NAME

Cairo − Perl interface to the cairo 2d vector graphics library

SYNOPSIS

```
use Cairo;

my $surface = Cairo::ImageSurface->create ('argb32', 100, 100);
my $cr = Cairo::Context->create ($surface);

$cr->rectangle (10, 10, 40, 40);
$cr->set_source_rgb (0, 0, 0);
$cr->fill;

$cr->rectangle (50, 50, 40, 40);
$cr->set_source_rgb (1, 1, 1);
$cr->fill;

$cr->show_page;

$surface->write_to_png ('output.png');
```

ABSTRACT

Cairo provides Perl bindings for the vector graphics library cairo. It supports multiple output targets, including PNG, PDF and SVG. Cairo produces identical output on all those targets.

API DOCUMENTATION

This is a listing of the API Cairo provides. For more verbose information, refer to the cairo manual at <http://cairographics.org/manual/>.

Drawing

Cairo::Context — The cairo drawing context

Cairo::Context is the main object used when drawing with Cairo. To draw with Cairo, you create a *Cairo::Context*, set the target surface, and drawing options for the *Cairo::Context*, create shapes with methods like `$cr->move_to` and `$cr->line_to`, and then draw shapes with `$cr->stroke` or `$cr->fill`.

Cairo::Context's can be pushed to a stack via `$cr->save`. They may then safely be changed, without loosing the current state. Use `$cr->restore` to restore to the saved state.

`$cr` = Cairo::Context->create ($surface)
    `$surface:` *Cairo::Surface*
`$cr`->save
`$cr`->restore
`$status` = `$cr`->status
`$surface` = `$cr`->get_target
`$cr`->push_group [1.2]
`$cr`->push_group_with_content ($content) [1.2]
    `$content:` *Cairo::Content*
`$pattern` = `$cr`->pop_group [1.2]
`$cr`->pop_group_to_source [1.2]
`$surface` = `$cr`->get_group_target [1.2]
`$cr`->set_source_rgb ($red, `$green`, `$blue`)
    `$red:` double
    `$green:` double
    `$blue:` double
`$cr`->set_source_rgba ($red, `$green`, `$blue`, `$alpha`)

    $red: double
    $green: double
    $blue: double
    $alpha: double
$cr->set_source ($source)
    $source: *Cairo::Pattern*
$cr->set_source_surface ($surface, $x, $y)
    $surface: *Cairo::Surface*
    $x: double
    $y: double
$source = $cr->get_source
$cr->set_antialias ($antialias)
    $antialias: *Cairo::Antialias*
$antialias = $cr->get_antialias
$cr->set_dash ($offset, ...)
    $offset: double
    ...: list of doubles
$cr->set_fill_rule ($fill_rule)
    $fill_rule: *Cairo::FillRule*
$fill_rule = $cr->get_fill_rule
$cr->set_line_cap ($line_cap)
    $line_cap: *Cairo::LineCap*
$line_cap = $cr->get_line_cap
$cr->set_line_join ($line_join)
    $line_join: *Cairo::LineJoin*
$line_join = $cr->get_line_join
$cr->set_line_width ($width)
    $width: double
$width = $cr->get_line_width
$cr->set_miter_limit ($limit)
    $limit: double
($offset, @dashes) = $cr->get_dash [1.4]
$limit = $cr->get_miter_limit
$cr->set_operator ($op)
    $op: *Cairo::Operator*
$op = $cr->get_operator
$cr->set_tolerance ($tolerance)
    $tolerance: double
$tolerance = $cr->get_tolerance
$cr->clip
$cr->clip_preserve
($x1, $y1, $x2, $y2) = $cr->clip_extents [1.4]
$bool = $cr->in_clip ($x, $y) [1.10]
    $x: double
    $y: double
@rectangles = $cr->copy_clip_rectangle_list [1.4]
$cr->reset_clip
$cr->fill
$cr->fill_preserve
($x1, $y1, $x2, $y2) = $cr->fill_extents
$bool = $cr->in_fill ($x, $y)
    $x: double

    $y: double
$cr->mask ($pattern)
    $pattern: *Cairo::Pattern*
$cr->mask_surface ($surface, $surface_x, $surface_y)
    $surface: *Cairo::Surface*
    $surface_x: double
    $surface_y: double
$cr->paint
$cr->paint_with_alpha ($alpha)
    $alpha: double
$cr->stroke
$cr->stroke_preserve
($x1, $y1, $x2, $y2) = $cr->stroke_extents
$bool = $cr->in_stroke ($x, $y)
    $x: double
    $y: double
$cr->tag_begin($name, $atts) [1.16]
    $name: string
    $atts: string
$cr->tag_end($name) [1.16]
    $name: string
Predefined names:
    Cairo::TAG_DEST [1.16]
    Cairo::TAG_LINK [1.16]
$cr->copy_page
$cr->show_page

