:

1. Directory name, string.
2. Parent path; an empty string for the root items.
3. Icon width in pixels, integer.
4. Drive icon; defined only for the root items under non-unix hosts in order to reflect the drive type (hard, floppy, etc).

Properties

closedGlyphs INTEGER

Number of horizontal equal-width images, contained in `closedIcon` property.

Default value: 1

closedIcon ICON

Provides an icon representation for the collapsed items.

openedGlyphs INTEGER

Number of horizontal equal-width images, contained in `openedIcon` property.

Default value: 1

openedIcon OBJECT

Provides an icon representation for the expanded items.

path STRING

Runtime-only property. Selects current file system path.

showDotDirs BOOLEAN

Selects if the directories with the first dot character are shown the tree view. The treatment of the dot-prefixed names as hidden is traditional to unix, and is of doubtful use under win32 and os2.

Default value: 0

Methods

files [FILE_TYPE]

If 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 `showDotDirs` property value is 0, the dot-prefixed names are not included.

Used internally inside `Expand` notification.

4.23 Prima::PodView

POD browser widget

Synopsis

```
use Prima qw(Application);
use Prima::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 viewer of POD content. It heavily employs its ascendant class the *Prima::TextView* section, and is in turn base for the toolkit's default help viewer the *Prima::HelpViewer* section.

Usage

The package consists of the 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 content loading functions:

open_read

Clears the current content and enters the reading mode. In this mode the content can be appended by calling the *read* entry that pushes the raw POD content to the parser.

read TEXT

Supplies TEXT string to the parser. Manages basic indentation, but the main formatting is performed inside the *add* entry and the *add_formatted* entry

Must be called only within open_read/close_read brackets

add TEXT, STYLE, INDENT

Formats TEXT string of a given STYLE (one of `STYLE_XXX` constants) with INDENT space.

Must be called only within open_read/close_read brackets.

add_formatted Format, TEXT

Adds a pre-formatted TEXT with a given Format, supplied by `=begin` or `=for` POD directives. Prima::PodView understands 'text' and 'podview' FORMATs; the latter format is for Prima::PodView itself and contains small number of commands, aimed at inclusion of images into the document.

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 cancelled. It is used in conjunction with the following command, the *img* entry, to allow a POD manpage 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 if scaling is required, `width` and `height` options can be set. When the image is a multiframe image, the frame index can be set by `frame` option. Special `cut` option, if set to a true value, activates the the *cut* entry behavior if (and only if) the image load operation was unsuccessful. This make 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 would be selected.

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 `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 an a document model, generated by open_read/close_read session. The model is a set of same text blocks defined by the *Prima::TextView* section, except that the header length is only three integers:

```
M_INDENT       - the block X-axis indent
M_TEXT_OFFSET  - same as BLK_TEXT_OFFSET
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 to the full text blocks, wrapped and pushed into TextView-provided storage. In parallel, links and the corresponding event rectangles are calculated on this stage.

Topics

Prima::PodView provides the `::topicView` property, which governs whether the man page is viewed by topics or as a whole. When it is viewed as topics, {`modelRange`} array selects the model blocks that include the topic. Thus, having a single model loaded, text blocks change dynamically.

Topics contained in {`topics`} array, each is an array with indices of `T_XXX` constants:

```
T_MODEL_START - beginning of topic
T_MODEL_END   - length of a topic
T_DESCRIPTION - topic name
T_STYLE       - STYLE_XXX constant
T_ITEM_DEPTH  - depth of =item recursion
T_LINK_OFFSET - offset in links array, started in the topic
```

Styles

`::styles` property provides access to the styles, applied to different pod text parts. These styles are:

```
STYLE_CODE   - style for pre-formatted text and C<>
STYLE_TEXT   - normal text
STYLE_HEAD_1 - =head1
STYLE_HEAD_2 - =head2
STYLE_ITEM   - =item
STYLE_LINK   - style for L<> text
```

Each style is a hash with the following keys: `fontId`, `fontSize`, `fontStyle`, `color`, `backColor`, fully analogous to the tb::BLK_DATA_XXX options. This functionality provides another layer of accessibility to the pod formatter.

In addition to styles, Prima::PodView defined `colormap` entries for `STYLE_LINK` and `STYLE_CODE`:

```
COLOR_LINK_FOREGROUND
COLOR_LINK_BACKGROUND
COLOR_CODE_FOREGROUND
COLOR_CODE_BACKGROUND
```

The default colors in the styles are mapped into these entries.

Link and navigation methods

Prima::PodView provides a hand-icon mouse pointer highlight for the link entries and as an interactive part, the link documents or topics are loaded when the user presses the mouse button on the link. The mechanics below that is as follows. {`contents`} of event rectangles, (see the *Prima::TextView* section) is responsible for distinguishing whether a mouse is inside a link or not. When the link is activated, `link_click` is called, which, in turn, calls `load_link` method. If the page is loaded successfully, depending on `::topicView` property value, either `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 a 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 format of *perlpod* `L<>` link: "manpage/section". Loads the manpage, if necessary, and selects the section.

load_bookmark BOOKMARK

Loads a bookmark string, prepared by the *make_bookmark* entry function. Used internally.

load_content CONTENT

Loads content into the viewer. Returns `undef` is there is no POD context, 1 otherwise.

make_bookmark [**WHERE**]

Combines the information about the currently viewing manpage, 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 '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 a manpage.

If the location cannot be stored or defined, `undef` is returned.

4.24 Prima::ScrollBar

Standard scroll bars class

Description

The class `Prima::ScrollBar` implements both vertical and horizontal scrollbars in *Prima*. This is a purely Perl class without any C-implemented parts except those inherited from `Prima::Widget`.

Synopsis

```
use Prima::ScrollBar;

my $sb = Prima::ScrollBar-> create( 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 `Change` notification during mouse tracking events. Generally it should only be set to 0 on very slow computers.

Default value is 1 (logical true).

growMode INTEGER

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

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`.

Default value: 100.

min INTEGER

Sets the lower limit for `value`.

Default value: 0

minThumbSize INTEGER

A minimal thumb breadth in pixels. The thumb cannot have main dimension lesser than this.

Default value: 21

pageStep INTEGER

This determines the increment/decrement to `value` during "page"-related operations, for example clicking the mouse on the strip outside the thumb, or pressing `PgDn` or `PgUp`.

Default value: 10

partial INTEGER

This tells the scrollbar how many of imaginary units the thumb should occupy. See `whole` below.

Default value: 10

selectable BOOLEAN

Default value is 0 (logical false). If set to 1 the widget receives keyboard focus; when in focus, the thumb is blinking.

step INTEGER

This determines the minimal increment/decrement to `value` during mouse/keyboard interaction.

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.

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.

Default value is 0 (logical false).

whole INTEGER

This tells the scrollbar how many of 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 of the proportion between the visible size of something (document, for example) and the whole size of that "something". Useful to struggle against infamous "bird" size of Windows scrollbar thumbs.

Default value is 100.

Methods

get_default_size

Returns two integers, the default platform dependant width of a vertical scrollbar and height of a horizontal scrollbar.

Events

Change

The `Change` notification is sent whenever the thumb position of scrollbar is changed, subject to a certain limitations when `autoTrack` is 0. The notification conforms the general *Prima* rule: it is sent when appropriate, regardless to whether this was a result of user interaction, or a side effect of some method programmer has called.

Track

If `autoTrack` is 0, called when the user changes the thumb position by the mouse instead of `Change`.

Example

```
use Prima;
use Prima::Application name => 'ScrollBar test';
use Prima::ScrollBar;

my $w = Prima::Window-> create(
        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;
```

4.25 Prima::ScrollWidget

Scrollable generic document widget.

Description

`Prima::ScrollWidget` is a simple class that declares two pairs of properties, *delta* and *limit* for vertical and horizontal axes, which define a a virtual document. *limit* is the document dimension, and *delta* is the current offset.

`Prima::ScrollWidget` is a descendant of `Prima::GroupScroller`, and, as well as its ascendant, provides same user navigation by two scrollbars. The scrollbars' `partial` and `whole` properties are maintained if the document or widget extensions change.

API

Properties

deltas X, Y

Selects horizontal and vertical document offsets.

deltaX INTEGER

Selects horizontal document offset.

deltaY INTEGER

Selects vertical document offset.

limits X, Y

Selects horizontal and vertical document extensions.

limitX INTEGER

Selects horizontal document extension.

limitY INTEGER

Selects vertical document extension.

Events

Scroll DX, DY

Called whenever the client area is to be scrolled. The default action calls `Widget::scroll` .

4.26 Prima::Sliders

Sliding bars, spin buttons and input lines, dial widget etc.

Description

The module is a set of widget classes, with one common property; - all of these provide input and / or output of an integer value. This property unites the following set of class hierarchies:

```
Prima::AbstractSpinButton
        Prima::SpinButton
        Prima::AltSpinButton

Prima::SpinEdit

Prima::Gauge

Prima::AbstractSlider
        Prima::Slider
        Prima::CircularSlider
```

Prima::AbstractSpinButton

Provides a generic interface to spin-button class functionality, which includes range definition properties and events. Neither `Prima::AbstractSpinButton`, nor its descendants store the integer value. These provide a mere possibility for the user to send incrementing or decrementing commands.

The class is not usable directly.

Properties

state INTEGER

Internal state, reflects widget modal state, for example, is set to non-zero when the user performs a mouse drag action. The exact meaning of `state` is defined in the descendant classes.

Events

Increment DELTA

Called when the user presses a part of a widget that is responsible for incrementing or decrementing commands. DELTA is an integer value, indicating how the associated value must be modified.

TrackEnd

Called when the user finished the mouse transaction.

Prima::SpinButton

A rectangular spin button, 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 mouse, fires `Increment` events with delta value, based on a vertical position of the mouse pointer.

Prima::AltSpinButton

A rectangular spin button, consists of two push-buttons, associated with singular increment and decrement command. Comparing to `Prima::SpinButton`, the class is less functional but has more stylish look.

Prima::SpinEdit

The class is a numerical input line, paired with a spin button. The input line value can be change three ways - either as a direct traditional keyboard input, or as spin button actions, or as mouse wheel response. The class provides value storage and range selection properties.

Properties

circulate BOOLEAN

Selects the value modification rule when the increment or decrement action hits the range. If 1, the value is changed to the opposite limit value (for example, if value is 100 in range 2-100, and the user clicks on 'increment' button, the value is changed to 2).

If 0, the value does not change.

Default value: 0

editClass STRING

Assigns an input line class.

Create-only property.

Default value: `Prima::InputLine`

editDelegations ARRAY

Assigns the input line list of delegated notifications.

Create-only property.

editProfile HASH

Assigns hash of properties, passed to the input line during the creation.

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/decrementing actions of the mouse wheel.

Default value: 10

spinClass STRING

Assigns a spin-button class.

Create-only property.

Default value: `Prima::AltSpinButton`

spinProfile ARRAY

Assigns the spin-button list of delegated notifications.

Create-only property.

spinDelegations HASH

Assigns hash of properties, passed to the spin-button during the creation.

Create-only property.

step INTEGER

Determines the multiplication factor for incrementing/decrementing actions of the spin-button.

Default value: 1

value INTEGER

Selects integer value in range from `min` to `max`, 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 width of a 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 a 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 risen look
```

Default value: `gr::Sink`.

threshold INTEGER

Selects the threshold value used to determine if the changes to `value` are reflected immediately or 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 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 integer VALUE into a string format and puts into REF scalar reference. Default stringifying conversion is identical to `sprintf("%2d%%")` one.

Prima::AbstractSlider

The class provides basic functionality of a sliding bar, equipped with tick marks. Tick marks are supposed to be drawn alongside the main sliding axis or circle and provide visual feedback for the user.

The class is not usable directly.

Properties

autoTrack BOOLEAN

A boolean flag, selects the way notifications execute when the user mouse-drags the sliding bar. If 1, `Change` notification is executed as soon as `value` is changed. If 0, `Change` is deferred until the user finished the mouse drag; instead, `Track` notification is executed when the bar is moved.

This property can be used when the action, called on `Change` performs very slow, 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 use 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 circle. ARRAY consists of hashes, each for one tick. The hash must contain at least `value` key, with integer value. The two additional keys, `height` and `text`, select the height of a tick mark in pixels and the text drawn near the mark, correspondingly.

If ARRAY is `undef`, no ticks are drawn.

scheme INTEGER

`scheme` is a write-only property, that creates a set of tick marks using one of the predefined scale designs, selected by `ss::XXX` constants. Each constant produces different scale; some make use of `increment` integer property, which selects a step by which the additional text marks are drawn. As an example, `ss::Thermometer` design with default `min`, `max`, and `increment` values would look like that:

```
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
```

snap BOOLEAN

If 1, `value` cannot accept values that are not on the tick scale. When set such a value, it is rounded to the closest tick mark. If 0, `value` can accept any integer value in range from `min` to `max`.

Default value: 0

step INTEGER

Integer delta for singular increment / decrement commands and a threshold for `value` when `snap` value is 0.

Default value: 1

value INTEGER

Selects integer value between `min` and `max` and the corresponding sliding bar position.

Default value: 0.

Events

Change

Called when `value` value is changed, with one exception: if the user moves the sliding bar while `autoTrack` is 0, `Track` notification is called instead.

Track

Called when the user moves the sliding bar while `autoTrack` value is 0; this notification is a substitute to `Change`.

Prima::Slider

Presents a linear sliding bar, movable along a linear shaft.

Properties

ribbonStrip BOOLEAN

If 1, the parts of shaft are painted with different colors, to increase visual feedback. If 0, the shaft is painted with single default background color.

Default value: 0

shaftBreadth INTEGER

Breadth of the shaft in pixels.

Default value: 6

tickAlign INTEGER

One of `tka::XXX` constants, that correspond to the situation of tick marks:

```
tka::Normal        - ticks are drawn on the left or on 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 dependant on `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 the dial and two increment / decrement buttons. The tick marks are drawn around the perimeter of the dial; 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 range from `min` to `max` into the corresponding angle, and return two real values: cosine and sine of the angle.

offset2pt X, Y, VALUE, RADIUS

Converts integer value in 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 (X,Y) format.

xy2val X, Y

Converts widget coordinates X and Y into value in range from `min` to `max`, and return two scalars: the value and the boolean flag, which is set to 1 if the (X,Y) point is inside the dial circle, and 0 otherwise.

Events

Stringify VALUE, REF

Converts integer VALUE into a string format and puts into REF scalar reference. The resulting string is displayed in the center of the dial.

Default conversion routine simply copies VALUE to REF as is.

4.27 Prima::StartupWindow

A simplistic startup banner window

Description

The module, when imported by `use` call, creates a temporary window which appears with 'loading...' text while the modules required by a program are loading. The window parameters can be modified by passing custom parameters after `use Prima::StartupWindow` statement, which are passed to `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::StartupWindow; # the window is created here

use Prima::Buttons;
.... # lots of 'use' of other modules

no Prima::StartupWindow; # the window is discarded here
```

4.28 Prima::TextView

Rich text browser widget

Description

Prima::TextView accepts blocks of formatted text, and provides basic functionality - scrolling and user selection. The text strings are stored as one large text chunk, available by the `::text` and `::textRef` properties. A block of a formatted text is an array with fixed-length header and the following instructions.

A special package `tb::` provides the block constants and simple functions for text block access.

Capabilities

Prima::TextView is mainly the text block functions and helpers. It provides function for wrapping text block, calculating block dimensions, drawing and converting coordinates from (X,Y) to a block position. Prima::TextView is centered around the text functionality, and although any custom graphic of arbitrary complexity can be embedded in a text block, the internal coordinate system is used (TEXT_OFFSET, BLOCK), where TEXT_OFFSET is a text offset from the beginning of a block and BLOCK is an index of a block.

The functionality does not imply any text layout - this is up to the class descendants, they must provide they own layout policy. The only policy Prima::TextView requires is that 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 includes change of color, backColor and font, and the painting of text strings. Other types of graphics can be achieved by supplying custom code.

Block header

A block's fixed header consists of `tb::BLK_START - 1` integer scalars, each of those 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 second group is enclosed in `tb::BLK_DATA_START` - `tb::BLK_DATA_END` range, like the whole header is contained in 0 - `tb::BLK_START - 1` range. This is done for the backward compatibility, if the future development changes the length of the header.

The first group fields define the text block dimension, aperture position and text offset (remember, the text is stored as one big chunk). The second defines the initial color and font settings. Prima::TextView needs all fields of every block to be initialized before displaying. the *block_wrap* entry method can be used for automated assigning of these fields.

Block parameters

The scalars, beginning from `tb::BLK_START`, represent the commands to the renderer. These commands have their own parameters, that follow the command. The length of a command is located in `@oplen` array, and must not be changed. 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 offset `tb::BLK_TEXT_OFFSET + TEXT_OFFSET`, with a length TEXT_LENGTH. The third parameter TEXT_WIDTH contains the width of the text in pixels. Such the two-part offset scheme is made for simplification or an imaginary code, that would alter (insert to, or delete part of) the big text chunk; the updating procedure would not need to traverse all commands, but just 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 `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 `tb::COLOR_INDEX` flags, in such case it is an index in `::colormap` property array.

Relative to: `tb::BLK_COLOR`, `tb::BLK_BACKCOLOR`.

OP_FONT - KEY, VALUE

As the font is a complex property, that itself includes font name, size, direction, etc keys, `OP_FONT` KEY represents one of the three parameters - `tb::F_ID`, `tb::F_SIZE`, `tb::F_STYLE`. All three have different VALUE meaning.

