

Guile-Cairo

version 1.4, updated 17 July 2007

Carl Worth
Andy Wingo
(many others)

This manual is for Guile-Cairo (version 1.4, updated 17 July 2007)

Copyright 2002-2007 Carl Worth and others

Permission is granted to copy, distribute and/or modify this document under the terms of either the GNU Lesser General Public License (LGPL) version 2.1 or the Mozilla Public License (MPL) version 1.1.

Short Contents

1	cairo_t	1
2	Paths	14
3	Patterns	21
4	Transformations	28
5	Text	30
6	cairo_font_face_t	34
7	Scaled Fonts	35
8	Font Options	38
9	FreeType Fonts	40
10	Win32 Fonts	41
11	cairo_surface_t	42
12	Image Surfaces	45
13	PDF Surfaces	47
14	PNG Support	48
15	PostScript Surfaces	49
16	SVG Surfaces	52
17	cairo_matrix_t	53
18	Error handling	55
19	Version Information	56
20	Types	58
	Concept Index	59
	Function Index	60

1 cairo_t

The cairo drawing context

1.1 Overview

`<cairo-t>` is the main object used when drawing with cairo. To draw with cairo, you create a `<cairo-t>`, set the target surface, and drawing options for the `<cairo-t>`, create shapes with functions like `cairo-move-to` and `cairo-line-to`, and then draw shapes with `cairo-stroke` or `cairo-fill`.

`<cairo-t>`'s can be pushed to a stack via `cairo-save`. They may then safely be changed, without losing the current state. Use `cairo-restore` to restore to the saved state.

1.2 Usage

`cairo-create (target <cairo-surface-t>) ⇒ (ret <cairo-t>)` [Function]

Creates a new `<cairo-t>` with all graphics state parameters set to default values and with *target* as a target surface. The target surface should be constructed with a backend-specific function such as `cairo-image-surface-create` (or any other `'cairo_<backend>_surface_create'` variant).

target target surface for the context
ret a newly allocated `<cairo-t>`.

`cairo-save (cr <cairo-t>)` [Function]

Makes a copy of the current state of *cr* and saves it on an internal stack of saved states for *cr*. When `cairo-restore` is called, *cr* will be restored to the saved state. Multiple calls to `cairo-save` and `cairo-restore` can be nested; each call to `cairo-restore` restores the state from the matching paired `cairo-save`.

It isn't necessary to clear all saved states before a `<cairo-t>` is freed. If the reference count of a `<cairo-t>` drops to zero in response to a call to `cairo-destroy`, any saved states will be freed along with the `<cairo-t>`.

cr a `<cairo-t>`

`cairo-restore (cr <cairo-t>)` [Function]

Restores *cr* to the state saved by a preceding call to `cairo-save` and removes that state from the stack of saved states.

cr a `<cairo-t>`

`cairo-get-target (cr <cairo-t>) ⇒ (ret <cairo-surface-t>)` [Function]

Gets the target surface for the cairo context as passed to `cairo-create`.

This function will always return a valid pointer, but the result can be a "nil" surface if *cr* is already in an error state, (ie. `cairo-status'!='CAIRO_STATUS_SUCCESS'`). A nil surface is indicated by `cairo-surface-status'!='CAIRO_STATUS_SUCCESS'`.

cr a cairo context
ret the target surface. This object is owned by cairo. To keep a reference to it, you must call `cairo-surface-reference`.

cairo-push-group (*cr* <cairo-t>) [Function]

Temporarily redirects drawing to an intermediate surface known as a group. The redirection lasts until the group is completed by a call to **cairo-pop-group** or **cairo-pop-group-to-source**. These calls provide the result of any drawing to the group as a pattern, (either as an explicit object, or set as the source pattern).

This group functionality can be convenient for performing intermediate compositing. One common use of a group is to render objects as opaque within the group, (so that they occlude each other), and then blend the result with translucence onto the destination.

Groups can be nested arbitrarily deep by making balanced calls to **cairo-push-group/cairo-pop-group**. Each call pushes/pops the new target group onto/from a stack.

The **cairo-push-group** function calls **cairo-save** so that any changes to the graphics state will not be visible outside the group, (the **pop_group** functions call **cairo-restore**).

By default the intermediate group will have a content type of **CAIRO_CONTENT_COLOR_ALPHA**. Other content types can be chosen for the group by using **cairo-push-group-with-content** instead.