Paths — Creating paths and manipulating path data

```
  $path = [
    { type => "move-to", points => [[1, 2]] },
    { type => "line-to", points => [[3, 4]] },
    { type => "curve-to", points => [[5, 6], [7, 8], [9, 10]] },
    ...
    { type => "close-path", points => [] },
  ];
```

Cairo::Path is a data structure for holding a path. This data structure serves as the return value for `$cr->copy_path` and `$cr->copy_path_flat` as well the input value for `$cr->append_path`.

Cairo::Path is represented as an array reference that contains path elements, represented by hash references with two keys: *type* and *points*. The value for *type* can be either of the following:

```
move-to
line-to
curve-to
close-path
```

The value for *points* is an array reference which contains zero or more points. Points are represented as array references that contain two doubles: *x* and *y*. The necessary number of points depends on the*type* of the path element:

`move-to`: 1 point
`line_to`: 1 point
`curve-to`: 3 points
`close-path`: 0 points

The semantics and ordering of the coordinate values are consistent with `$cr->move_to`, `$cr->line_to`, `$cr->curve_to`, and `$cr->close_path`.

Note that the paths returned by Cairo are implemented as tied array references which do **not** support adding, removing or shuffling of path segments. For these operations, you need to make a shallow copy first:

```
my @path_clone = @{$path};
# now you can alter @path_clone which ever way you want
```

The points of a single path element can be changed directly, however, without the need for a shallow copy:

```
$path->[$i]{points} = [[3, 4], [5, 6], [7, 8]];
```

`$path` = `$cr`->copy_path
`$path` = `$cr`->copy_path_flat
`$cr`->append_path ($path)
    `$path`: *Cairo::Path*
`$bool` = `$cr`->has_current_point [1.6]
($x, `$y`) = `$cr`->get_current_point
`$cr`->new_path
`$cr`->new_sub_path [1.2]
`$cr`->close_path
($x1, `$y1`, `$x2`, `$y2`) = `$cr`->path_extents [1.6]
`$cr`->arc ($xc, `$yc`, `$radius`, `$angle1`, `$angle2`)
    `$xc`: double
    `$yc`: double
    `$radius`: double
    `$angle1`: double
    `$angle2`: double
`$cr`->arc_negative ($xc, `$yc`, `$radius`, `$angle1`, `$angle2`)
    `$xc`: double
    `$yc`: double
    `$radius`: double
    `$angle1`: double
    `$angle2`: double
`$cr`->curve_to ($x1, `$y1`, `$x2`, `$y2`, `$x3`, `$y3`)
    `$x1`: double
    `$y1`: double
    `$x2`: double
    `$y2`: double
    `$x3`: double
    `$y3`: double
`$cr`->line_to ($x, `$y`)
    `$x`: double
    `$y`: double
`$cr`->move_to ($x, `$y`)
    `$x`: double
    `$y`: double
`$cr`->rectangle ($x, `$y`, `$width`, `$height`)
    `$x`: double
    `$y`: double
    `$width`: double
    `$height`: double
`$cr`->glyph_path (...)
    ...: list of *Cairo::Glyph*'s
`$cr`->text_path ($utf8)
    `$utf8`: string in utf8 encoding

$cr->rel_curve_to ($dx1, $dy1, $dx2, $dy2, $dx3, $dy3)
    $dx1: double
    $dy1: double
    $dx2: double
    $dy2: double
    $dx3: double
    $dy3: double
$cr->rel_line_to ($dx, $dy)
    $dx: double
    $dy: double
$cr->rel_move_to ($dx, $dy)
    $dx: double
    $dy: double