Relative to: `tb::BLK_FONT_ID`, `tb::BLK_FONT_SIZE`, `tb::BLK_FONT_STYLE`.

F_STYLE

Contains a combination of `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 widget's font size. As such, 0 is a default value, and -2 is the widget's 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 vary when relying on the combined font size value .

If `F_SIZE` value is added to a `F_HEIGHT` constant, then it is treated as a 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 a index in the `::fontPalette` property array, which contains font hashes with the other font keys initialized - name, encoding, and pitch. These three are minimal required set, and the other font keys can be also selected.

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 in the account. If FLAGS does not contain `tb::X_EXTEND`, then in addition to the block expansion, current coordinate is also

moved to (X,Y). In this regard, `(OP_TRANSPOSE,0,0,0)` and `(OP_TRANSPOSE,0,0,X_EXTEND)` are identical and are empty operators.

There are formatting-only flags,in effect with the *block_wrap* entry function. `X_DIMENSION_FONT_HEIGHT` indicates that (X,Y) values must be multiplied to the current font height. Another flag `X_DIMENSION_POINT` does the same but multiplies by current value of the *resolution* entry property divided by 72 (basically, treats X and Y not as pixel but 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 re-written every time the new graphic is invented. As an example, see how the *Prima::PodView* section deals with the images.

OP_CODE - SUB, PARAMETER

Contains a custom code pointer SUB with a parameter PARAMETER, passed when a 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 a block. The format of the state is the same as of text block, so one may notice that for readability F_ID, F_SIZE, F_STYLE constants are paired to BLK_FONT_ID, BLK_FONT_SIZE and BLK_FONT_STYLE.

The SUB code is executed only when the block is about to draw.

OP_WRAP ON_OFF

`OP_WRAP` is only in effect in the *block_wrap* entry method. ON_OFF is a boolean flag, selecting if the wrapping is turned on or off. the *block_wrap* entry does not support stacking for the wrap commands, so the `(OP_WRAP,1,OP_WRAP,1,OP_WRAP,0)` has same effect as `(OP_WRAP,0)`. If ON_OFF is 1, wrapping is disabled - all following commands treated an non-wrapable until `(OP_WRAP,0)` is met.

OP_MARK PARAMETER, X, Y

`OP_MARK` is only in effect in 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, easy marking the geometrical positions that undergo the block wrapping.

As can be noticed, these opcodes are far not enough for the full-weight rich text viewer. However, the new opcodes can be created using `tb::opcode`, that accepts the opcode length and returns the new opcode value.

Rendering methods

block_wrap

`block_wrap` is the function, that is used to wrap a block into a given width. It returns one or more text blocks with fully assigned headers. The returned blocks are located one below another, providing an illusion that the text itself is wrapped. It does not only traverses the opcodes and sees if the command fit or not 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 `OP_WRAP` command brackets. The commands inside these brackets are not wrapped; `OP_WRAP` commands are removed from the output blocks.

In general, `block_wrap` copies all commands and their parameters as is, (as it is supposed to do), but some commands are treated especially:

- `OP_TEXT`'s third parameter, `TEXT_WIDTH`, is disregarded, and is recalculated for every `OP_TEXT` met.

- If `OP_TRANSPOSE`'s third parameter, `X_FLAGS` contains `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.

- `OP_MARK`'s second and third parameters assigned to the current (X,Y) coordinates.

- `OP_WRAP` removed from the output.

block_draw CANVAS, BLOCK, X, Y

The `block_draw` draws BLOCK onto CANVAS in screen coordinates (X,Y). It can not only be used for drawing inside begin_paint/end_paint brackets; CANVAS can be an arbitrary `Prima::Drawable` descendant.

Coordinate system methods

Prima::TextView employs two its own coordinate systems: (X,Y)-document and (TEXT_OFFSET,BLOCK)-block.

The document coordinate system is isometric and measured in pixels. Its origin is located into the imaginary point of the beginning of the document (not of the first block!), in the upper-left point. X increases to the right, Y increases downwards. The block header values BLK_X and BLK_Y are in document coordinates, and the widget's pane extents (regulated by `::paneSize`, `::paneWidth` and `::paneHeight` properties) are also in document coordinates.

The block coordinate system in an-isometric - 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 described different coordinate system converters

screen2point X, Y

Accepts (X,Y) in the screen coordinates (O is a lower left widget corner), returns (X,Y) in document coordinates (O is upper left corner of a document).

xy2info X, Y

Accepts (X,Y) is document coordinates, returns (TEXT_OFFSET,BLOCK) coordinates, where TEXT_OFFSET is text offset from the beginning of a block (not related to the big text chunk) , 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 X coordinate where TEXT_OFFSET begins in a BLOCK index.

info2text_offset

Accepts (TEXT_OFFSET,BLOCK) coordinates and returns the text offset with regard to the big text chunk.

text_offset2info TEXT_OFFSET

Accepts big text offset and returns (TEXT_OFFSET,BLOCK) coordinates

text_offset2block TEXT_OFFSET

Accepts big text offset and returns BLOCK coordinate.

Text selection

The text selection is performed automatically when the user selects the region with a 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) 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 `Ctrl+Insert` key combination.

Event rectangles

Partly as an option for future development, partly as a hack a concept of 'event rectangles' was introduced. Currently, {`contents`} private variable points to an array of objects, equipped with `on_mousedown`, `on_mousemove`, and `on_mouseup` methods. These are called within the widget's mouse events, so the overloaded classes can define the interactive content without overloading the actual mouse events (which is although easy but is dependent on Prima::TextView own mouse reactions).

As an example the *Prima::PodView* section uses the event rectangles to catch the mouse events over the document links. Theoretically, every 'content' is to be bound with a separate logical layer; when the concept was designed, a html-browser was in mind, so such layers can be thought as (in the html world) links, image maps, layers, external widgets.

Currently, `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.

4.29 Prima::Themes

Object themes management

Description

Provides layer for theme registration in Prima. Themes are loosely grouped alternations of default class properties and behavior, by default stored in `Prima/themes` subdirectory. The theme realization is implemented as interception of object profile during its creation, inside `::profile_add`. Various themes apply various alterations, one way only - once an object is applied a theme, it cannot be neither changed nor revoked thereafter.

Theme configuration can be stored in an rc file, *~/.prima/themes*, and is loaded automatically, unless `$Prima::Themes::load_rc_file` 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
```

`Prima::Themes` namespace provides registration and execution functionality. `Prima::Themes::Proxy` is a class for overriding certain methods, for internal realization of a theme.

For interactive theme selection use *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 object with another theme while 'cyan' is active
Class->create( theme => 'yellow');
# remove a theme
Prima::Themes::uninstall('cyan');
```

Prima::Themes

load @THEME_MODULES

Load THEME_MODULES from files via `use` clause, dies on error. Can be used instead of explicit `use`.

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, second item of the matched pair.

If $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 $INSTALLER subroutine is passed, it is called during install and uninstall, with two parameters, the name of the theme and boolean install/uninstall flag. When install flag is 1, the theme is about to be installed; the subroutine is expected to return a boolean success flag. Otherwise, subroutine 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 previosuly loaded and registered loaded THEMES; the installed themes are now used 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 keys from $PROFILE_DEFAULT.

load_rc [$INSTALL = 1]

Reads data *~/.prima/themes* and loads listed modules. If $INSTALL = 1, installs the themes from the rc file.

save_rc

Writes configuration of currently installed themes into rc file, returns success flag. If success flag is 0, `$!` contains the error.

Prima::Themes::Proxy

An instance of `Prima::Themes::Proxy`, created as

Prima::Themes::Proxy-> new($OBJECT)

is a non-functional wrapper for any Perl object $OBJECT. All methods of $OBJECT, except `AUTOLOAD`, `DESTROY`, and `new`, are forwarded to $OBJECT itself transparently. The class can be used, for example, to deny all changes to `lineWidth` inside object's painting routine:

```
package ConstLineWidth;
use vars qw(@ISA);
@ISA = qw(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

5 Standard dialogs

5.1 Prima::ColorDialog

Standard color selection facilities

Synopsis

```
use Prima qw(StdDlg Application);

my $p = Prima::ColorDialog-> create(
        quality => 1,
);
printf "color: %06x", $p-> value if $p-> execute == mb::OK;
```

Description

The module contains two packages, `Prima::ColorDialog` and `Prima::ColorComboBox`, used as standard tools for interactive color selection. `Prima::ColorComboBox` is a modified combo widget, which provides selecting from predefined palette but also can invoke `Prima::ColorDialog` window.

Prima::ColorDialog

Properties

quality BOOLEAN

Used to increase 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 color from HSV to RGB format and returns three 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.

rgb2value RED, GREEN, BLUE

Combines separate channels into single 24-bit RGB value and returns the result.

value2rgb COLOR

Splits 24-bit RGB value into three channels, red, green, and blue and returns three integer values.

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 error flag is set, the conversion has failed.

hs2xy HUE, SATURATION

Maps hue and saturation onto 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 number of SHADES given, drawn on a BACK_COLOR background, and returns a `Prima::DeviceBitmap` object.

create_wheel_shape SHADES

Creates a circular 1-bit mask, with radius derived from SHAPES. SHAPES must be same as passed to the *create_wheel* entry. Returns `Prima::Image` object.

Events

BeginDragColor $PROPERTY

Called when the user starts dragginh a color from the color wheel by with left mouse button and combination of Alt, Ctrl, and Shift keys. $PROPERTY is one of `Prima::Widget` color properties, and depends on combination of keys:

```
Alt              backColor
Ctrl             color
Alt+Shift        hiliteBackColor
Ctrl+Shift       hiliteColor
Ctrl+Alt         disabledColor
Ctrl+Alt+Shift   disabledBackColor
```

Default action reflects the property to be changes in the dialog title

Change

The notification is called when the the *value* entry property is changed, either interactively or as a result of direct call.

EndDragColor $PROPERTY, $WIDGET

Called when the user releases the mouse drag over a Prima widget. Default action sets `$WIDGET->$PROPERTY` to the current color value.

Variables

$colorWheel

Contains cached result of the *create_wheel* entry call.

$colorWheelShape

Contains cached result of the *create_wheel_shape* entry call.

Prima::ColorComboBox

Events

Colorify INDEX, COLOR_PTR

`nt::Action` callback, designed to map combo palette index into a RGB color. INDEX is an integer from 0 to the *colors* entry - 1, COLOR_PTR is a reference to a result scalar, where the notification is expected to write the resulting color.

Properties

colors INTEGER

Defines amount of colors in the fixed palette of the combo box.

value COLOR

Contains the color selection as 24-bit integer value.

5.2 Prima::FindDialog, Prima::ReplaceDialog

Standard interface dialogs to find and replace options selection.

Synopsis

```
use Prima::StdDlg;

my $dlg = Prima::FindDialog-> create( findStyle => 0);
my $res = $dlg-> execute;
if ( $res == mb::Ok) {
        print $dlg-> findText, " is to be found\n";
} elsif ( $res == mb::ChangeAll) {
        print "all occurences of ", $dlg-> findText,
                " is to be replaced by ", $dlg-> replaceText;
}
```

The `mb::ChangeAll` constant, one of possible results of `execute` method, is defined in the *Prima::StdDlg* section module. Therefore it is recommended to include this module instead.

Description

The module provides two classes - Prima::FindDialog and Prima::ReplaceDialog; Prima::ReplaceDialog is exactly same as Prima::FindDialog except that its default the *findStyle* entry property value is set to 0. One can use a dialog-caching technique, arbitrating between the *findStyle* entry value 0 and 1, and use only one instance of Prima::FindDialog.

The module does not provide the actual search algorithm; this must be implemented by the programmer. The toolkit currently include some facilitation to the problem - the part of algorithm for `Prima::Edit` class is found in the **find** entry in the *Prima::Edit* section, and the another part - in *examples/editor.pl* example program. the *Prima::HelpWindow* section also uses the module, and realizes its own searching algorithm.

API

Properties

All the properties select the user-assigned values, except the *findStyle* entry.

findText STRING

Selects the text string to be found.

Default value: ”

findStyle BOOLEAN

If 1, the dialog provides only 'find text' interface. If 0, the dialog provides also 'replace text' interface.

Default value: 1 for `Prima::FindDialog`, 0 for `Prima::ReplaceDialog`.

options INTEGER

Combination of `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 that is to replace the found text.

Default value: ”

scope

One of `fds::` constants. Represents the scope of the search: it can be started from the cursor position, of from the top or of the bottom of the text.

```
fds::Cursor
fds::Top
fds::Bottom
```

Default value: `fds::Cursor`

5.3 Prima::FileDialog

File system related widgets and dialogs.

Synopsis

open a file use Prima qw(Application); use Prima::StdDlg;

```
my $open = Prima::OpenDialog-> new(
        filter => [
                ['Perl modules' => '*.pm'],
                ['All' => '*']
        ]
);
print $open-> fileName, " is to be opened\n" if $open-> execute;

# save a file
my $save = Prima::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 widgets for file and drive selection, and also standard open file, save file, and change directory dialogs.

Prima::DirectoryListBox

A directory listing list box. Shows the list of subdirectories and upper directories, hierarchy-mapped, with the folder images and outlines.

Properties

closedGlyphs INTEGER

Number of horizontal equal-width images, contained in the *closedIcon* entry property.

Default value: 1

closedIcon ICON

Provides an icon representation for the directories, contained in the current directory.

indent INTEGER

A positive integer number of pixels, used for offset of the hierarchy outline.

Default value: 12

openedGlyphs INTEGER

Number of horizontal equal-width images, contained in the *openedIcon* entry property.

Default value: 1

openedIcon OBJECT

Provides an icon representation for the directories, contained in the directories above the current directory.

path STRING

Runtime-only property. Selects a file system path.

showDotDirs BOOLEAN

Selects if the directories with the first dot character are shown the view. The treatment of the dot-prefixed names as hidden is traditional to unix, and is of doubtful use under win32 and os2.

Default value: 1

Methods

files [FILE_TYPE]

If 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.

Prima::DriveComboBox

Provides drive selection combo-box for non-unix systems.

Properties

firstDrive DRIVE_LETTER

Create-only property.

Default value: 'A:'

DRIVE_LETTER can be set to other value to start the drive enumeration from. Some OSes can probe eventual diskette drives inside the drive enumeration routines, so it might be reasonable to set DRIVE_LETTER to `C:` string for responsiveness increase.

drive DRIVE_LETTER

Selects the drive letter.

Default value: 'C:'

Prima::FileDialog

Provides a standard file dialog, allowing to navigate by the file system and select one or many files. The class can operate in two modes - 'open' and 'save'; these modes are set by the *Prima::OpenDialog* section and the *Prima::SaveDialog* section. Some properties behave differently depending on the mode, which is stored in the *openMode* entry property.

Properties

createPrompt BOOLEAN

If 1, and a file selected 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 single-file selection, assigns the selected file name, For multiple-file selection, on get-call returns list of the selected files; on set-call, accepts a single string, where the file names are separated by the space character. The eventual space characters must be quoted.

filter ARRAY

Contains 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 *filter* entry array of 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 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 overwrittten.

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, types by the user, exists before closing the dialog.

Default value: 1

showDotFiles BOOLEAN

Selects if the directories with the first dot character are shown the files view.

Default value: 0

showHelp BOOLEAN

Create-only property. If 1, 'Help' button is inserted in the dialog.

Default value: 1

sorted BOOLEAN

Selects whether the file list appears sorted by name (1) or not (0).

Default value : 1

system BOOLEAN

Create-only property. If set to 1, `Prima::FileDialog` returns instance of `Prima::sys::XXX::FileDialog` system-specific file dialog, if available for the *XXX* platform.

`system` knows only how to map `FileDialog`, `OpenDialog`, and `SaveDialog` classes onto the system-specific file dialog classes; the inherited classes are not affected.

Methods

reread

Re-reads the currently selected directory.

Prima::OpenDialog

Descendant of the *Prima::FileDialog* section, tuned for open-dialog functionality.

Prima::SaveDialog

Descendant of the *Prima::FileDialog* section, tuned for save-dialog functionality.

Prima::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 the view.

Default value: 0

showHelp

Create-only property. If 1, 'Help' button is inserted in the dialog.

Default value: 1

5.4 Prima::FontDialog

Standard font dialog

Synopsis

```
use Prima::FontDialog;
my $f = Prima::FontDialog-> create;
return unless $f-> execute == mb::OK;
$f = $f-> logFont;
print "$_:$f->{$_}\n" for sort keys %$f;
```

Description

The dialog provides selection of font by name, style, size, and encoding. The font selected is returned by 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`.

showHelp BOOLEAN

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 left mouse button.

Default action reflects the status in the dialog title

EndDragFont $WIDGET

Called when the user releases the mouse drag over a Prima widget. Default action applies currently selected font to $WIDGET.

5.5 Prima::ImageDialog

File open and save dialogs.

Description

The module provides dialogs specially adjusted for image loading and saving.

Prima::ImageOpenDialog

Provides a preview feature, 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 `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 by default contains {`extras`} hash variable set, which contains extra information returned by the loader. See the *Prima::image-load* section for more information.