Patterns — Gradients and filtered sources

$status = $pattern->status
$type = $pattern->get_type [1.2]
$pattern->set_extend ($extend)
    $extend: *Cairo::Extend*
$extend = $pattern->get_extend
$pattern->set_filter ($filter)
    $filter: *Cairo::Filter*
$filter = $pattern->get_filter
$pattern->set_matrix ($matrix)
    $matrix: *Cairo::Matrix*
$matrix = $pattern->get_matrix
$pattern = Cairo::SolidPattern->create_rgb ($red, $green, $blue)
    $red: double
    $green: double
    $blue: double
$pattern = Cairo::SolidPattern->create_rgba ($red, $green, $blue, $alpha)
    $red: double
    $green: double
    $blue: double
    $alpha: double
($r, $g, $b, $a) = $pattern->get_rgba [1.4]
$pattern = Cairo::SurfacePattern->create ($surface)
    $surface: *Cairo::Surface*
$surface = $pattern->get_surface [1.4]
$pattern = Cairo::LinearGradient->create ($x0, $y0, $x1, $y1)
    $x0: double
    $y0: double
    $x1: double
    $y1: double
($x0, $y0, $x1, $y1) = $pattern->get_points [1.4]
$pattern = Cairo::RadialGradient->create ($cx0, $cy0, $radius0, $cx1, $cy1, $radius1)
    $cx0: double
    $cy0: double
    $radius0: double
    $cx1: double
    $cy1: double
    $radius1: double
($x0, $y0, $r0, $x1, $y1, $r1) = $pattern->get_circles [1.4]

`$pattern`->add_color_stop_rgb ($offset, `$red`, `$green`, `$blue`)
    `$offset`: double
    `$red`: double
    `$green`: double
    `$blue`: double
`$pattern`->add_color_stop_rgba ($offset, `$red`, `$green`, `$blue`, `$alpha`)
    `$offset`: double
    `$red`: double
    `$green`: double
    `$blue`: double
    `$alpha`: double
`@stops` = `$pattern`->get_color_stops [1.4]
    A color stop is represented as an array reference with five elements: offset, red, green, blue, and alpha.

Regions — Representing a pixel-aligned area

`$region` = Cairo::Region->create (...) [1.10]
    ...: zero or more *Cairo::RectangleInt*
`$status` = `$region`->status [1.10]
`$num` = `$region`->num_rectangles [1.10]
`$rect` = `$region`->get_rectangle ($i) [1.10]
    `$i`: integer
`$bool` = `$region`->is_empty [1.10]
`$bool` = `$region`->contains_point ($x, `$y`) [1.10]
    `$x`: integer
    `$y`: integer
`$bool` = `$region_one`->equal ($region_two) [1.10]
    `$region_two`: *Cairo::Region*
`$region`->translate ($dx, `$dy`) [1.10]
    `$dx`: integer
    `$dy`: integer
`$status` = `$dst`->intersect ($other) [1.10]
`$status` = `$dst`->intersect_rectangle ($rect) [1.10]
`$status` = `$dst`->subtract ($other) [1.10]
`$status` = `$dst`->subtract_rectangle ($rect) [1.10]
`$status` = `$dst`->union ($other) [1.10]
`$status` = `$dst`->union_rectangle ($rect) [1.10]
`$status` = `$dst`->xor ($other) [1.10]
`$status` = `$dst`->xor_rectangle ($rect) [1.10]
    `$other`: *Cairo::Region*
    `$rect`: *Cairo::RectangleInt*

Transformations — Manipulating the current transformation matrix

`$cr`->translate ($tx, `$ty`)
    `$tx`: double
    `$ty`: double
`$cr`->scale ($sx, `$sy`)
    `$sx`: double
    `$sy`: double
`$cr`->rotate ($angle)
    `$angle`: double
`$cr`->transform ($matrix)
    `$matrix`: *Cairo::Matrix*
`$cr`->set_matrix ($matrix)

    $matrix: *Cairo::Matrix*
$matrix = $cr->get_matrix
$cr->identity_matrix
($x, $y) = $cr->user_to_device ($x, $y)
    $x: double
    $y: double
($dx, $dy) = $cr->user_to_device_distance ($dx, $dy)
    $dx: double
    $dy: double
($x, $y) = $cr->device_to_user ($x, $y)
    $x: double
    $y: double
($dx, $dy) = $cr->device_to_user_distance ($dx, $dy)
    $dx: double
    $dy: double

Text — Rendering text and sets of glyphs

Glyphs are represented as anonymous hash references with three keys: *index*, *x* and *y*. Example:

```
  my @glyphs = ({ index => 1, x => 2, y => 3 },
                { index => 2, x => 3, y => 4 },
                { index => 3, x => 4, y => 5 });
```

$cr->select_font_face ($family, $slant, $weight)
    $family: string
    $slant: *Cairo::FontSlant*
    $weight: *Cairo::FontWeight*
$cr->set_font_size ($size)
    $size: double
$cr->set_font_matrix ($matrix)
    $matrix: *Cairo::Matrix*
$matrix = $cr->get_font_matrix
$cr->set_font_options ($options)
    $options: *Cairo::FontOptions*
$options = $cr->get_font_options
$cr->set_scaled_font ($scaled_font) [1.2]
    $scaled_font: *Cairo::ScaledFont*
$scaled_font = $cr->get_scaled_font [1.4]
$cr->show_text ($utf8)
    $utf8: string
$cr->show_glyphs (...)
    ...: list of glyphs
$cr->show_text_glyphs ($utf8, $glyphs, $clusters, $cluster_flags) [1.8]
    $utf8: string
    $glyphs: array ref of glyphs
    $clusters: array ref of clusters
    $cluster_flags: *Cairo::TextClusterFlags*
$face = $cr->get_font_face
$extents = $cr->font_extents
$cr->set_font_face ($font_face)
    $font_face: *Cairo::FontFace*
$cr->set_scaled_font ($scaled_font)
    $scaled_font: *Cairo::ScaledFont*
$extents = $cr->text_extents ($utf8)

$utf8: string
$extents = $cr->glyph_extents (...)
...: list of glyphs
$face = Cairo::ToyFontFace->create ($family, $slant, $weight) [1.8]
$family: string
$slant: *Cairo::FontSlant*
$weight: *Cairo::FontWeight*
$family = $face->get_family [1.8]
$slang = $face->get_slant [1.8]
$weight = $face->get_weight [1.8]

Fonts

Cairo::FontFace — Base class for fonts

$status = $font_face->status
$type = $font_face->get_type [1.2]

Scaled Fonts — Caching metrics for a particular font size

$scaled_font = Cairo::ScaledFont->create ($font_face, $font_matrix, $ctm, $options)
$font_face: *Cairo::FontFace*
$font_matrix: *Cairo::Matrix*
$ctm: *Cairo::Matrix*
$options: *Cairo::FontOptions*
$status = $scaled_font->status
$extents = $scaled_font->extents
$extents = $scaled_font->text_extents ($utf8) [1.2]
$utf8: string
$extents = $scaled_font->glyph_extents (...)
...: list of glyphs
($status, $glyphs, $clusters, $cluster_flags) = $scaled_font->text_to_glyphs ($x, $y, $utf8) [1.8]
$x: double
$y: double
$utf8: string
$font_face = $scaled_font->get_font_face [1.2]
$options = $scaled_font->get_font_options [1.2]
$font_matrix = $scaled_font->get_font_matrix [1.2]
$ctm = $scaled_font->get_ctm [1.2]
$scale_matrix = $scaled_font->get_scale_matrix [1.8]
$type = $scaled_font->get_type [1.2]

Font Options — How a font should be rendered

$font_options = Cairo::FontOptions->create
$status = $font_options->status
$font_options->merge ($other)
$other: *Cairo::FontOptions*
$hash = $font_options->hash
$bools = $font_options->equal ($other)
$other: *Cairo::FontOptions*
$font_options->set_antialias ($antialias)
$antialias: *Cairo::AntiAlias*
$antialias = $font_options->get_antialias
$font_options->set_subpixel_order ($subpixel_order)
$subpixel_order: *Cairo::SubpixelOrder*

$subpixel_order = $font_options->get_subpixel_order
$font_options->set_hint_style ($hint_style)
    $hint_style: *Cairo::HintStyle*
$hint_style = $font_options->get_hint_style
$font_options->set_hint_metrics ($hint_metrics)
    $hint_metrics: *Cairo::HintMetrics*
$hint_metrics = $font_options->get_hint_metrics

FreeType Fonts — Font support for FreeType

If your cairo library supports it, the FreeType integration allows you to load font faces from font files. You can query for this capability with `Cairo::HAS_FT_FONT`. To actually use this, you'll need the Font::FreeType module.

my $face = Cairo::FtFontFace->create ($ft_face, $load_flags=0)
    $ft_face: *Font::FreeType::Face*
    $load_flags: integer

    This method allows you to create a *Cairo::FontFace* from a *Font::FreeType::Face*. To obtain the latter, you can for example load it from a file:

```
my $file = '/usr/share/fonts/truetype/ttf-bitstream-vera/Vera.ttf';
my $ft_face = Font::FreeType->new->face ($file);
my $face = Cairo::FtFontFace->create ($ft_face);
```