Synopsis

```
my $dlg = Prima::ImageOpenDialog-> create;
my $img = $dlg-> load;
return unless $img;
print "$_:$img->{extras}->{$_}\n" for sort keys %{$img-> {extras}};
```

Proprties

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 `Prima::Image::load` method. In particular, it can be used to disable the default loading of extra information in {`extras`} 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 progressive 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 profile key `progressViewer` is supplied, its value is treated as a `Prima::ImageViewer` instance, and it is used to display image 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::ImageSaveDialog

Provides a save dialog where the user can select 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. The standard toolkit dialogs reside under in `Prima::Image` namespace, in *Prima/Image* sub-directory. For example, `Prima::Image::gif` provides the selection of transparency color, and `Prima::Image::jpeg` the image quality control. If the image passed to the the *image* entry property contains {`extras`} variable, the data are read and used as the default values. In particular, {`extras`}->-{`codecID`} field, responsible for the file format, if present, affects the default file format selection.

Synopsis

```
my $dlg = Prima::ImageSaveDialog-> create;
return unless $dlg-> save( $image );
print "saved as ", $dlg-> fileName, "\n";
```

Properties

image IMAGE

Selects the image to be saved. This property is to be used for the standard invocation of dialog, via `execute`. It is not needed when the execution and saving is invoked via the *save* entry method.

Methods

save IMAGE, %PROFILE

Invokes the dialog, and, if the execution was successful, saves the IMAGE according to the user selection and PROFILE hash. PROFILE is not used for the default options, but is passed directly to `Prima::Image::save` call, possibly overriding selection of the user. Returns 1 in case of success, 0 in case of error. If the error occurs, the user is notified before the method returns.

5.6 Prima::MsgBox

Standard message and input dialog boxes

Description

The module contains two methods, `message_box` and `input_box`, that invoke correspondingly the standard message and one line text input dialog boxes.

Synopsis

```
use Prima;
use Prima::Application;
use Prima::MsgBox;

my $text = Prima::MsgBox::input_box( 'Sample input box', 'Enter text:', '');
$text = '(none)' unless defined $text;
Prima::MsgBox::message( "You have entered: '$text'", mb::Ok);
```

API

input_box TITLE, LABEL, INPUT_STRING, [BUTTONS = mb::OkCancel, PROFILES = {}]

Invokes standard dialog box, that contains an input line, a text label, and buttons used for ending dialog session. The dialog box uses TITLE string to display as the window title, 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 BUTTONS integer parameter, which can be a combination of the button `mb::XXX` constants, different combinations of push buttons can be displayed in the dialog.

PROFILE parameter is a hash, that contains customization parameters for the buttons and the input line. To access input line `inputLine` hash key is used. See the *Buttons and profiles* entry for more information on BUTTONS and PROFILES.

Returns different results depending on the caller context. In array context, returns two values: the result of `Prima::Dialog::execute`, which is either `mb::Cancel` or one of `mb::XXX` constants of the dialog buttons; and the text entered. The input text is not restored to its original value if the dialog was cancelled. In scalar context returns the text entered, if the dialog was ended with `mb::OK` or `mb::Yes` result, or `undef` otherwise.

message TEXT, [OPTIONS = mb::Ok | mb::Error, PROFILES = {}]

Same as `message_box` call, with application name passed as the title string.

message_box TITLE, TEXT, [OPTIONS = mb::Ok | mb::Error, PROFILES = {}]

Invokes standard dialog box, that contains a text label, a predefined icon, and buttons used for ending dialog session. The dialog box uses TITLE string to display as the window title, TEXT to display as a main message. Value of OPTIONS integer parameter is combined from two different sets of `mb::XXX` constants. The first set is the buttons constants, - `mb::OK`, `mb::Yes` etc. See the *Buttons and profiles* entry for the explanations. The second set consists of the following message type 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 if no sound is desired, a special constant `mb::NoSound` can be used.

PROFILE parameter is a hash, that contains customization parameters for the buttons. See the *Buttons and profiles* entry for the explanations.

Returns the result of `Prima::Dialog::execute`, which is either `mb::Cancel` or one of `mb::XXX` constants of the specified dialog buttons.

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, PROFILES hash parameter can be used. In this hash, predefined keys can be used to 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 equal to the `mb::` constant of the desired button. If this option is not set, the leftmost button is selected as the default.

helpTopic TOPIC

Used to select the help TOPIC, invoked in the help viewer window if `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 a 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 `buttons` key. The hash is a 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.

5.7 Prima::PrintDialog

Standard printer setup dialog

Description

Provides a standard dialog that allows the user to select a printer and its options. The toolkit does not provide the in-depth management of the printer options; this can only be accessed by executing a printer-specific setup window, called by `Prima::Printer::setup_dialog`. The class invokes this method when the user presses 'Properties' button. Otherwise, the class provides only selection of a printer from the printer list.

When the dialog finished successfully, the selected printer is set as the current by writing to `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::PrintDialog;

$dlg = Prima::PrintSetupDialog-> create;
if ( $dlg-> execute) {
        my $p = $::application-> get_printer;
        if ( $p-> begin_doc ) {
                $p-> text_out( 'Hello world', 10, 10);
                $p-> end_doc;
        }
}
$dlg-> destroy;
```

5.8 Prima::StdDlg

Wrapper module to the toolkit standard dialogs

Description

Provides a unified access to the toolkit dialogs, so there is no need to `use` the corresponding module explicitly.

Synopsis

```
use Prima::StdDlg;

Prima::FileDialog-> create-> execute;
Prima::FontDialog-> create-> execute;

# open standard file open dialog
my $file = Prima::open_file;
print "You've selected: $file\n" if defined $file;
```

API

The module accesses the following dialog classes:

Prima::open_file

Invokes standard file open dialog and return the selected file(s). Uses system-specific standard file open dialog, if available.

Prima::save_file

Invokes standard file save dialog and return the selected file(s). Uses system-specific standard file save dialog, if available.

Prima::OpenDialog

File open dialog.

See the **Prima::OpenDialog** entry in the *Prima::FileDialog* section

Prima::SaveDialog

File save dialog.

See the **Prima::SaveDialog** entry in the *Prima::FileDialog* section

Prima::ChDirDialog

Directory change dialog.

See the **Prima::ChDirDialog** entry in the *Prima::FileDialog* section

Prima::FontDialog

Font selection dialog.

See the *Prima::FontDialog* section.

Prima::FindDialog

Generic 'find text' dialog.

See the *Prima::EditDialog* section.

Prima::ReplaceDialog

Generic 'find and replace text' dialog.

See the *Prima::EditDialog* section.

Prima::PrintSetupDialog

Printer selection and setup dialog.

See the *Prima::PrintDialog* section.

Prima::ColorDialog

Color selection dialog.

See the **Prima::ColorDialog** entry in the *Prima::ColorDialog* section.

Prima::ImageOpenDialog

Image file load dialog.

See the **Prima::ImageOpenDialog** entry in the *Prima::ImageDialog* section.

Prima::ImageSaveDialog

Image file save dialog.

See the **Prima::ImageSaveDialog** entry in the *Prima::ImageDialog* section.

6 Visual Builder

6.1 VB

Visual Builder for the Prima toolkit

Description

Visual Builder is a RAD-style suite for designing forms under the Prima toolkit. It provides rich set of perl-composed widgets, whose can be inserted into a form by simple actions. The form can be stored in a file and loaded by either user program or a simple wrapper, `utils/fmview.pl`; the form can be also stored as a valid perl program.

A form file typically has *.fm* extension, an can be loaded by using the *Prima::VB::VBLoader* section module. The following example is the only content of `fmview.pl`:

```
use Prima qw(Application VB::VBLoader);
my $ret = Prima::VBLoad( $ARGV[0] );
die "$@\n" unless $ret;
$ret-> execute;
```

and is usually sufficient for executing a form file.

Help

The builder provides three main windows, that are used for designing. 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 designated 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 children of that widget. After the widget is inserted, its properties are accessible via 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 have to be saved.

This command is an alias to a 'new file' icon on the panel.

Open

Invokes a file open dialog, so a *.fm* form file can be opened. After the successful file load, all form widgets are visible and available for editing.

This command is an alias to an '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 *.fm* form file, then it can be re-loaded either in the builder or in the user program (see the *Prima::VB::VBLoader* section for details). If the form is saved as *.pl* program, then it can not be loaded; instead, the program can be run immediately without the builder or any supplementary code.

Once the user assigned a name and a type for the form, it is never asked when selecting this command.

This command is an alias to a 'save on disk' icon on the panel.

Save as

Same as the *Save* entry, except that a new name or type of file are 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 have to be saved.

Edit

Copy

Copies the selected widgets into the clipboard, so they can be inserted later by using the *Paste* entry command. The form window can not be copied.

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 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 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 an inserted widget. This is an advanced option, and can lead to confusions 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 a loaded form does not contain a valid widget class; in such case *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, `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 a widget from being selected by mouse, to avoid occasional positional changes. This is useful when a widget is used as owner for many sub-widgets.

Ctrl+mouse click locks and unlocks a widget.

View

Object inspector

Brings the object inspector window, if it was hidden or closed.

Add widgets

Opens a file dialog, where the additional VB modules can be located. 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

Reset the guidelines on the form window into a center position.

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 `fmview.pl`. The execution session ends either by closing the form window or by calling the *Break* entry command.

This command is an alias to a 'run' icon on the panel.

Break

Explicitly terminates the execution session, initiated by 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 a help viewer on the *Prima::Widget* section manpage and tries to open a topic, corresponding to the current selection of the object inspector property or event list. While this 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 a common parent for all widgets, created by the builder. The form window provides the following basic navigation functionality.

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 menu option the *Snap to guidelines* entry is on, the widgets moving and sizing operations treat the guidelines as the 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 *Select all* entry command from the menu. To prevent widgets from being occasionally selected, lock them with "Edit/Toggle lock" command or Ctrl+mouse click.

Moving

Dragging the mouse can move the selected widgets. The 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; other two are the horizontal only and the vertical only.

Sizing

The sizeable widgets can be dynamically resized. Regardless to the amount of the selected widgets, 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 `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 the properties or the events are selected, the left panel of the inspector provides the properties or events list, 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` (the detailed explanation see in the *Prima::Object* section). Among these are such basic properties as `origin`, `size`, `name`, `color`, `font`, `visible`, `enabled`, `owner` and many others. All the widgets share some common denominator, but almost all provide their own intrinsic properties. Each property can be selected by the right-pane hosted 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 the builder keeps a name-keyed list; another consequence of this fact is that no widgets of same name can exist simultaneously within 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 `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 extracted by native toolkit functionality. To read on 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 ground `Prima::Widget` functionality. The other widgets are (hopefully) documented in their modules, for example, `Prima::ScrollBar` documentation can be found in the *Prima::ScrollBar* section.

6.2 Prima::VB::VBLoader

Visual Builder file loader

Description

The module provides functionality for loading resource files, created by Visual Builder. After the successful load, the newly created window with all children is returned.

Synopsis

The simple way to use the loader is as that:

```
use Prima qw(Application);
use Prima::VB::VBLoader;
Prima::VBLoad( './your_resource.fm',
        Form1 => { centered => 1 },
)-> execute;
```

A more complicated but more proof code can be met in the toolkit:

```
use Prima qw(Application);
eval "use Prima::VB::VBLoader"; die "$@\n" if $@;
$form = Prima::VBLoad( $fi,
        'Form1'     => { visible => 0, centered => 1},
);
die "$@\n" unless $form;
```

All form widgets can be supplied with custom parameters, all together combined in a hash of hashes and passed as the second parameter to `VBLoad()` function. The example above supplies values for `::visible` and `::centered` to `Form1` widget, which is default name of 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 a fm file but from other data source, the *AUTOFORM_REALIZE* entry call can be used to transform perl array into set of widgets:

```
$form = AUTOFORM_REALIZE( [ Form1 => {
        class   => 'Prima::Window',
        parent  => 1,
        profile => {
                name => 'Form1',
                size => [ 330, 421],
        }], {});
```

Real-life examples are met across the toolkit; for instance, *Prima/PS/setup.fm* dialog is used by `Prima::PS::Setup`.

API

Methods

check_version HEADER

Scans HEADER, - the first line of a .fm file for 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 a version string.

GO_SUB SUB [@EXTRA_DATA]

Depending on value of 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 `sub{}` context and returns the code reference. If evaluation fails, EXTRA_DATA is stored in `Prima::VB::VBLoader::eventContext` array and the exception is re-thrown. `Prima::VB::VBLoader::builderActive` is an internal flag that helps the Visual Builder use the module interface without actual SUB evaluation.

AUTOFORM_REALIZE WIDGETS, PARAMETERS

WIDGETS is an array reference that contains evaluated data of the read content of .fm file (its data format is preserved). PARAMETERS is a hash reference with custom parameters passed to widgets during creation. The widgets are distinguished by the names. Visual Builder ensures that no widgets have equal names.

`AUTOFORM_REALIZE` creates the tree of widgets and returns the root window, which is usually named `Form1`. It automatically resolves parent-child relations, so the order of WIDGETS does not matter. Moreover, if a parent widget is passed as a parameter to a children widget, the parameter is deferred and passed after the creation using `::set` call.

During the parsing and creation process internal notifications can be invoked. These notifications (events) are stored in .fm file and usually provide class-specific loading instructions. See the *Events* entry for details.

AUTOFORM_CREATE FILENAME, %PARAMETERS

Reads FILENAME in .fm file format, checks its version, loads, and creates widget tree. Upon successful load the root widget is returned. The parsing and creation is performed by calling `AUTOFORM_REALIZE`. If loading fails, `die()` is called.

Prima::VBLoad FILENAME, %PARAMETERS

A wrapper around `AUTOFORM_CREATE`, exported in `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 < symbol (the syntax is influenced by `CORE::open`).

%PARAMETERS is a hash with custom parameters passed to widgets during creation. The widgets are distinguished by the 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 `$@` variable.

Events

The events, stored in .fm file are called during the loading process. The module provides no functionality for supplying the events during the load. This interface is useful only for developers of Visual Builder - ready classes.

The events section is located in `actions` section of widget entry. There can be more than one event of each type, registered to different widgets. NAME parameter is a string with name of the widget; INSTANCE is a hash, created during load for every widget provided to keep internal event-specific or class-specific data there. `extras` section of widget entry is present there as an only predefined key.

Begin NAME, INSTANCE

Called upon beginning of widget tree creation.

FormCreate NAME, INSTANCE, ROOT_WIDGET

Called after the creation of a form, which reference is contained in ROOT_WIDGET.

Create NAME, INSTANCE, WIDGET.

Called after the creation of the widget. The newly created widget is passed in WIDGET

Child NAME, INSTANCE, WIDGET, CHILD_NAME

Called before child of WIDGET is created with CHILD_NAME as name.

ChildCreate NAME, INSTANCE, WIDGET, CHILD_WIDGET.

Called after 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 format of .fm file is that is should be evaluated by perl `eval()` call without special manipulations, and kept as plain text. The file begins with a header, which is a #-prefixed string, and contains a signature, version of file format, and 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 `sub{}` statement. After evaluation, this sub returns array of paired scalars, where each first item is a widget name and second item is 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 hash of events. The events are evaluated via `GO_SUB` mechanism and executed during creation of the widget tree. See the *Events* entry for details.

code STRING

Contains a code, executed before the form is created. This key is present only on the root widget record.

class STRING

Contains name of a class to be instantiated.

extras HASH

Contains a class-specific parameters, used by events.

module STRING

Contains name of perl module that contains the class. The module will be `use`'d by the loader.

parent BOOLEAN

A boolean flag; set to 1 for the root widget only.

profile HASH

Contains profile hash, passed as parameters to the widget during its creation. If custom parameters were passed to `AUTOFORM_CREATE`, these are coupled with `profile` (the custom parameters take precedence) before passing to the `create()` call.

6.3 Prima::VB::CfgMaint

Maintains visual builder widget palette configuration.

Description

The module is used by the Visual Builder and `cfgmaint` programs, to maintain the Visual Builder widget palette. The installed widgets are displayed in main panel of the Visual Builder, and can be maintained by `cfgmaint`.

Usage

The Visual Builder widget palette configuration is contained in two files - the system-wide `Prima::VB::Config` and the user `~/.prima/vbconfig`. The user config file take the precedence when loaded by the Visual Builder. The module can select either configuration by assigning `$systemWide` boolean property.

The widgets are grouped in pages, which are accessible by names.

New widgets can be added to the palette by calling `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 format of a module with VB-loadable widgets.

The module must define a package with same name as the module. In the package, `class` sub must be declared, that returns an array or paired scalars, where each first item in a pair corresponds to the widget class and the second to a hash, that contains the class loading information, and must contain the following keys:

class STRING

Name of the VB-representation class, which represents the original widget class in the Visual Builder. This is usually a lightweight class, which does not contain all functionality of the original class, but is capable of visually reflecting changes to the class properties.

icon PATH

Sets an image file, where the class icon is contained. 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 `:` 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 put. The current implementation of the Visual Builder provides four pages: `General,Additional,Sliders,Abstract`. If the page is not present, new page is automatically created when the widget class is registered.

RTModule STRING

Sets the module name, that contains the original class.

The reader is urged to explore *Prima::VB::examples::Widgety* file, which contains an example class `Prima::SampleWidget`, its VB-representation, and a property `lineRoundStyle` definition example.

API

Methods

add_module FILE

Reads FILE module and loads all VB-loadable widgets from it.

classes

Returns string declaration of all registered classes in format of `classes` registration procedure (see the *Format* entry).

open_cfg

Loads class and pages information from either a system-wide or a user configuration file. If succeeds, the information is stored in `@pages` and `%classes` variables (the old information is lost) and returns 1. If fails, returns 0 and string with the error explanation; `@pages` and `%classes` content is undefined.

pages

Returns array of page names

read_cfg

Reads information from both system-wide and user configuration files, and merges the information. If succeeds, returns 1. If fails, returns 0 and 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 `$backup` flag is set to 1, the old file renamed with `.bak` extension. If succeeds, returns 1. If fails, returns 0 and string with the error explanation.

Files

Prima::VB::Config.pm, `~/.prima/vbconfig`.

6.4 Prima::VB::CfgMaint

Maintains visual builder widget palette configuration.

Description

The module is used by the Visual Builder and `cfgmaint` programs, to maintain the Visual Builder widget palette. The installed widgets are displayed in main panel of the Visual Builder, and can be maintained by `cfgmaint`.

Usage

The Visual Builder widget palette configuration is contained in two files - the system-wide `Prima::VB::Config` and the user `~/.prima/vbconfig`. The user config file take the precedence when loaded by the Visual Builder. The module can select either configuration by assigning `$systemWide` boolean property.

The widgets are grouped in pages, which are accessible by names.

New widgets can be added to the palette by calling `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 format of a module with VB-loadable widgets.

The module must define a package with same name as the module. In the package, `class` sub must be declared, that returns an array or paired scalars, where each first item in a pair corresponds to the widget class and the second to a hash, that contains the class loading information, and must contain the following keys:

class STRING

Name of the VB-representation class, which represents the original widget class in the Visual Builder. This is usually a lightweight class, which does not contain all functionality of the original class, but is capable of visually reflecting changes to the class properties.

icon PATH

Sets an image file, where the class icon is contained. 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 `:` 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 put. The current implementation of the Visual Builder provides four pages: `General,Additional,Sliders,Abstract`. If the page is not present, new page is automatically created when the widget class is registered.

RTModule STRING

Sets the module name, that contains the original class.

The reader is urged to explore *Prima::VB::examples::Widgety* file, which contains an example class `Prima::SampleWidget`, its VB-representation, and a property `lineRoundStyle` definition example.

API

Methods

add_module FILE

Reads FILE module and loads all VB-loadable widgets from it.

classes

Returns string declaration of all registered classes in format of `classes` registration procedure (see the *Format* entry).

open_cfg

Loads class and pages information from either a system-wide or a user configuration file. If succeeds, the information is stored in `@pages` and `%classes` variables (the old information is lost) and returns 1. If fails, returns 0 and string with the error explanation; `@pages` and `%classes` content is undefined.

pages

Returns array of page names

read_cfg

Reads information from both system-wide and user configuration files, and merges the information. If succeeds, returns 1. If fails, returns 0 and 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 `$backup` flag is set to 1, the old file renamed with `.bak` extension. If succeeds, returns 1. If fails, returns 0 and string with the error explanation.

Files

Prima::VB::Config.pm, `~/.prima/vbconfig`.

7 PostScript printer interface

7.1 Prima::PS::Drawable

PostScript interface to Prima::Drawable

Synopsis