Surfaces

Cairo::Surface — Base class for surfaces

$similar = Cairo::Surface->create_similar ($other, $content, $width, $height)
    $other: *Cairo::Surface*
    $content: *Cairo::Content*
    $width: integer
    $height: integer

    For hysterical reasons, you can also use the following syntax:

```
$similar = $other->create_similar ($content, $width, $height)
```

$new = Cairo::Surface->create_for_rectangle ($target, $x, $y, $width, $height) [1.10]
    $target: *Cairo::Surface*
    $x: double
    $y: double
    $width: double
    $height: double
$status = $surface->status
$surface->finish
$surface->flush
$font_options = $surface->get_font_options
$content = $surface->get_content [1.2]
$surface->mark_dirty
$surface->mark_dirty_rectangle ($x, $y, $width, $height)
    $x: integer
    $y: integer
    $width: integer
    $height: integer
$surface->set_device_offset ($x_offset, $y_offset)
    $x_offset: integer
    $y_offset: integer

($x_offset, $y_offset) = $surface->get_device_offset [1.2]
$surface->set_fallback_resolution ($x_pixels_per_inch, $y_pixels_per_inch) [1.2]
$x_pixels_per_inch: double
$y_pixels_per_inch: double
($x_pixels_per_inch, $y_pixels_per_inch) = $surface->get_fallback_resolution [1.8]
$type = $surface->get_type [1.2]
$surface->set_mime_data ($mime_type, $mime_data) [1.10]
$mime_data = $surface->get_mime_data ($mime_type) [1.10]
$bool = $surface->supports_mime_type ($mime_type) [1.12]
$mime_type: string
Predefined MIME types:
Cairo::Surface::MIME_TYPE_JP2 [1.10]
Cairo::Surface::MIME_TYPE_JPEG [1.10]
Cairo::Surface::MIME_TYPE_PNG [1.10]
Cairo::Surface::MIME_TYPE_URI [1.10]
Cairo::Surface::MIME_TYPE_UNIQUE_ID [1.12]
Cairo::Surface::MIME_TYPE_JBIG2 [1.14]
Cairo::Surface::MIME_TYPE_JBIG2_GLOBAL [1.14]
Cairo::Surface::MIME_TYPE_JBIG2_GLOBAL_PARAMS [1.14]
Cairo::Surface::MIME_TYPE_CCITT_FAX [1.16]
Cairo::Surface::MIME_TYPE_CCITT_FAX_PARAMS [1.16]
Cairo::Surface::MIME_TYPE_EPS [1.16]
Cairo::Surface::MIME_TYPE_EPS_PARAMS [1.16]
$mime_data: binary data string
$status = $surface->copy_page [1.6]
$status: *Cairo::Status*
$status = $surface->show_page [1.6]
$status: *Cairo::Status*
$boolean = $surface->has_show_text_glyphs [1.8]

Image Surfaces — Rendering to memory buffers

$surface = Cairo::ImageSurface->create ($format, $width, $height)
$format: *Cairo::Format*
$width: integer
$height: integer
$surface = Cairo::ImageSurface->create_for_data ($data, $format, $width, $height, $stride)
$data: image data
$format: *Cairo::Format*
$width: integer
$height: integer
$stride: integer
$data = $surface->get_data [1.2]
$format = $surface->get_format [1.2]
$width = $surface->get_width
$height = $surface->get_height
$stride = $surface->get_stride [1.2]
$stride = Cairo::Format::stride_for_width ($format, $width) [1.6]
$format: *Cairo::Format*
$width: integer

PDF Surfaces — Rendering PDF documents

$surface = Cairo::PdfSurface->create ($filename, $width_in_points, $height_in_points) [1.2]

$filename: string
$width_in_points: double
$height_in_points: double
$surface = Cairo::PdfSurface->create_for_stream ($callback, $callback_data, $width_in_points, $height_in_points) [1.2]
$callback: *Cairo::WriteFunc*
$callback_data: scalar
$width_in_points: double
$height_in_points: double
$surface->set_size ($width_in_points, $height_in_points) [1.2]
$width_in_points: double
$height_in_points: double
$surface->restrict_to_version ($version) [1.10]
$version: *Cairo::PdfVersion*
@versions = Cairo::PdfSurface::get_versions [1.10]
$string = Cairo::PdfSurface::version_to_string ($version) [1.10]
$version: *Cairo::PdfVersion*
$item_id = $surface->add_outline($parent_id, $name, $attributes, $flags) [1.16]
$item_id: int, item ID
$parent_id: parent item id or Cairo::PdfSurface::OUTLINE_ROOT
$name: string, item display
$attributes: string, item attributes
$flags: list reference, item flags
$surface->set_metadata($name, $value) [1.16]
$name: string
$value: string
$surface->set_page_label($label) [1.16]
$label: string, page label
$surface->set_thumbnail_size($width, $height) [1.16]
$width: int, thumbnail width
$height: int, thumbnail height