```
use Prima;
use Prima::PS::Drawable;

my $x = Prima::PS::Drawable-> create( onSpool => sub {
        open F, ">> ./test.ps";
        print F $_[1];
        close F;
});
$x-> begin_doc;
$x-> font-> size( 30);
$x-> text_out( "hello!", 100, 100);
$x-> end_doc;
```

Description

Realizes the Prima library interface to PostScript level 2 document language. The module is designed to be compliant with Prima::Drawable interface. All properties' behavior is as same as Prima::Drawable's, except those described below.

Inherited properties

::resolution

Can be set while object is in normal stage - cannot be changed if document is opened. Applies to fillPattern realization and general pixel-to-point and vice versa calculations

::region

- ::region is not realized (yet?)

Specific properties

::copies

amount of copies that PS interpreter should print

::grayscale

could be 0 or 1

::pageSize

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

::rotate and ::scale

along with Prima::Drawable::translate provide PS-specific transformation matrix manipulations. ::rotate is number, measured in degrees, counter-clockwise. ::scale is array of two numbers, respectively x- and y-scale. 1 is 100%, 2 is 200% etc.

::useDeviceFonts

1 by default; optimizes greatly text operations, but takes the risk that a character could be drawn incorrectly or not drawn at all - this behavior depends on a particular PS interpreter.

::useDeviceFontsOnly

If 1, the system fonts, available from Prima::Application interfaces can not be used. It is designed for developers and the outside-of-Prima applications that wish to use PS generation module without graphics. If 1, `::useDeviceFonts` is set to 1 automatically.

Default value is 0

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 pixel to points and vice versa.

fill & stroke

Wrappers for PS outline that is expected to be filled or stroked. Apply colors, line and fill styles if necessary.

spool

Prima::PS::Drawable is not responsible for output of generated document, it just calls ::spool when document is closed through ::end_doc. By default just skips data. Prima::PS::Printer handles spooling logic.

fonts

Returns Prima::Application::font plus those that defined into Prima::PS::Fonts module.

7.2 Prima::PS::Encodings

Manage latin-based encodings

Synopsis

```
use Prima::PS::Encodings;
```

Description

This module provides code tables for major latin-based encodings, for the glyphs that usually provided by every PS-based printer or interpreter. Prima::PS::Drawable uses these encodings when it decides whether the document have to be supplied with a bitmap character glyph or a character index, thus relying on PS interpreter capabilities. Latter is obviously preferable, but as it's not possible to know beforehand what glyphs are supported by PS interpreter, the Latin glyph set was selected as a ground level.

files

It's unlikely that users will need to supply their own encodings, however this can be accomplished by:

```
use Prima::PS::Encodings;
$Prima::PS::Encodings::files{iso8859-5} = 'PS/locale/greek-iso';
```

fontspecific

The only non-latin encoding currently present is 'Specific'. If any other specific-encoded fonts are to be added, the encoding string must be added as a key to %fontspecific

load

Loads encoding file by given string. Tries to be smart to guess actual file from identifier string returned from setlocale(NULL). If fails, loads default encoding, which defines only glyphs from 32 to 126. Special case is 'null' encoding, returns array of 256 .notdef's.

unique

Returns list of Latin-based encoding string unique keys.

7.3 Prima::PS::Fonts

PostScript device fonts metrics

Synopsis

```
use Prima;
use Prima::PS::Fonts;
```

Description

This module primary use is to be invoked from Prima::PS::Drawable module. Assumed that some common fonts like Times and Courier are supported by PS interpreter, and it is assumed that typeface is preserved more-less the same, so typesetting based on font's a-b-c metrics can be valid. 35 font files are supplied with 11 font families. Font files with metrics located into 'fonts' subdirectory.

query_metrics($fontName)

Returns font metric hash with requested font data, uses $defaultFontName if give name is not found. Metric hash is the same as Prima::Types::Font record, plus 3 extra fields: 'docname' containing font name (equals always to 'name'), 'chardata' - hash of named glyphs, 'charheight' - the height that 'chardata' is rendered to. Every hash entry in 'chardata' record contains four numbers - suggested character index and a, b and c glyph dimensions with height equals 'charheight'.

enum_fonts($fontFamily)

Returns font records for given family, or all families perpesented by one member, if no family name given. If encoding specified, returns only the fonts with the encoding given. Compliant to Prima::Application::fonts interface.

files & enum_families

Hash with paths to font metric files. File names not necessarily should be as font names, and it is possible to override font name contained in the file just by specifying different font key - this case will be recognized on loading stage and loaded font structure patched correspondingly.

Example:

```
$Prima::PS::Fonts::files{Standard Symbols} = $Prima::PS::Fonts::files{Symbol};

$Prima::PS::Fonts::files{'Device-specific symbols, set 1'} = 'my/devspec/data.1';
$Prima::PS::Fonts::files{'Device-specific symbols, set 2'} = 'my/devspec/data.2';
$Prima::PS::Fonts::enum_families{DevSpec} = 'Device-specific symbols, set 1';
```

font_pick($src, $dest, %options)

Merges two font records using Prima::Drawable::font_match, picks the result and returns new record. $variablePitchName and $fixedPitchName used on this stage.

Options can include the following fields:

- resolution - vertical resolution. The default value is taken from font resolution.

enum_family($fontFamily)

Returns font names that are presented in given family

7.4 Prima::PS::Printer

PostScript interface to Prima::Printer

Synopsis

use Prima; use Prima::PS::Printer;

Description

Realizes the Prima printer interface to PostScript level 2 document language through Prima::PS::Drawable module. Allows different user profiles to be created and managed with GUI setup dialog. The module is designed to be compliant with Prima::Printer interface.

Also contains convenience classes (File, LPR, Pipe) for non-GUI use.

Synopsis

```
use Prima::PS::Printer;

my $x;
if ( $preview) {
        $x = Prima::PS::Pipe-> create( command => 'gv -');
} elsif ( $print_in_file) {
        $x = Prima::PS::File-> create( file => 'out.ps');
} else {
        $x = Prima::PS::LPR-> create( args => '-Pcolorprinter');
}
$x-> begin_doc;
$x-> font-> size( 300);
$x-> text_out( "hello!", 100, 100);
$x-> end_doc;
```

Printer options

Below is the list of options supported by `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

Scaling FLOAT

1 is 100%, 1.5 is 150%, etc.

Orientation

One of : `Portrait`, `Landscape`.

UseDeviceFonts BOOLEAN

If 1, use limited set of device fonts in addition to exported bitmap fonts.

UseDeviceFontsOnly BOOLEAN

If 1, use limited set of device fonts instead of exported bitmap fonts. Its usage may lead to that some document fonts will be mismatched.

MediaType STRING

An arbitrary string representing 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.

MediaColor STRING

A string identifying the color of the medium.

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.

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.

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.

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.

ManualFeed BOOLEAN

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.

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.

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.

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.

MatchAll BOOLEAN

A flag specifying whether input media request should match to all non-null values - MediaColor, MediaWeight etc.

8 C interface to the toolkit

8.1 Prima::internals

Prima internal architecture

Description

This documents elucidates the internal structures of the Prima toolkit, its loading considerations, object and class representation and C coding style.

Bootstrap

Initializing

For a perl script, Prima is no more but an average module that uses DynaLoader. As 'use Prima' code gets executed, a 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 Prima module is ready to use. All wrapping code for built-in functionality that can be seen from perl is located into two modules - Prima::Const and Prima::Classes.

Constants

Prima defines lot of constants for different purposes (e.g. colors, font styles etc). Prima does not follow perl naming conventions here, on the reason 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 used first), Prima registers these functions during boot_Prima stage. As soon as perl code tries to get a constant's value, an AUTOLOAD function is called, which is binded inside Prima::Const. Constants are widely used both in C and perl code, and are defined in apricot.h in that way so perl constant definition comes along with C one. 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 of UV's (unsigned integers) and a register_fe_constants() function, which should be called at boot_Prima stage. This way feRead becomes C analog to fe::Read in perl.

Classes and methods

Virtual method tables

Prima implementation of classes uses virtual method tables, or VMTs, in order to make the classes inheritable and their methods overrideable. The VMTs are usual C structs, that contain pointers to functions. 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.

Built-in classes already contain all information needed for method overloading, but when a new class is derived from existing one, new VMT is have to be created as well. The actual sub-classing is performed inside build_dynamic_vmt() and build_static_vmt(). gimme_the_vmt() function creates new VMT instance on the fly and caches the result for every new class that is derived from Prima class.

C to Perl and Perl to C calling routines

Majority of Prima methods is 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.

NB - Do not mix XS calls to xs language (*perlxs* manpage) - the latter is a meta-language for simplification of coding tasks and is not used in 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 gencls tool (the *gencls* section manpage), and typical Prima method consists of three functions, two of which are thunks.

First function, say Class_init(char*), would initialize a class (for example). It is written fully in C, so in order to be called from perl code a registration step must be taken for a second function, Class_init_FROMPERL(), that would look like

```
newXS( "Prima::Class::init", Class_init_FROMPERL, "Prima::Class");
```

Class_init_FROMPERL() is a first thunk, that translates the parameters passed from perl to C and the result back from C function to perl. This step is almost fully automatized, so one never bothers about writing 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 perl code. 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 return values from a perl function back to C. This thunk is also generated automatically by 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 into core C files. In the class declaration files these methods are prepended with 'public' or 'weird' modifiers, when methods with no special data handling needs use 'method' or 'static' modifiers.

NB - functions that are not allowed to be seen from perl have 'c_only' modifier, and therefore do not need 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
            Timer
            Widget
                    Application
                    Window
```

These classes can be seen from perl with Prima:: prefix. Along with these, Utils class is defined. 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 marked with 'package' prefix, when others are marked with 'object' (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 Prima::Object descendant object. Prima employs blessed hashes for 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 it is not enough to create Prima objects. Every Prima::Object descendant class therefore is equipped with create() method, that allocates object instance and calls bless() itself. Parameters that come with create() call are formed into a hash and passed to init() method, that is also present on every object. Note the fact that although perl-coded init() returns the hash, it 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 perl stack, using parse_hv() and push_hv() functions.

Objects can be created from perl by using 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);
```

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 create() method, every object class has destroy() method. 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 memory leak problem.

For user code it is sufficient to overload done() and/or cleanup() methods, or just onDestroy notifications. It is highly recommended to avoid overloading destroy method, since it can be called in 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 object data after their creation represent an object instance. All Prima objects are blessed hashes, and the hash key __CMATE__ holds a C pointer to a memory which is occupied by C data instance, or a "mate". It keeps all object variables and a pointer to VMT. Every object has its own copy of data instance, but the VMTs can be shared. In order to reach to C data instance gimme_the_mate() function is used. As a first parameter 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

It was decided to divide object life stage in several steps. Every stage is mirrored into PObject(self)-> stage integer variable, which can be one of csXXX constants. Currently it has six:

csConstructing

Initial stage, is set until create() is finished. Right after init() is completed, setup() method is called.

csNormal

After create() is finished and before destroy() started. If an object is csNormal and csConstructing stage, Object_alive() result would be non-zero.

csDestroying

destroy() started. This stage includes calling of cleanup() and done() routines.

csFrozen

cleanup() started.

csFinalizing

done() started

csDead

Destroy finished

Coding techniques

Accessing object data

C coding has no specific conventions, except when a code is an object method. Object syntax for accessing object instance data is also fairly standard. For example, accessing component's 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 code could to be called also 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
```

This calling is preferred, comparing to direct call of Component_get_name(), primarily because get_name() is a method and can be overridden from user code.

Calling perl code

call_perl_indirect() function accepts object, its method name and parameters list with parameter format string. It has several wrappers for easier 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 format string represents a parameters type, and characters can be:

```
'i' - integer
's' - char *
'n' - float
'H' - Handle
'S' - SV *
'P' - Point
'R' - Rect
```

The format string can be prepended with '<' character, in which case SV * scalar (always scalar, even if code returns nothing or array) value is returned. The caller is responsible for freeing the return value.

Exceptions

As descriped in *perlguts* manpage, 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

```
SvTRUE( GvSV( errgv))
```

statement. It is guaranteed to be false if there was no exception condition. But in some situations, namely, when no perl_call_* functions are called or error value is already assigned before calling code, there is a wrapping technique that keeps previous error message and looks like:

```
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 workaround to a "false alarm" situation, if SvTRUE(GvSV(errgv)) is true before perl_call_method().

Object protection

After the object destroy stage is completed, it is possible that object's data instance is gone, and even simple stage check might cause segmentation fault. To avoid this, bracketing functions called `protect_object()` and `unprotect_object()` are used. protect_object() increments reference count to the object instance, thus delaying its freeing until decrementing unprotect_object() is called.

All C code that references to an object must check for its stage after every routine that switches to perl code, because the object might be destroyed inside the call. Typical code example would be like:

```
function( 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 == csDead) return;

     // proceed with the object
}
```

Usual C code never checks for object stage before the call, because gimme_the_mate() function returns NULL if object's stage is csDead, and majority of Prima C code is prepended with this call, thus rejecting invalid references on early stage. If it is desired to get the C mate for objects that are in csDead stage, use `gimme_the_real_mate()` function instead.

init

Object's method init() is responsible for setting all its initial properties to the object, but all code that is executed inside init must be aware that the object's stage is csConstructing. init() consists of two parts: calling of ancestor's init() and setting properties. Examples are many in both C and perl code, but in short it looks like:

```
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 into profile, but the mechanism guarantees that all keys that are listed in profile_default() are conveyed to init(). For explicit checking of key presence pexists() macro is used, and pdelete() is used for key deletion, although is it not recommended to use pdelete() inside init().

Object creation and returning

As described is 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-> monochrome ? 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(), have to increase and decrease object's reference count if some perl functions are to be executed before returning object to perl, otherwise it might be destroyed before its time.

```
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 would be assigned to a scalar, for example.

Thus

```
$c = Prima::Object-> create();
```

and Prima::Object-> create;

have different results. But for some classes, namely Widget ant its descendants, and also for Timer, AbstractMenu, Printer and Clipboard the code above would have same result - the objects would not be killed. That is because these objects call Component_attach() during init-stage, automatically increasing their reference count. Component_attach() and its reverse Component_detach() account list of objects, attributed to each other. Object can be attached to multiple objects, but cannot be attached more that once to another object.

Notifications

All Prima::Component descendants are equipped with the mechanism that allows multiple user callbacks routines to be called on different events. This mechanism is used heavily in event-driven programming. Component_notify() is used to call user notifications, and its format string has same format as accepted by perl_call_indirect(). The only difference that it always has to be prepended with '<s', - this way the call success flag is set, and first parameter have to be the name of the notification.

```
Component_notify( self, "<sH", "Paint", self);
Component_notify( self, "<sPii", "MouseDown", self, point, int, int);
```

Notifications mechanism accounts the reference list, similar to attach-detach mechanism, because all notifications can be attributed to different objects. The membership in this list does not affect the reference counting.

Multiple property setting

Prima::Object method set() is designed to assign several properties at one time. Sometimes it is more convenient to write

```
$c-> set( index => 10, name  => "Sample" );
```

than to invoke several methods one by one. set() performs this calling itself, but for performance reasons it is possible to overload this method and code special conditions for multiple assignment. As an example, Prima::Image type conversion code is exemplified:

```
void
Image_set( Handle self, HV * profile)
{
```

```
   ...
   if ( pexist( type))
   {
      int newType = pget_i( type);
      if ( !itype_supported( newType))
         warn("RTC0100: 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);
}
```

If type conversion is performed along with palette change, some efficiency is gained by supplying both 'type' and 'palette' parameters at a time. Moreover, because ordering of the fields is not determined by default (although that be done by supplying '__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 it might be only solution to code Class_set() explicitly.

If it is desired to specify exact order how atomic properties have to be called, __ORDER__ anonymous array have to be added to 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 an application. There can be only one Application instance at a time, or none at all.

Macros and functions

dG_EVAL_ARGS

Defines 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 result under ancestorName key

gimme_the_vmt

```
PVMT( const char *className);
```

Returns 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 nilHandle if object's 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 pointer to a value, if found, nil otherwise

prima_hash_delete

```
void*( PHash self, const void *key, int keyLen, Bool kill);
```

Deletes hash key and returns associated value. if kill is true, calls free() on the value and returns nil.

prima_hash_store

```
void( PHash self, const void *key, int keyLen, void *val);
```

Stores new value into hash. If the key is already present, old value is overwritten.

prima_hash_count

```
int(PHash self)
```

Returns 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 nil is returned. action have to be function declared as

```
Bool action_callback( void * value, int keyLen, void * key, void * params);
```

params is a pointer to an 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, described in perlguts manpage. These functions aliased to a code with the workaround of perl bug which emerges when 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 perl pointer to a method searched by a scalar and a name If cacheIt true, caches the hierarchy traverse result for a speedup.

query_method

```
CV * ( Handle object, char *methodName, Bool cacheIt);
```

Returns perl pointer to a method searched by an object and a name If cacheIt 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);
```

Core function for calling Prima methods. Is used by the following three functions, but is never called directly. Format is described in `Calling perl code` section.

call_perl

```
SV * ( Handle self, char *subName, const char *format, ...);
```

Calls method of an object pointer by a Handle

sv_call_perl

```
SV * ( SV * mate, char *subName, const char *format, ...);
```

Calls method of an object pointed by a SV*

cv_call_perl

```
SV * ( SV * mate, Sv * coderef, const char *format, ...);
```

Calls arbitrary perl code with mate as first parameter. Used in notifications mechanism.

Object_create

```
Handle( char * className, HV * profile);
```

Creates an exemplar of className class with parameters in profile. Never returns nilHandle, throws an exception instead.

create_object

```
void*( const char *objClass, const char *format, ...);
```

Convenience wrapper to Object_create. Uses format specification that is described in `Calling perl code`.

create_instance

```
Handle( const char * className)
```

Convenience call to `Object_create` with parameters in hash 'profile'.

Object_destroy

```
void( Handle self);
```

Destroys object. One of few Prima function that can be called in re-entrant fashion.

Object_alive

```
void( Handle self);
```

Returns non-zero is object is alive, 0 otherwise. In particular, current implementation returns 1 if 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);
```

restricts object pointer from deletion after Object_destroy(). Can be called several times on an object. Increments Object. protectCount.

unprotect_object

```
void( Handle obj);
```

Frees object pointer 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 to 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 result of executed perl code and stores them into hv.

pexist

```
Bool(char*key)
```

Return true if a key is present into hash 'profile'

pdelete

```
void(char*key)
```

Deletes a key in hash 'profile'

pget_sv, pget_i, pget_f, pget_c, pget_H, pget_B

```
TYPE(char*key)
```

Returns value of (SV*, int, float, char*, Handle or Bool) that is associated to a key in 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 a 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 hash 'profile' and increments reference count to a newly created scalar.

pset_sv_noinc

```
void(char* key, SV * sv)
```

Assigns scalar value to a key in hash 'profile' without reference count increment.

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 list data.

plist_destroy

```
void ( PList self);
```

Destroys list data and frees 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 items that is located at given index or nilHandle if the index is out of range.

list_delete

```
void( PList self, Handle item);
```

Removes the item from list.

list_delete_at

```
void( PList self, int index);
```

Removes the item located at given index from a list.

list_delete_all

```
void ( PList self, Bool kill);
```

Removes all items from the list. If kill is true, calls free() on every item before.

list_first_that

```
int( PList self, void * action, void * params);
```

Enumerates all list entries, calling action procedure on each. If action returns true, enumeration stops and the index is returned. Otherwise -1 is returned. action have to be a function declared as

```
Bool action_callback( Handle item, void * params);
```

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.

8.2 Prima::codecs

How to write a codec for Prima image subsystem

Description

How to write a codec for 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 particular library is a tedious task, especially if one wants many formats. 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 show it. Everything else comes later - if ever. So, in a way to not scare you off, we start it simple.

Load

Define a callback function like:

```
static Bool
load( PImgCodec instance, PImgLoadFileInstance fi)
{
}
```

Just that function is not enough for whole mechanism to work, but bindings will come later. Let us imagine we work with an imaginary library libduff, that we want to load files of .duf format. *[To discern imaginary code from real, imaginary will be prepended with _ - like, _libduff_loadfile]*. So, we call _libduff_loadfile(), that 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 4-bytes boundary,
   // happily libduff has 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; so we can use it if libduff trusts handles vs names:

```
{
  _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 have to implement minimal open_load() and close_load(). Simplest open_load() returns non-null pointer - it is enough to report 'o.k'

```
static void *
open_load( PImgCodec instance, PImgLoadFileInstance fi)
{
   return (void*)1;
}
```

Its result will be available in `PImgLoadFileInstance-> instance`, just in case. If it was dynamically allocated, free it in close_load(). Dummy close_load() is doing simply nothing:

```
static void
close_load( PImgCodec instance, PImgLoadFileInstance fi)
{
}
```

Writing to `PImage-> data`

As mentioned above, Prima insists on keeping its image data in 32-bit aligned scanlines. If libduff allows reading from file by scanlines, we can use this possibility as well:

```
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 image is filled in reverse - Prima images are built like classical XY-coordinate grid, where Y ascends upwards.

Here ends the simple part. You can skip down to the *Registering with image subsystem* entry part, if you want it fast.

Single-frame loading

Palette

Our libduff can be black-and-white in two ways - where 0 is black and 1 is white and vice versa. While 0B/1W is perfectly corresponding to imbpp1 | imGrayScale and no palette operations are needed (Image cares automatically about these), 0W/1B is although black-and-white grayscale but should be treated like general imbpp1 type.

```
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;
}
```

NB. Image creates palette with size calculated by exponent of 2, since it can't know beforehand of the actual palette size. If color palette for, say, 4-bit image contains 15 of 16 possible for 4-bit image colors, code like

```
i-> palSize = 15;
```

does the trick.

Data conversion

As mentioned before, Prima defines image scanline size to be aligned to 32 bits, and the formula for lineSize calculation is

```
lineSize = (( width * bits_per_pixel + 31) / 32) * 4;
```

Prima defines number of converting routines between different data formats. Some of them can be applied to scanlines, and some to whole image (due sampling algorithms). These are defined in img_conv.h, and probably ones that you'll need would be `bc_format1_format2`, which work on scanlines and probably ibc_repad, which combines some `bc_XX_XX` with 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 easy method for determining this situation, but you have to convert your data in appropriate way to keep picture worthy of its name. Note the BYTEORDER symbol that is defined (usually) in sys/types.h

Load with no data

If a high-level code just needs image information rather than all its bits, codec can provide it in a smart way. Old code will work, but will eat memory and time. A flag `PImgLoadFileInstance-> noImageData` is indicating if image data is needed. On that condition, codec needs to report only dimensions of the image - but the type must be set anyway. Here comes 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 4-bytes boundary,
    // happily libduff has 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 string literal upon the expansion. Hash used for storage is a lexical of type `HV*`. Code

```
HV * profile = fi-> frameProperties;
pset_i( width, _l-> width);
```

is a prettier way for

```
hv_store(
    fi-> frameProperties,
    "width", strlen( "width"),
    newSViv( _l-> width),
    0);
```

hv_store(), HV's and SV's along with other funny symbols are described in perlguts.pod in Perl installation.

Return extra information

Image attributes are dimensions, type, palette and data. However, it is only Prima point of view - different formats can supply number of extra information, often irrelevant but sometimes useful. From perl code, Image has a hash reference 'extras' on object, where comes all this stuff. Codec can report also such data, storing it in `PImgLoadFileInstance-> frameProperties`. Data should be stored in native perl format, so if you're not familiar with perlguts, you better read it, especially if you want return arrays and hashes. But just in simple, you can return:

1. integers: pset_i(integer, _l-> integer);
2. floats: pset_f(float, _l-> float);
3. strings: pset_c(string, _l-> charstar); - note - no malloc codec from you required
4. prima objects: pset_H(Handle, _l-> primaHandle);
5. SV's: pset_sv_noinc(scalar, newSVsv(sv));
6. hashes: pset_sv_noinc(scalar, (SV *) newHV()); - hashes created through newHV() can be filled just in the same manner as described here
7. arrays: pset_sv_noinc(scalar, (SV *) newAV()); - arrays (AV) are described in perlguts also, but most useful function here is av_push. To push 4 values, for example, follow 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 flag `PImgLoadFileInstance-> loadExtras`. Codec may skip this flag, the extra information will not be returned, even if `PImgLoadFileInstance-> frameProperties` was changed. However, it is advisable to check for the flag, just for an efficiency. All keys, possibly assigned to frameProperties should be enumerated for high-level code. These strings should be represented into `char ** PImgCodecInfo-> loadOutput` array.

```
static char * loadOutput[] = {
   "hotSpotX",
   "hotSpotY",
   nil
};

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 X11 bitmap can provide location of hot spot, two integers, X and Y. The type of the data is not specified.

Loading to icons

If high-level code wants an Icon instead of an Image, Prima takes care for producing and-mask automatically. However, if codec knows explicitly about transparency mask stored in a file, it might change object in the way it fits better. Mask is stored on Icon in a `-> mask` field.

a) Let us imagine, that 4-bit image always carries a transparent color index, in 0-15 range. In this case, following code will create desirable 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.
b) if explicit bit mask is given, code will be like:

```
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 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 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).

open_load() and close_load()

open_load() and close_load() are used as brackets for load requests, and although they come to full power in multiframe load requests, it is very probable that correctly written codec should use them. Codec that assigns `false` to `PImgCodecInfo-> canLoadMultiple` claims that it cannot load those images that have index different from zero. It may report 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 codec can provide some file information without loading frames at all. It can be any information, of whatever kind. It have to be stored into the hash `PImgLoadFileInstance-> fileProperties`, to be filled once on open_load(). The only exception is `PImgLoadFileInstance-> frameCount`, which can be filled on open_load(). Actually, frameCount could be filled on any load stage, except close_load(), to make sense in frame positioning. Even single frame codec is advised to fill this field, at least to tell whether file is empty (frameCount == 0) or not (frameCount == 1). More about frameCount comes into chapters dedicated to multiframe requests. For strictly single-frame codecs it is therefore advised to care for open_load() and close_load().

Load input

So far codec is expected to respond for noImageData hint only, and it is possible to allow a high-level code to alter 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 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 load_defaults() function. This functions returns newly created (by calling newHV()) hash, with accepted keys and their default (and always valid) value pairs. Example below defines speed_vs_memory integer value, that should be 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

Image subsystem defines no severity gradation for codec errors. If error occurs during load, codec returns false value, which is `null` on open_load() and `false` on load. It is advisable to explain the error, otherwise the user gets just "Loading error" string. To do so, error message is to be copied to `PImgLoadFileInstance-> errbuf`, which is `char[256]`. On an extreme severe error codec may call croak(), which jumps to the closest G_EVAL block. If there is no G_EVAL blocks then program aborts. This condition could also happen if 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

In order to indicate that a codec is ready to read multiframe images, it must set `PImgCodecInfo-> canLoadMultiple` flag to true. This only means, that codec should respond to the `PImgLoadFileInstance-> frame` field, which is integer that can be in range from `0` to `PImgLoadFileInstance-> frameCount - 1`. It is advised that codec should change the frameCount from its original value `-1` to 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 it strongly 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 an loading error or an end-of-file condition. Codec may initialize frameCount at any time during open_load() or load(), even together with false return value.

Saving

Approach for handling saving requests is very similar to a load ones. For the same reason and with same restrictions functions save_defaults() open_save(), save() and close_save() are defined. Below shown a typical saving code and highlighted differences from load. As an example we'll take existing codec_X11.c, which defines extra hot spot 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)
{
}
```

Save request takes into account defined supported types, that are defined in `PImgCodecInfo-> saveTypes`. Prima converts image to be saved into one of these formats, before actual save() call takes place. Another boolean flag, `PImgSaveFileInstance-> append` is summoned to govern appending to or rewriting a file, but this functionality is under design. Its current value is a hint, if true, for a codec not to rewrite but rather append the frames to an existing file. Due to increased complexity of the code, that should respond to the append hint, this behavior is not required.

Codec may set two of PImgCodecInfo flags, canSave and canSaveMultiple. Save requests will never be called if canSave is false, and append requests along with multiframe save requests would be never invoked for a codec with canSaveMultiple set to false. Scenario for a multiframe save request is the same as for a load one. All the issues concerning palette, data converting and saving extra information are actual, however there's no corresponding flag like loadExtras - codec is expected to save all information what it can extract from `PImgSaveFileInstance-> objectExtras` hash.

Registering with image subsystem

Finally, the code have to be registered. It is not as illustrative but this part better not to be oversimplified. A codec's callback functions are set into ImgCodecVMT structure. Those function slots that are unused should not be defined as dummies - those are already defined and gathered under struct CNullImgCodecVMT. That's why all functions in the illustration code were defined as static. A codec have to provide some information that Prima uses to decide what codec should load this particular file. If no explicit directions given, Prima asks those codecs whose file extensions match to file's. init() should return pointer to the filled struct, that describes codec's capabilities:

```
// extensions to file - might be several, of course, thanks to dos...
static char * myext[] = { "duf", "duff", nil };
```

```
// 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
   nil,    // features
   "",     // module
   true,   // canLoad
   false,  // canLoadMultiple
   false,  // canSave
   false,  // canSaveMultiple
   mybpp,  // save types
   nil,    // 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 into `PImgCodec-> instance`, and info into `PImgCodec-> info`. If dynamic memory was allocated for these structs, it can be freed on done() invocation. Finally, the function that is invoked from Prima, is the only that required to be exported, 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, nil);
}
```

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 library name, that is required for a compilation with Prima. File with the codec should be called codec_duff.c (is our case) and put into img directory in Prima source tree. Following these rules, Prima will be assembled with libduff.a (or duff.lib, or whatever, the actual library name is system dependent) - if the library is present.

8.3 gencls

Class interface compiler for Prima core modules

Synopsis

```
gencls --h --inc --tml -O -I<name> --depend --sayparent filename.cls
```

Description

Creates headers with C macros and structures for Prima core module object definitions.

Arguments

gencls accepts the following arguments:

–h

Generates .h file (with declarations to be included in one or more files)

–inc

Generates .inc file (with declarations to be included in only file)

-O

Turns optimizing algorithm for .inc files on. Algorithm is based on an assumption, that some functions are declared identically, therefore the code piece that handles the parameter and result conversion can be shared. With `-O` flag on, a thunk body is replaced to a call to a function, which name is made up from all method parameters plus result. Actual function is not written in .inc file, but in .tml file. All duplicate declarations from a set of .tml files can be removed and the reminder written to one file by the *tmlink* entry utility.

–tml

Generates .tml file. Turns `-O` automatically on.

-Idirname

Adds a directory to a search path, where the utility searches for .cls files. Can be specified several times.

–depend

Prints out dependencies for a given file.

–sayparent

Prints out the immediate parent of a class inside given file.

Syntax

In short, the syntax of a .cls file can be described by the following scheme:

```
[ zero or more type declarations ]
[ zero or one class declaration ]
```

Gencls produces .h, .inc or .tml files, with a base name of the .cls file, if no object or package name given, or with a name of the object or the package otherwise.

Basic scalar data types

Gencls has several built-in scalar data types, that it knows how to deal with. To 'deal' means that it can generate a code that transfers data of these types between C and perl, using XS (see *perlguts*) library interface.

The 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 int. The data undergo no conversion to int in transfer process, but it is stored instead to perl scalar using newSViv() function, which, in turn, may lose bits or a sign.

Derived data types

The syntax for a new data types definition is as follows:

```
<scope> <prefix> <id> <definition>
```

A scope can be one of two pragmas, `global` or `local`. They hint the usage of a new data type, whether the type will be used only for one or more objects. Usage of `local` is somewhat resembles C pragma static. Currently the only difference is that a function using a complex local type in the parameter list or as the result is not a subject for `-O` optimization.