PNG Support — Reading and writing PNG images

$surface = Cairo::ImageSurface->create_from_png ($filename)
$filename: string
Cairo::ReadFunc: $data = sub { my ($callback_data, $length) = @_; }
$data: binary image data, of length $length
$callback_data: scalar, user data
$length: integer, bytes to read
$surface = Cairo::ImageSurface->create_from_png_stream ($callback, $callback_data)
$callback: *Cairo::ReadFunc*
$callback_data: scalar
$status = $surface->write_to_png ($filename)
$filename: string
Cairo::WriteFunc: sub { my ($callback_data, $data) = @_; }
$callback_data: scalar, user data
$data: binary image data, to be written
$status = $surface->write_to_png_stream ($callback, $callback_data)
$callback: *Cairo::WriteFunc*
$callback_data: scalar

PostScript Surfaces — Rendering PostScript documents

$surface = Cairo::PsSurface->create ($filename, $width_in_points, $height_in_points) [1.2]

$filename: string
$width_in_points: double
$height_in_points: double
$surface = Cairo::PsSurface->create_for_stream ($callback, $callback_data, $width_in_points, $height_in_points) [1.2]
$callback: *Cairo::WriteFunc*
$callback_data: scalar
$width_in_points: double
$height_in_points: double
$surface->set_size ($width_in_points, $height_in_points) [1.2]
$width_in_points: double
$height_in_points: double
$surface->dsc_begin_setup [1.2]
$surface->dsc_begin_page_setup [1.2]
$surface->dsc_comment ($comment) [1.2]
$comment: string
$surface->restrict_to_level ($level) [1.6]
$level: *Cairo::PsLevel*
@levels = Cairo::PsSurface::get_levels [1.6]
$string = Cairo::PsSurface::level_to_string ($level) [1.6]
$level: *Cairo::PsLevel*
$surface->set_eps ($eps) [1.6]
$eps: boolean
$eps = $surface->get_eps [1.6]

Recording Surfaces — Records all drawing operations

$surface = Cairo::RecordingSurface->create ($content, $extents) [1.10]
$content: *Cairo::Content*
$extents: *Cairo::Rectangle*
($x0, $y0, $width, $height) = $surface->ink_extents [1.10]
$extents_ref = $surface->get_extents [1.12]
$extents_ref: *Cairo::Rectangle* reference

SVG Surfaces — Rendering SVG documents

$surface = Cairo::SvgSurface->create ($filename, $width_in_points, $height_in_points) [1.2]
$filename: string
$width_in_points: double
$height_in_points: double
$surface = Cairo::SvgSurface->create_for_stream ($callback, $callback_data, $width_in_points, $height_in_points) [1.2]
$callback: *Cairo::WriteFunc*
$callback_data: scalar
$width_in_points: double
$height_in_points: double
$surface->restrict_to_version ($version) [1.2]
$version: *Cairo::SvgVersion*
@versions = Cairo::SvgSurface::get_versions [1.2]
$string = Cairo::SvgSurface::version_to_string ($version) [1.2]
$version: *Cairo::SvgVersion*

Utilities

Version Information — Run-time and compile-time version checks.

`$version_code` = Cairo->lib_version
`$version_string` = Cairo->lib_version_string
These two functions return the version of libcairo that the program is currently running against.

`$version_code` = Cairo->LIB_VERSION
Returns the version of libcairo that Cairo was compiled against.

`$version_code` = Cairo->LIB_VERSION_ENCODE ($major, `$minor`, `$micro`)
`$major`: integer
`$minor`: integer
`$micro`: integer

Encodes the version `$major.$minor.$micro` as an integer suitable for comparison against `Cairo->lib_version` and `Cairo->LIB_VERSION`.

SEE ALSO

<http://cairographics.org/documentation>
Lists many available resources including tutorials and examples

<http://cairographics.org/manual/>
Contains the reference manual

AUTHORS

Ross McFarland <rwmcfa1 at neces dot com>
Torsten Schoenfeld <kaffeetisch at gmx dot de>

COPYRIGHT