Scalar types

New scalar types may only be aliased to the existing ones, primarily for C coding convenience. 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 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 .h header file in form

```
#define id1 id2
```

C macros with parameters are not allowed. id1 and id2 are not required to be present in .cls name space, and no substitution during .cls file processing is made. This pragma usage is very limited.

Complex types

Complex data types can be arrays, structs and hashes. They can be a combination or a vector of scalar (but not complex) data types.

Gencls allows several combinations of complex data types that C language does not recognize. These will be described below.

Complex data types do not get imported into perl code. A perl programmer must conform to the data type used when passing parameters to 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 array references are not used, and the number of items in all array parameters must be exactly as the dimensions of the arrays.

Note: the following declaration will not compile with C compiler, as C cannot return arrays. However it is not treated as an error by gencls:

```
Array func();
```

Structs

Syntax:

```
<scope> @id {
   <basic_scalar_type> <id>;
   ...
   <basic_scalar_type> <id>;
};
```

Example:

```
global @Struc {
   int    number;
   string id;
}
```

Example of 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 array references are not used, and both number and order of items in all array parameters must be set exactly as dimensions and order of the structs. Struct field names are not used in perl code as well.

Hashes

Syntax:

```
<scope> %id {
   <basic_scalar_type> <id>;
   ...
   <basic_scalar_type> <id>;
};
```

Example:

```
global %Hash {
   int    number;
   string id;
}
```

Example of 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 hash references are used and returned. When a hash is passed from perl code it might have some or all fields unset. The C structure is filled and passed to a C function, and the fields that were unset are assigned to a corresponding C_TYPE_UNDEF value, where TYPE is one of NUMERIC, STRING and POINTER literals.

Back conversion does not count on these values and always returns all hash keys with a corresponding pair.

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 during initialization code. A namespace can be either `object` or `package`.

The package namespace syntax allows only declaration of functions inside a `package` block.

```
package <Package ID> {
    <function description>
    ...
}
```

The object namespace syntax includes variables and properties as well as functions (called methods in the object syntax). The general object namespace syntax is

```
object <Class ID> [(Parent class ID)] {
   <variables>
   <methods>
   <properties>
}
```

Within an object namespace the inheritance syntax can be used:

```
object <Class ID> ( <Parent class ID>)  { ... }
```

or a bare root object description (with no ancestor)

```
object <Class ID> { ... }
```

for the object class declaration.

Functions

Syntax:

```
[<prefix>] <type> <function_name> (<parameter list>) [ => <alias>];
```

Examples:

```
      int   package_func1( int a, int b = 1) => c_func_2;
      Point package_func2( Struc * x, ...);
method void  object_func3( HV * profile);
```

A prefix is used with object functions (methods) only. More on the prefix in 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 type conversion that corresponds to the rules described above in the *Basic scalar data types* entry section.

If a function has parameters and/or result of a type that cannot be converted automatically between C and perl, it gets declared but not exposed to perl namespace. The corresponding warning is issued. It is not possible using gencls syntax to declare a function with custom parameters or result data. For such a purpose the explicit C declaration of code along with `newXS` call must be made.

Example: ellipsis (...) cannot be converted by gencls, however 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 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.

Default parameters can be as many as possible, but they have to be in the end of the function parameter list. 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. Gencls expects a conformant function to be present in C code, with fixed name and parameter list. However, if the task of such function is a wrapper to an identical function published under another name, aliasing can be preformed to save both code and speed.

Example:

```
package Package {
   void func( int x) => internal;
}
```

A function declared in that way will not call Package_func() C function, but internal() function instead. The only request is that internal() function must have identical parameter and result declaration to a func().

Inline hash

A handy way to call a function with a hash as a parameter from perl was devised. If a function is declared with the last parameter or type `HV*`, then parameter translation from perl to C is performed as if all the parameters passed were a hash. This hash is passed to a C function and it's content 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, which perl code is typical

```
sub init
{
   my %ret = shift-> SUPER::init( @_);
   ...
   return %ret;
}
```

and C code is usually

```
void Obj_init ( HV * profile) {
    inherited init( profile);
    ... [ modify profile content ] ...
}
```

Methods

Methods are functions called in a context of an object. Virtually all methods need to have an access to an object they are dealing with. Prima objects are visible in C as Handle data type. Such Handle is actually a pointer to an object instance, which in turn contains a pointer to the object virtual methods table (VMT). To facilitate an OO-like syntax, this Handle parameter is almost never mentioned in all methods of an object description in a cls file, although being implicit counted, so every cls method declaration

```
method void a( int x)
```

for an object class Object is reflected in C as

```
void Object_a( Handle self, int x)
```

function declaration. Contrary to package functions, that gencls is unable to publish if it is unable to deal with the unsupported on unconvertible parameters, there is a way to issue such a declaration with a method. The primary use for that is the method name gets reserved in the object's VMT.

Methods are accessible in C code by the direct name dereferencing of a `Handle self` as a corresponding structure:

```
((( PSampleObject) self)-> self)-> sample_method( self, ...);
```

A method can have one of six prefixes that govern C code generation:

method

This is the first and the most basic method type. It's prefix name, `method` is therefore was chosen as the most descriptive name. Methods are expected to be coded in C, the object handle is implicit and is not included into a .cls description.

```
method void a()
```

results in

```
void Object_a( Handle self)
```

C declaration. A published method automatically converts its parameters and a result between C and perl.

public

When the methods that have parameters and/or result that cannot be automatically converted between C and perl need to be declared, or the function declaration does not fit into C syntax, a `public` prefix is used. The methods declared with `public` is expected to communicate with perl by means of XS (see *perlxs*) interface. It is also expected that a `public` method creates both REDEFINED and FROMPERL functions (see the *Prima::internals* section for details). Examples are many throughout Prima source, and will not be shown here. `public` methods usually have void result and no parameters, but that does not matter much, since gencls produces no conversion for such methods.

import

For the methods that are unreasonable to code in C but in perl instead, gencls can be told to produce the corresponding wrappers using `import` prefix. This kind of a method can be seen as `method` inside-out. `import` function does not need a C counterpart, except the auto-generated code.

static

If a method has to be able to work both with and without an object instance, it needs to be prepended with `static` prefix. `static` methods are all alike `method` ones, except that `Handle self` first parameter is not implicitly declared. If a `static` method is called without an object (but with a class), like

```
Class::Object-> static_method();
```

its first parameter is not a object but a "Class::Object" string. If a method never deals with an object, it is enough to use its declaration as

```
static a( char * className = "");
```

but is if does, a

```
static a( SV * class_or_object = nil);
```

declaration is needed. In 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. `weird` prefix denotes a method that combined properties both from `static` and `public`. In other words, gencls generates no conversion code and expects no `Handle self` as a first parameter for such a method. As an example Prima::Image::load can be depicted, which can be called using a wide spectrum of calling semantics (see the *Prima::image-load* section for details).

c_only

As its name states, `c_only` is a method that is present on a VMT but is not accessible from perl. It can be overloaded from C only. Moreover, it is allowed to register a perl function with a name of a `c_only` method, and still these entities will be wholly independent from each other - the overloading will not take place.

NB: methods that have result and/or parameters data types that can not be converted automatically, change their prefix to `c_only`. Probably this is the wrong behavior, and such condition have to signal an error.

Properties

Prima toolkit introduces an entity named property, that is expected 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 pair of methods like Object::set_color and Object::get_color, a property Object::color is devised. A property is a method with the special considerations, in particular, when it is called without parameters, a 'get' mode is implied. In contrary, if it is called with one parameter, a 'set' mode is triggered. Note that on both 'set' and 'get' invocations `Handle self` first implicit parameter is always present.

Properties can operate with different, but fixed amount of parameters, and perform a 'set' and 'get' functions only for one. By default the only parameter is the implicit `Handle self`:

```
property char * name
```

has C counterpart

```
char * Object_name( Handle self, Bool set, char * name)
```

Depending on a mode, `Bool set` is either `true` or `false`. In 'set' mode a C code result is discarded, in 'get' mode the parameter value is undefined.

The syntax for multi-parameter property is

```
property long pixel( int x, int y);
```

and 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 property name are always initialized, in both 'set' and 'get' modes.

Instance variables

Every object is characterized by its unique internal state. Gencls syntax allows a variable declaration, for 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 recognized. As a workaround, pointers to typedef'd entities are used. Example:

```
object SampleObject {
   int x;
   List list;
   struct { int x } s; # illegal declaration
}
```

Variables are accessible in C code by direct name dereferencing of a `Handle self` as a corresponding structure:

```
 (( PSampleObject) self)-> x;
```

9 Miscellaneous

9.1 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 on X, Win32 and OS/2 PM.

Yeah, right. So what is Prima again?

Ok. 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 really 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.

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. The latter and set of 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 is almost all files of the toolkit.

I'd like to contribute.

You can do this is 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 thing that needs a particular attention, since my native language is not English, so if there are volunteers for polishing of 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, instructions on how to subscribe to the Prima mailing list, documentation, and some other useful info.

What is better, source or binary?

Depends where your are and what are your goals. On unix, the best is to use the source. On win32 and os2 the binaries probably are preferred. If you happen to use cygwin you probably still better off using the source.

How to install binary distribution?

First, check if you've downloaded Prima binary for the correct version of Perl. For win32 ActiveState builds, difference in the minor digits of the Perl version shouldn't be a problem, for example, binary distribution for Perl build #805 should work with Perl build #808, etc etc.

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 source?

Type the following:

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

If the 'perl Makefile.PL' fails complaining to strange errors, you can check makefile.log to see if anything is wrong. A typical situation here is that Makefile.PL may report that is cannot find Perl library, for example, where there actually it invokes the compiler in a wrong way.

Note, that in order to get Prima working from sources, your system must contain graphic libraries, such as libungif or ligjpeg, for Prima to load graphic files.

What's about the graphic libraries?

To load and save images, Prima employs graphic libraries. Such as, to load GIF files, libungif 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 really useful. If Makefile.PL wouldn't find any of the supported graphic libraries, it would abort unless WANTNOCODECS=1 parameter was supplied to it.

On every supported platform Prima can make use of the following graphic libraries:

```
libX11   - XBM bitmaps
libXpm   - Xpm pixmaps
libjpeg  - JPEG images
libungif - GIF images
libpng   - PNG images
libtiff  - tiff images
```

Alternatively, on win32 and os2 there is a binary PRIGPARH library distributed together with the Prima binary distributions, which supports its own set of graphic files. The PRIGPARH is a modified GBM graphic library, which (GBM) is no longer supported, but nevertheless it is useful for Prima. The use of PRIGPARH is preferred on win32 and os2, and Makefile.PL would favor it before the other graphic libraries. To compile and run Prima with PRIGPARH, library (.lib or .a) and runtime (.dll) files must be present in the LIBPATH and PATH, correspondingly.

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 cost of a lacking support for a graphic format.

How'd I check what libraries are compiled in?

```
perl -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 a weird directory so Makefile.PL must be told to use it by adding LIBPATH+=/some/weird/lib, and possibly INCPATH+=/some/weird/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 error

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 into the Prima mailing list.

Prima doesn't run

Again, there are reasons for Prima to fail during the start.

First, check whether all main files are installed correctly. *Prima.pm* must be in your perl directory, and Prima library file (*Prima.a* or *Prima.so* for unix, *Prima.dll* for win32, and *PrimaDI.dll* for os2) 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 something like

```
Can't locate Prima.pm in @INC
```

If Prima library or one of the libraries it depends on cannot be found, perl Dynaloader would complain. On win32 and os2 this usually happen when prigraph.dll (and/or priz.dll on os2) are not found. If this is the case, try to copy these files into your PATH, for example in C:/WINNT .

Prima error: Can't open display

This error happens when you've compiled Prima for X11, and no connection to X11 display can be established. Check your DISPLAY environment variable, or use –display parameter when running Prima. If you do not want Prima to connect to the display, for example, to use it inside of a CGI script, either use –no-x11 parameter or include `use Prima::noX11` statement in your program.

X11: my fonts are bad!

Check whether you've 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 perl tree, and under 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 a pdf file on the Prima web site www.prima.eu.org, which contains the same set of documentation but composed as a single book. Its sources are in utils/makedoc directory, somewhat rudimentary and require an installation of latex and dvips to produce one of tex, dvi, ps, or pdf targets.

I've found a bug!

Send the bug report into the mailing list.

PROGRAMMING

How can I use .fm files of 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 X11 display.

```
use Prima::noX11; # this prevents Prima from connecting to X11 display
use Prima;
my $i = Prima::Image-> load( ... )
```

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 governed by none of the `Prima::Edit` properties, you've to overload `::on_paint`. It is not that hard as you might think.

If the feature is generic enough, you can send a patch in the list.

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 a 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 own 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 `designScale` property of the parent window, which contains 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 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 to enumerate.

- Test your widget class with different default settings, such as colors, fonts, parent sizes, widget properties such as buffered and visible.
- Try to avoid special properties for `create`, 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 re-define notification types. These have large number of options, to be programmed once and then used as a DWIM helper. Consider for which notifications user callback routines (onXxxx) would be best to be called first, or last, whether a notification should be of multiple or single callback type.

  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 `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 `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, and outside inclusive-exclusive.

Note, that `buffered` property does not guarantee that the widget output would be actually buffered.

- Write some documentation and example of use.

How would I add my widget class to the VB palette?

Check Prima/VB/examples/Widgety.pm . This file, if loaded through 'Add widget' command in VB, adds example widget class and example VB property into the VB palette and Object Inspector.

How would I use unicode/UTF8 in Prima?

Basically,

```
$::application-> wantUnicodeInput(1)
```

is enough to tell Prima to provide input in Unicode/UTF8. Note, that if the data received in that fashion are to be put through file I/O, the `'utf8'` IO layer must be selected (see the *open* entry).

Prima can input and output UTF8 text if the underlying system capabilities support that (check Prima::Application::get_system_value, `sv::CanUTF8_Input` and `sv::CanUTF8_Output`). Displaying UTF8 text is transparent, 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 - keyboard and clipboard - can be treated and reported to Prima either as UTF8 or plain text, depending on `Prima::Application::wantUnicodeInput` property.

The keyboard input is 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 POD text that comes with my program / package ?

```
$::application-> open_help( $0 );
$::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, bu 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 hands: instead of entering the main loop with

```
run Prima;
```

one can write

```
while ( 1) {
   ... do some calculations ..
   $::application->yield;
}
```

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, and communicate with it via its stdin and/or stdout (see *perlipc* how to do that), and use the *Prima::File* section to asyncronously read data passed through a pipe or a socket.

Note: Win32 runtime library does not support asynchronous pipes, only asyncronous sockets. Cygwin does support both asyncronous pipes and sockets.

How do I post an asynchronous message?

`Prima::Component::post_message` method posts a message through the system event dispatcher and returns immediately; when the message is arrived, `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, that 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`, that doesn't have any interface elements on its own, and provides only page flipping function. The sub-widgets, therefore, are to be addressed as `$TabbedNotebook-> Notebook-> MyButton`.

9.2 Prima::Const

Predefined constants

Description

`Prima::Const` and the *Prima::Classes* section is 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 assembled 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 `Prima::` string.

This document provides description of all core-coded constants. The constants are also described in the articles together with the corresponding methods and properties. For example, `nt` constants are also described in 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::Auto           - mask update based on corner pixel values
```

apc:: - OS type

See the **get_system_info** entry in the *Prima::Application* section

```
apc::Os2
apc::Win32
apc::Unix
```

bi:: - border icons

See the **borderIcons** entry in the *Prima::Window* section

```
bi::SystemMenu  - system menu button and/or close button
                  ( usually with icon ) is shown
bi::Minimize    - minimize button
bi::Maximize    - maximize ( and eventual restore )
bi::TitleBar    - 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  - thick border with interactive resize capabilities
```

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
```

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::User                    user-defined icon
```

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 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 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 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::EditDialog* section

```
fdo::MatchCase
fdo::WordsOnly
fdo::RegularExpression
fdo::BackwardSearch
fdo::ReplacePrompt
```

fds:: - find / replace dialog scope type

See the *Prima::EditDialog* section

```
fds::Cursor
fds::Top
fds::Bottom
```

fe:: - file events constants

See the *Prima::File* section

```
fe::Read
fe::Write
fe::Exception
```

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
```

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
```

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 owner's right side
gm::GrowLoY     widget's bottom side is kept in constant
```

```
                distance from owner's top side
gm::GrowHiX     widget's right side is kept in constant
                distance from owner's right side
gm::GrowHiY     widget's top side is kept in constant
                distance from owner's top side
gm::XCenter     widget is kept in center on its owner's
                horizontal axis
gm::YCenter     widget is kept in center on its owner's
                vertical axis
gm::DontCare    widgets origin is maintained 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::PM
gui::Windows
gui::XLib
gui::GTK2
```

le:: - line end styles

See the **lineEnd** entry in the *Prima::Drawable* section

```
le::Flat
le::Square
le::Round
```

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
```

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
ict::Ordered         - 8x8 ordered halftone dithering
ict::ErrorDiffusion  - error diffusion dithering with static palette
ict::Optimized       - error diffusion dithering with optimized palette
```

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      - sum of pixel values
is::Sum2     - sum of squares of pixel values
```

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
```

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
```

Message box icon and bell constants

```
mb::Error
mb::Warning
mb::Information
mb::Question
```

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
```

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::OutlineCollaps
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
```

sv:: - system value indices

See also the **get_system_value** entry in the *Prima::Application* section

```
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
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 system accepts color pointer icons.
sv::CanUTF8_Input    - 1 if system can generate key codes in unicode
sv::CanUTF8_Output   - 1 if system can output utf8 text
```

ta:: - alignment constants

Used in: the *Prima::InputLine* section, the *Prima::ImageViewer* section, the *Prima::Label* section, the *Prima::Terminals* section.

```
ta::Left
ta::Right
ta::Center

ta::Top
ta::Bottom
ta::Middle
```

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 text with respect to tab expansion
tw::ExpandTabs            - expands tab characters
tw::BreakSingle           - determines if text is broken to 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 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
```

9.3 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

Prima dispatches events by calling notifications registered on one or more objects interested in the events. Also, one event hook can be installed that would receive all events occurred on all objects. `Prima::EventHook` provides multiplex access to the core event hook and introduces set of dispatching rules so the user hook subs receive only a defined subset of events.

The filtering criteria are event names and object hierarchy.

API

install SUB, %RULES

Installs SUB into hook list using hash of RULES.

The SUB is called with variable list of parameters, formed so first passed parameters from `'param'` key (see below), then event source object, then event name, and finally parameters to the event. 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 further called.

Rules can contain the following keys:

event

Event is either a string, an array of strings, or `undef` value. In the latter case it is equal to `'*'` string, which selects all events to be passed in the SUB. A string is either name of an event, or one of pre-defined event groups, declared in `%groups` package hash. The group names are:

```
ability
focus
geometry
```

```
keyboard
menu
mouse
objects
visibility
```

These contain respective events. See source for detailed description.

In case `'event'` key is an array of strings, each of the strings is also name of either an event or a group. In this case, if `'*'` string or event duplicate names are present in the list, SUB is called several times which is obviously inefficient.

object

A Prima object, or an array of Prima objects, or undef; the latter case matches all objects. If an object is defined, the SUB is called if event source is same as the object.

children

If 1, SUB is called using 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 Prima object hierarchy, as filter object with `'children'` set to 1.

Setting together object to `undef` and children to 1 is inefficient.

param

A scalar or array of scalars passed as first parameters to SUB whenever it is called.

deinstall SUB

Removes the hook sub for the hook list.

NOTES

`Prima::EventHook` by default automatically starts and stops Prima event hook mechanism when appropriate. If it is not desired, for example for your own event hook management, set `$auto_hook` to 0.

9.4 Prima::IniFile

Support of Windows-like initialization files

Description

The module contains a class, that provides mapping of text initialization file to a two-level hash structure. The first level is called sections, which groups the second level hashes, called items. Sections must have unique keys. The items hashes values are arrays of text strings. The methods, operated 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, 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 name of 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 array of values for ITEM in SECTION. If called in scalar context, and there is more than one value, the first value in list is returned.

items SECTION [HINTS]

Returns items in SECTION. HINTS parameters is 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 number of values in ITEM in SECTION.

read FILENAME, %PROFILE

Flushes the old content and opens 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 file, and the latter keep the precedence. Does not return any success values but, warns if any error is occurred.

replace_values SECTION, ITEM, @VALUES

Removes all values form 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 array of section names.

set_values SECTION, ITEM, @VALUES

Assigns VALUES to ITEM in SECTION. If number of new values are equal or greater than the number of the old, the method is 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 under name {`changed`}; in case it is `undef`, no actual write is performed.

9.5 Prima::IntUtils

Internal functions

Description

The module provides packages, containing common functionality for some standard classes. The packages are designed as a code containers, not as widget classes, and are to be used as secondary ascendants in the widget inheritance declaration.

Prima::MouseScroller

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.

Prima::IntIndents

Provides the common functionality for the widgets that delegate part of their surface to the border elements. A list box can be of an example, where its scroll bars and 3-d borders are such elements.

Properties

indents ARRAY

Contains four integers, specifying the breadth of decoration elements for each side. The first integer is width of the left element, the second - height of the lower element, the third - width of the right element, the fourth - height of the upper element.

The property can accept and return the array either as a 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 are related to the current size of a widget, however, can be overridden by specifying WIDTH and HEIGHT. TYPE is an integer, indicating the 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.

Prima::GroupScroller

The class is used for widgets that contain optional scroll bars, and provides means for their maintenance. The class is the descendant of the *Prima::IntIndents* section, and adjusts the the *indents* entry property when scrollbars are shown or hidden, or the *borderWidth* entry is changed.

The class does not provide range selection for the scrollbars; the descentant classes must implement that.

The descendant classes must follow the guidelines:

- A class must provide `borderWidth`, `hScroll`, and `vScroll` property keys in profile_default() . A class may provide `autoHScroll` and `autoVScroll` property keys in profile_default() .
- A class' init() method must set {`borderWidth`}, {`hScroll`}, and {`vScroll`} variables to 0 before the initialization, call `setup_indents` method, and then assign the properties from the object profile.

  If a class provides `autoHScroll` and `autoVScroll` properties, these must be set to 0 before the initialization.
- If a class needs to overload one of `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 3d-shade border around the widget.

Recommended default value: 2

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.

Properties

setup_indents

The method is never called directly; it should be called whenever widget layout is changed so that indents are affected. The method is a request to recalculate indents, depending on the 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` is called and internal widget layout is updated, to give a chance to follow-up the layout changes.

9.6 Prima::StdBitmap

Shared access to the standard toolkit bitmaps

Description

The toolkit contains *sysimage.gif* image library, which consists of a predefined set of images, used in several toolkit modules. To provide a unified access to the images this module can be used. The images are assigned a `sbmp::` constant, which is used as an index on a load request. If loaded successfully, images are cached and the successive requests return the cached values.

The images can be loaded as `Prima::Image` and `Prima::Icon` instances. To discriminate, two methods are used, correspondingly `image` and `icon`.

Synopsis

```
use Prima::StdBitmap;
my $logo = Prima::StdBitmap::icon( sbmp::Logo );
```

API

Methods

icon INDEX

Loads INDEXth image frame and returns `Prima::Icon` instance.

image INDEX

Loads INDEXth image frame and returns `Prima::Image` instance.

load_std_bmp INDEX, AS_ICON, USE_CACHED_VALUE, IMAGE_FILE

Loads INDEXth image frame from IMAGE_FILE and returns it as either a `Prima::Image` or as a `Prima::Icon` instance, depending on value of boolean AS_ICON flag. If USE_CACHED_VALUE boolean flag is set, the cached images loaded previously can be used. If this flag is unset, the cached value is never used, and the created image is not stored in the cache. Since the module's intended use is to provide shared and read-only access to the image library, USE_CACHED_VALUE set to 0 can be used to return non-shared images.

Constants

An index value passed to the methods must be one of `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::OutlineCollaps
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

`$sysimage` scalar is initialized to the file name to be used as a source of standard image frames by default. It is possible to alter this scalar at run-time, which causes all subsequent image frame request to be redirected to the new file.

9.7 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 only stress factor implemented is change of the default font size, which is set to different value every time the module is invoked.

To use the module functionality it is enough to include a typical

```
use Prima::Stress;
```

code, or, if the program is invoked by calling perl, by using

```
perl -MPrima::Stress program
```

syntax. The module does not provide any methods.

9.8 Prima::Tie

Tie widget properties to scalars or 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* property 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 property, in particular to popular properties as *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);
```

You can work directly with the text or items rather than at a remove. Furthermore, if the only reason you keep an object around after creation is to access its text or items, you no no longer need to do so:

```
tie @some_array, 'Prima::Tie::items', Prima::ListBox->create(@args);
```

As opposed to:

```
my $widget = Prima::ListBox->create(@args);
tie @some_array, 'Prima::Tie::items', $widget;
```

`Prima::Tie::items` requires `::items` property to be available on the widget. Also, it takes advantage of additional `get_items`, `add_items`, and the like if available.

Prima::Tie::items

The class is applicable to `Prima::ListViewer`, `Prima::ListBox`, `Prima::Header`, and their descendants, and in 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::ColorDialog`, `Prima::SpinEdit`, `Prima::Gauge`, `Prima::Slider`, `Prima::CircularSlider`, and `Prima::ScrollBar`.

9.9 Prima::Utils

Miscellanneous routines

Description

The module contains several helper routines, implemented in both C and perl. Whereas the C-coded parts are accessible only if 'use Prima;' statement was issued prior to the 'use Prima::Utils' invocation, the perl-coded are always available. This makes the module valuable when used without the rest of toolkit code.

API

alarm $TIMEOUT, $SUB, @PARAMS

Calls SUB with PARAMS after TIMEOUT milliseconds.

beep [FLAGS = mb::Error]

Invokes the system-depended sound and/or visual bell, corresponding to one of following constants:

```
mb::Error
mb::Warning
mb::Information
mb::Question
```

get_gui

Returns one of `gui::XXX` constants, reflecting the graphic user interface used in the system:

```
gui::Default
gui::PM
gui::Windows
gui::XLib
gui::GTK2
```

get_os

Returns one of `apc::XXX` constants, reflecting the platfrom. Currently, the list of the supported platforms is:

```
apc::Os2
apc::Win32
apc::Unix
```

ceil DOUBLE

Obsolete function.

Returns stdlib's ceil() of DOUBLE

find_image PATH

Converts PATH from perl module notation into a file path, and searches for the file in `@INC` paths set. If a file is found, its full filename is returned; otherwise `undef` is returned.

floor DOUBLE

Obsolete function.

Returns stdlib's floor() of DOUBLE

getdir PATH

Reads content of PATH directory 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 call of `stat` for every file.

path [FILE]

If called with no parameters, returns path to a directory, usually *~/.prima*, that can be used to contain 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 anonymous array to drive letters, used by the system. FIRST_DRIVE can be set to other value to start enumeration from. Some OSes can probe eventual diskette drives inside the drive enumeration routines, so there is a chance to increase responsiveness of the function it might be reasonable to set FIRST_DRIVE to `C:` string.

If the system supports no drive letters, empty array reference is returned (unix).

query_drive_type DRIVE

Returns one of `dt::XXX` constants, describing the type of 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
```

sound [FREQUENCY = 2000, DURATION = 100]

Issues a tone of FREQUENCY in Hz with DURATION 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 on one of the three formats:

```
#rgb
#rrggbb
#rrrgggbbb
```

and returns 24-bit RGB integer value.

9.10 Prima::Widgets

Miscellaneous widget classes

Description

The module was designed to serve as a collection of small widget classes that do not group well with the other, more purposeful classes. The current implementation contains the only class, `Prima::Panel`.

Prima::Panel

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 into `Prima::Label`'s.

Properties

borderWidth INTEGER

Width of 3d-shade border around the widget.

Default value: 1

image OBJECT

Selects image to be drawn as a tiled background. If `undef`, the background is drawn with the background color.

imageFile PATH

Set the image FILE to be loaded and displayed. Is rarely used since does not return a loading success flag.

raise BOOLEAN

Style of 3d-shade border around the widget. If 1, the widget is 'risen'; if 0 it is 'sunken'.

Default value: 1

zoom INTEGER

Selects zoom level for image display. The acceptable value range is between 1 and 10.

Default value: 1

9.11 Prima::gp-problems

Problems, questionable or intricate topics in 2-D Graphics

Introduction

One of the most important goals of the Prima project is portability between different operating systems. Independently to efforts in keeping Prima internal code that it 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 minimal set of capabilities, that programmer can rely upon. Following items are guaranteedly supported by Prima:

Minimal capabilities

Distinct black and white colors

Line widths 0 and 1

One monospaced font

Solid fill

rop::Copy and rop::NoOper

Plotting primitives

SetPixel,GetPixel

Line,PolyLine,PolyLines

Ellipse,Arc,Chord,Sector

Rectangle

FillPoly

FillEllipse,FillChord,FillSector

TextOut

PutImage,GetImage

Information services

GetTextWidth,GetFontMetrics,GetCharacterABCWidths

GetImageBitsLayout

Properties

color

backColor

rop

backRop

lineWidth

lineJoin

lineStyle

fillPattern

fillPolyWinding

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 have to follow all obligations for the mode. For example, if platform supports full functionality for black color but limited functionality for the other colors, the wrapping code should not allow color property to be writable then.

Inevident issues

Colors

Black and white colors on paletted displays

Due the fact that paletted displays employ indexed color representation, 'black' and 'white' indices are not always 0 and 2^n-1, so result of raster image operations may look garbled (X). Win32 and OS/2 protect themselves from this condition by forcing white to be the last color in the system palette.

Example: if white color on 8-bit display occupies palette index 15 then 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 inevident issue happens mostly on 15- and 16-bits pixel displays. Internal color representation for the white color on a 15-color display (assuming R,G and B are 5-bits fields) is

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

that equals to 0xf8f8f8. (All)

Advise: do not check for 'blackness' and 'whiteness' merely by comparing a pixel value.

Pixel value coding

Status: internal

It is not checked how does Prima behave when a pixel value and a platform integer use different bit and/or byte priority (X).

Filled shapes

Dithering

If a non-solid pattern is selected and a background and/or a foreground color cannot be drawn as a solid, the correct rendering requires correspondingly 3 or 4 colors. Some rendering engines (Win9X) fail to produce correct results.

Overfill effect

In complex shapes (FillPoly, for example) the platform renderer can fill certain areas two or more times. Whereas the effect is not noticeable with rop::CopyPut, the other raster operations (like rop::Xor) produce incorrect picture. (OS/2)

NB - has nothing in common with the fill winding rule.

Workaround: Do not use raster operations with complex filled shapes

Pattern offset

For a widget that contains a pattern-filled shape, its picture will be always garbled after scrolling, because it is impossible to provide an algorithm for a correct rendering without a prior knowledge of the widget nature. (All)

Workaround: Do not use patterned backgrounds. Since the same effect is visible on dithered backgrounds, routine check for pure color might be applied.

Lines

Line caps over patterned styles

It is not clear, whether gaps between dashes should be a multiple to a line width or not. For example, lp::DotDot looks almost as a solid line when lineWidth is over 10 if the first (non-multiple) tactic is chosen. From the other hand it is hardly possible to predict the plotting strategy from a high-level code. The problem is related more to Prima design rather than to a platform-specific code. (All)

Workaround: use predefined patterns (lp::XXX)

Line joins

Joint areas may be drawn two (or more) times - the problem emerges if logical ROP (rop::Xor) is chosen.(OS/2)

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 afterwards.(OS/2, X).

Impact: system-dependent font description may not match to Prima's.

Advise: do not try to deduce Prima font metrics from system-dependent ones and vice versa.

Transparent plotting

No internal function for drawing transparent bitmaps (like fonts). Therefore, if a font emulation is desired, special ROPs cannot be reproduced. (OS/2, Win9X, WinNT)

Impact: font emulation is laborsome, primarily because the glyphs have to be plotted by consequential anding and xoring a bitmap. Full spectrum of the raster operations cannot be achieved with this approach.

Kerning

Prima do not use text kernings, nor encourages underlying platform-specific code to use it - primarily because of its complexity. From the other hand, sometimes glyph position cannot be determined correctly if no information for the text kerning is provided. (Win9X)

Fractional text position

If the font glyphs have fractional widths, it might be observed that letters may change their position in a string.

Example: A set of glyphs has width of 8.6 pixels for each symbol. If the string "abcd" is drawn at position 0, then black part of "d" starts at 25th pixel, but if "cd" is drawn at 17th, as it supposed to be if the integer arithmetics is used, it starts at 24th pixel. (OS/2)

Solution: Do not rely to Drawable::get_text_width information, because it always returns integer value, but to Drawable::get_font_abc, which returns real values.

Text background

If a text is drawn with non-CopyPut raster operation, text background is not expected to be mixed with symbols - however this is hardly reachable, so results differs for different platforms.

Text background may be only drawn with pure (non-dithered) color (Win9X,WinNT) - but this is (arguably) a more correct behavior.

Advise: Do not use ::rop2 and text background for special effects

Internal platform features

Font change notification is not provided. (X, OS/2)

Raster fonts cannot be synthesized (OS/2, partly X)

Raster operations (ROPs)

Background raster operations are not supported (X,Win9X,WinNT) and foreground ROPs have limited number of modes (OS/2,X). 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. Behavior of the current implementation is that they do not.

Arcs

Platforms tend to produce different results for angles outside 0 and 2pi. Although Prima assures that correct plotting would be performed for any angle, minor inconsistencies may be noticed. If emulating, note that 2 and 4-pi arcs are not the same - for example, they look differently with rop::Xor.

Palettes

Static palettes

Some displays are unable to change their hardware palette, so detecting 8- or 4- bits display doesn't automatically mean that palette is writable.(X)

Widget::palette

Widget::palette property is used for explicit declaration of extra color needs for a widget. The request might be satisfacted in different ways, or might not at all. It is advisable not to rely onto platform behavior for the palette operations.

Dynamic palette change

It is possible (usually on 8-bits displays) for a display to change asynchronously its hardware palette in order 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 Widget::palette. System palette is switched every time as different windows moved to the front.

OS/2 - not implemented, but in principle the same as under win32.

X - Any application can easily ruin system color table. Since this behavior is such by design, no workaround can be applied here.

Bitmaps

Invalid scaling

Scaling is invalid (Win9X) or not supported (X). Common mistake is to not take into an account the fractional pixels that appear when the scaling factor is more than 1. This mistake can be observed in Win9X.

Workaround: none

Large scale factors

Request for drawing a bitmap might fail if large scaling factor is selected. (OS/2,Win9X,WinNT). This effect is obviously due that fact that these platforms scale the bitmap into a memory before the plotting takes place.

Platform-specific peculiarities

OS/2

Some ROPs are ambiguous - SRCTRANSPARENT, for example. Some times they work, some times they don't. The particular behavior depends on a video driver.

Circles cannot be drawn using an even diameter.

Fast GDI operations on HWND_DESKTOP may be delayed, thus GetPixel may return invalid pixel values.

Windows 9X

Amount of GDI objects can not exceed some unknown threshold - experiments show that 128 objects is safe enough.

No transformations.

Color cursor creation routine is broken.

Filled shapes are broken.

X

No transformations

No bitmap scaling

No font rotation

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

OS/2

Palettes are not implemented

Win32

Plotting speed of DeviceBitmaps is somewhat less on 8-bit displays than Images and Icons. It is because DeviceBitmaps are bound to their original palette, so putting a DeviceBitmap onto different palette drawable employs inefficient algorithms in order to provide correct results.

X

Image that was first drawn on a paletted Drawable always seen in 8 colors if drawn afterwards on a Drawable with the different palette. That is because the image has special cache in display pixel format, but cache refresh on every PutImage call is absolutely inappropriate (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.

9.12 Prima::X11

Usage guide for X11 environment

Description

This document describes subtle topics one must be aware when programming or using Prima programs under X11.

The document covers various aspects of the toolkit and their implementation details with guidelines of the expected use. Also, standard X11 user-level and programming techniques are visited.

Basic command-line switches

`--help`

Prints the command-line arguments available and exits.

`--display`

Sets X display address in Xlib notation. If not set, standard Xlib (`XOpenDisplay(null)`) behavior applies.

Example:

```
--display=:0.1
```

`--visual`

Sets X visual, to be used by default. Example:

```
--visual=0x23
```

`--sync`

Turn off X synchronization

`--bg`, `--fg`

Set default background and foreground colors. Example:

```
--bg=BlanchedAlmond
```

`--font`

Sets default font. Example:

```
--font='adobe-helvetica-medium-r-*-*--*-120-*-*-*-*-*-*'
```

`--no-x11`

Runs Prima without X11 display initialized. This switch can be used for programs that use only OS-independent parts of Prima, such as image subsystem or PostScript generator, in environments where X is not present, for example, a CGI script. Obviously, any attempt to create instance of `Prima::Application` or otherwise access X-depended code under such conditions causes the program to abort.

There are alternatives to use the command switch. First, there is module `Prima::noX11` for the same purpose but more convenient to use as

```
perl -MPrima::noX11
```

construct. Second, there is a technique to continue execution even if connection to a X server failed:

```
use Prima::noX11;
use Prima;

my $error = Prima::XOpenDisplay();
if ( defined $error) {
     print "not connected to display: $error\n";
} else {
     print "connected to display\n";
}
```

The the *Prima::noX11* section module exports a single function `XOpenDisplay` into `Prima` namespace, to connect to the X display explicitly. The display to be connected to is `$ENV{DISPLAY}`, unless started otherwise on command line (with –display option) or with parameter to the `XOpenDisplay` function.

This technique may be useful to programs that use Prima imaging functionality and may or may not use windowing capabilites.

X resources database

X11 provides XRDB, the X resource database, a keyed list of arbitrary string values stored on the X server. Each key is a combination of names and classes of widgets, each in string form. The key is constructed so the leftmost substring (name or class) corresponds to the top-level item in the hierarchy, usually the application name or class. Although the XRDB can be changed via native X API, it is rarely done by applications. Instead, the user creates a file, usually named .Xdefaults, which contains the database in the string form.

The format of .Xdefaults directly reflects XRDB capabilities, one of the most important of which is globbing, manifested via * (star) character. Using 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 ListBox class must have yellow background.

The application itself is responsible for parsing the strings and querying the XRDB. Also, both class names and widget names, as well as database values are fully defined in terms of the application. There are some guidelines though, for example, colors and fonts best described in terms, native to the X server. Also, classes and names are distinguished by the case: classes must begin with the upper register letter. Also, not every character can be stored in the XRDB database (space, for example, cannot) , and therefore XRDB API automatically converts these to _ (underscore) characters.

Prima defines its all set of resources, divided in two parts: general toolkit settings and per-widget settings. The general settings functionality is partially overloaded by command-line arguments. Per-widget settings are fonts and colors, definable for each Prima widget.

All of the general settings are applicable to the top-level item of widget hierarchy, named after the application, and `Prima` class. Some of these, though, are needed to be initialized before the application instance itself is created, so these can be accessed via `Prima` class only, for example, `Prima.Visual`. Some, on the contrary, may occasionally overlap with per-widget syntax. In particular, one must vary not to mix

```
Prima.font: some-font
```

with

```
Prima*font: some-font
```

The former syntax is a general setting, and sets the default Prima font. The latter is a per-widget assignment, and explicitly sets font to **all** Prima widgets, effectively ruining the toolkit font inheritance scheme. The same is valid for an even more oppressive

```
*font: some-font
```

record.

The allowed per-widget settings are colors and font settings only (see corresponding sections). It is an arguably useful feature to map all 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 enumeration of all properties for each widget takes place.

All global settings have identical class and name, varied in the case of the first letter. For example, to set `Submenudelay` value, one can do it either by

```
Prima.Submenudelay: 10
```

or

```
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, for all properties their names and class are given as `PropetyClass.propertyname` index.

Fonts

Default fonts

Prima::Application defines set of `get_default_XXX_font` functions, where each returns some user-selected font, to be displayed correspondingly in menu, message, window captions, all other widgets, and finally a default font. While in other OS'es these are indeed standard configurable user options, raw X11 doesn't define any. Nevertheless, as the high-level code relies on these, 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. Default font for 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 `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, the default `*-*-*-*-*-*-*-*-*-*-*-*-*-*` pattern, where each star character can be replaced by a particular font property, as name, size, charset, and so on. To interactively select an appropriate font, use standard `xfontsel` program from X11 distribution.

Note, that encoding part of the font is recommended to left unspecified, otherwise it may clash with LANG environment variable, which is used by Prima font subsystem to determine which font to select when no encoding is given. This advice, though, is correct only when both LANG and encoding part of a desired font match. In order to force a particular font encoding, the property `Prima.font` must contain one.

Alternatively, and/or to reduce X font traffic, one may set `IgnoreEncodings.ignoreEncodings` property, which is a semicolon- separated list of encodings Prima must not account. This feature has limited usability when for example fonts in Asian encodings result in large font requests. Another drastic measure to decrease font traffic is a boolean property `Noscaledfonts.noscaledfonts`, which, if set to 1, restricts the choice of fonts to the non-scalable fonts only.

Xft fonts

Recently, Prima was made to compile with Xft library, which contrary to core X font API, can make use of client-side fonts. Plus, Xft offers appealing features as font antialiasing, unicode, and arguably a better font syntax. The Xft font syntax is inherited from `fontconfig` library and to be consulted from `man fonts-conf`, but currently (November 2003) basic font descriptions can be composed as follows:

```
Palatino-12
```

A font with name `Palatino` and size 12.

```
Arial-10:BI
```

A font with name `Arial`, size 10, bold, italic. The `fontconfig` syntax allows more than that, for example, arbitrary matrix transformations, but Prima can make use only of font name, size, and style flags.

`--no-xft`

> `--no-xft` command-line argument, and boolean `UseXFT.usexft` XRDB property can be used to disable use of the Xft library.

`--no-core-fonts`

> Disables all X11 core fonts, except `fixed` fonts. The `fixed` font is selected for the same reasons that 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 a 'classic' X look.

`--font-priority`

> Can be set to either `xft` or `core`, to select a 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 Xft antialiasing.

Colors

XRDB conventions

X traditionally contains a 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 displays capabilities. Thus, it is customary to write

```
color: green
```

for many applications, and these in turn call `XParseColor` to convert strings into RGB values.

Prima is no exception to the scheme. Each widget can be assigned eight color properties that belongs to two classes by means of standard XRDB techniques:

class `Foreground`

Widget properties: `color`, `hiliteBackColor`, `disabledColor`, `dark3DColor`

class `Background`

Widget properties: `backColor`, `hiliteColor`, `disabledBackColor`, `light3DColor`

Additionally, set of command-line arguments allows overriding default values for these:

- `--fg` - color
- `--bg` - backColor
- `--hilite-fg` - hiliteColor
- `--hilite-bg` - hiliteBackColor
- `--disabled-fg` - disabledColor
- `--disabled-bg` - disabledBackColor
- `--light` - light3DColor
- `--dark` - dark3DColor

Visuals

X protocol works with explicitly defined pixel values only. A pixel value, maximum 32-bit value, represents a color in a display. There are two different color coding schemes - direct color and indexed color. The direct color-coded pixel value can unambiguously be converted into a RGB-value, without any external information. The indexed-color scheme represents pixel value as an index in a palette, which resided on X server. Depending on the color cell value of the palette, RGB color representation can be computed. A X display can contain more than one palette, and allow (or disallow) modification of palette color cells depending on a visual, the palette is attributed to.

A visual is a X server resource, containing representation of color coding scheme, color bit depth, and modificability of the palette. X server can (and usually does) provide more than one visual, as well as different bit depths. There are six classes of visuals in X paradigm. In each, Prima behaves differently, also depending on display bit depth available. In particular, color dithering can be used on displays with less than 12-bit color depth. On displays with modifiable color palette, Prima can install its own values in palettes, which may result in an effect known as display flashing. To switch to a non-default visual, use `Prima.Visual` XRDB property or `--visual` command-line argument. List of visuals can be produced interactively by standard `xdpyinfo` command from X distribution, where each class of visual corresponds to one of six visual 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 rarely met.

GrayScale

Contains modifiable color palette, and 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 a 16-color EGA mode. This visual is rarely met.

PseudoColor

All color cells are modifiable. Typically, 8-bit displays define this class for a default visual. For both *StaticColor* and *PseudoColor* visuals dithering is always used, although for `PseudoColor` Prima resorts to that only if X server cannot allocate another color.

On `PseudoColor` and `GrayScale` Prima allocates a small set of colors, not used in palette modifications. When a bitmap is to be exported via clipboard, or displayed in menu, or sent to a window manager as an icon to be displayed, it is downgraded to using these colors only, which are though guaranteedly to stay permanent through life of the application.

TrueColor

Each pixel value is explicitly coded as RGB. Typical example 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 support this visual, and usually by default it is not set. Prima supports this mode in exactly same way as *TrueColor* without additional features. During testing, it appeared that non-default *DirectColor* visuals require explicit assignment of each pixel used, which is inappropriate for color-rich images, and therefore Prima refuses to work on a non-default *DirectColor* visual.

Images

As described in the previous section, X does not standardize pixel memory format for *TrueColor* and `DirectColor` visuals, so there is a chance that Prima wouldn'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 shared memory image X extension, which speeds up image display for X servers and clients running on same machine. The price for this is that if 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 *BSD.

The clipboard exchange of images is incompletely implemented, since Prima does not accompany (and neither reads) COLORMAP, FOREGROUND, and BACKGROUND clipboard data, which contains pixel RGB values for a paletted image. As a palliative, the clipboard-bound images are downgraded to a safe set of colors, locked immutable either by X server or Prima core.

On 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.

Window managers

The original design of X protocol did not include the notion of a window manager, and latter is was implemented as an ad-hoc patch, which results in race conditions when configuring widgets. The extreme situation may well happen when even a non-top level widget may be influenced by a window manager, when for example a top-level widget was reparented into another widget, but the window manager is not aware or this yet.

The consequences of this, as well as programming guidances are described in `Prima::Window`. Here, we describe other aspects of interactions with WMs, as WM protocols, hints, and properties.

Prima was tested with alternating success under the following window managers: mwm, kwin, wmaker, fvwm, fvwm2, enlightment, sawfish, blackbox, 9wm, olvm, twm, and in no-WM environment.

Protocols

Prima makes use of `WM_DELETE_WINDOW` and `WM_TAKE_FOCUS` protocols. While `WM_DELETE_WINDOW` use is straightforward and needs no further attention, `WM_TAKE_FOCUS` can be tricky, since X defines several of input modes for a widget, which behave differently for each WM. In particular, 'focus follows pointer' gives pains under twm and mwm, where navigation of drop-down combo boxes is greatly hindered by 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 inter-client communication manual (ICCCM) allows, repeatedly misinterpreted by window managers. Many (wmaker, for example) apply the coordinates given from the program not to the top-level widget itself, but to its decoration. mwm defines list of accepted icon sizes so these can be absurdly high, which adds confusion to a client who can create icon of any size, but unable to determine the best one.

Non-standard properties

Prima tries to use WM-specific hints, known for two window managers: mwm and kwin. For mwm (Motif window manager) Prima sets hints of decoration border width and icons only. For kwin (and probably to others, who wish to conform to specifications of http://www.freedesktop.org/) Prima uses `NET_WM_STATE` property, in particular its maximization and task-bar visibility hints.

Use of these explicitly contradicts ICCCM, and definitely may lead to bugs in future (at least with `NET_WM_STATE`, since Motif interface can hardly expected to be changed). To disable the use of non-standard WM properties, `--icccm` command-line argument can be set.

Unicode

X does not support unicode, and number of patches were applied to X servers and clients to make the situation change. Currently (2003) standard unicode practices are not emerged yet, so Prima copes up with what (in author's opinion) is most promising: Xft and iconv libraries.

Fonts

X11 supports 8-bit and 16-bit text string display, and neither can be used effectively to display unicode strings. A `XCreateFontSet` technique, which combines several fonts under one descriptor, or a similarly implemented technique is the only way to provide correct unicode display.

Also, core font transfer protocol suffers from ineffective memory representation, which creates latency when fonts with large span of glyphs is loaded. Such fonts, in still uncommon though

standard iso10646 encoding, are the only media to display multi-encoding text without falling back to hacks similar to `XCreateFontSet`.

These, and some other problems are efficiently solved by 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 iconv charset convertor library.

Input

Prima does not support extended input methods (XIM etc), primarily because the authors are not acquainted with CIJK problem domain. Volunteers are welcome.

Clipboard

Prima supports UTF8 text in clipboard via `UTF8_STRING` transparently, although not by default.

```
Prima::Application-> wantUnicodeInput(1)
```

is the easiest (see the *Prima::Application* section) way to initiate UTF8 clipboard text exchange.

Due to the fact that any application can take ownership over the clipboard at any time, `open/close` brackets are not strictly respected in X11 implementation. Practically, this means that when modern X11 clipboard daemons (KDE klipper, for example) interfere with Prima clipboard, the results may not be consistent from the programmer's view, for example, clipboard contains data after `clear` call, and the like. It must be noted though that this behavior is expected by the users.

Other XRDB resources

Timeouts

Raw X11 provides no such GUI helpers as double-click event, cursor, or menu. Neither does it provide the related time how often, for example, a cursor would blink. Therefore Prima emulates these, but allows the user to reprogram the corresponding timeouts. Prima recognizes the following properties, accessible either via application name or Prima class key. All timeouts are integer values, representing number of milliseconds for the corresponding timeout property.

Blinkinvisibletime.blinkinvisibletime: MSEC

Cursor stays invisible MSEC milliseconds.

Default value: 500

Blinkvisibletime.blinkvisibletime: MSEC

Cursor stays visible MSEC milliseconds.

Default value: 500

Clicktimeframe.clicktimeframe MSEC

If 'mouse down' and 'mouse up' events are follow in MSEC, 'mouse click' event is synthesized.

Default value: 200

Doubleclicktimeframe.doubleclicktimeframe MSEC

If 'mouse click' and 'mouse down' events are follow in MSEC, 'mouse double click' event is synthesized.

Default value: 200

Submenudelay.submenudelay MSEC

When the used clicks on a menu item, which 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 - 'first' and 'next' delay. These actions are not simulated within Prima core, and the corresponding timeouts are merely advisable to the programmer. Prima widgets use it for automatic scrolling, either by a scrollbar or by any other means. Also, `Prima::Button` in `autoRepeat` mode uses these timeouts for emulation of a key press.

`Scrollfirst` is a 'first' timeout.

Default value: 200

Scrollnext.scrollnext MSEC

A timeout used for same reasons as `Scrollfirst`, but after it is expired.

Default value: 50

Miscellaneous

Visual.visual: VISUAL_ID

Selects display visual by VISUAL_ID, which is usually has a form of `0x??`. Various visuals provide different color depth and access scheme. Some X stations have badly chosen default visuals (for example, default IRIX workstation setup has 8-bit default visual selected), so this property can be used to fix things. List of visuals, supported by a X display can be produced interactively by standard `xdpyinfo` command from X distribution.

Identical to `--visual` command-line argument.

See the *Color* entry for more information.

Wheeldown.wheeldown BUTTON

BUTTON is a number of X mouse button event, treated as '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 a number of X mouse button event, treated as 'mouse wheel up' event.

Default value: 4

Debugging

The famous 'use the source' call is highly actual with Prima. However, some debug information comes compiled in, and can be activated by `--debug` command-line key. Combination of letters to the key activates debug printouts 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 X protocol synchronization (at expense of operation slowdown though) is activated with `--sync` command-line argument, and can be used to ease the debugging.