As an example, here is how one might fill and stroke a path with translucence, but without any portion of the fill being visible under the stroke:

```
cairo_push_group (cr);
cairo_set_source (cr, fill_pattern);
cairo_fill_preserve (cr);
cairo_set_source (cr, stroke_pattern);
cairo_stroke (cr);
cairo_pop_group_to_source (cr);
cairo_paint_with_alpha (cr, alpha);
```

cr a cairo context

Since 1.2

cairo-pop-group (*cr* <cairo-t>) ⇒ (*ret* <cairo-pattern-t>) [Function]

Terminates the redirection begun by a call to **cairo-push-group** or **cairo-push-group-with-content** and returns a new pattern containing the results of all drawing operations performed to the group.

The **cairo-pop-group** function calls **cairo-restore**, (balancing a call to **cairo-save** by the **push_group** function), so that any changes to the graphics state will not be visible outside the group.

cr a cairo context

ret a newly created (surface) pattern containing the results of all drawing operations performed to the group. The caller owns the returned object and should call **cairo-pattern-destroy** when finished with it.

Since 1.2

`cairo-pop-group-to-source` (*cr* <cairo-t>) [Function]

Terminates the redirection begun by a call to `cairo-push-group` or `cairo-push-group-with-content` and installs the resulting pattern as the source pattern in the given cairo context.

The behavior of this function is equivalent to the sequence of operations:

```
cairo_pattern_t *group = cairo_pop_group (cr);
cairo_set_source (cr, group);
cairo_pattern_destroy (group);
```

but is more convenient as there is no need for a variable to store the short-lived pointer to the pattern.

The `cairo-pop-group` function calls `cairo-restore`, (balancing a call to `cairo-save` by the `push_group` function), so that any changes to the graphics state will not be visible outside the group.

cr a cairo context

Since 1.2

`cairo-get-group-target` (*cr* <cairo-t>) [Function]

⇒ (*ret* <cairo-surface-t>)

Gets the target surface for the current group as started by the most recent call to `cairo-push-group` or `cairo-push-group-with-content`.

This function will return NULL if called "outside" of any group rendering blocks, (that is, after the last balancing call to `cairo-pop-group` or `cairo-pop-group-to-source`).

cr a cairo context

ret the target group surface, or NULL if none. This object is owned by cairo. To keep a reference to it, you must call `cairo-surface-reference`.

Since 1.2

`cairo-set-source-rgb` (*cr* <cairo-t>) (*red* <double>) [Function]

(*green* <double>) (*blue* <double>)

Sets the source pattern within *cr* to an opaque color. This opaque color will then be used for any subsequent drawing operation until a new source pattern is set.

The color components are floating point numbers in the range 0 to 1. If the values passed in are outside that range, they will be clamped.

cr a cairo context

red red component of color

green green component of color

blue blue component of color

`cairo-set-source-rgba` (*cr* <cairo-t>) (*red* <double>) [Function]
 (*green* <double>) (*blue* <double>) (*alpha* <double>)

Sets the source pattern within *cr* to a translucent color. This color will then be used for any subsequent drawing operation until a new source pattern is set.

The color and alpha components are floating point numbers in the range 0 to 1. If the values passed in are outside that range, they will be clamped.

cr a cairo context
red red component of color
green green component of color
blue blue component of color
alpha alpha component of color

`cairo-set-source` (*cr* <cairo-t>) (*source* <cairo-pattern-t>) [Function]

Sets the source pattern within *cr* to *source*. This pattern will then be used for any subsequent drawing operation until a new source pattern is set.

Note: The pattern's transformation matrix will be locked to the user space in effect at the time of `cairo-set-source`. This means that further modifications of the current transformation matrix will not affect the source pattern. See `cairo-pattern-set-matrix`.

XXX: I'd also like to direct the reader's attention to some (not-yet-written) section on cairo's imaging model. How would I do that if such a section existed? (cworth).

cr a cairo context
source a <cairo-pattern-t> to be used as the source for subsequent drawing operations.

`cairo-set-source-surface` (*cr* <cairo-t>) [Function]
 (*surface* <cairo-surface-t>) (*x* <double>) (*y* <double>)

This is a convenience function for creating a pattern from *surface* and setting it as the source in *cr* with `cairo-set-source`.

The *x* and *y* parameters give the user-space coordinate at which the surface origin should appear. (The surface origin is its upper-left corner before any transformation has been applied.) The *x* and *y* patterns are negated and then set as translation values in the pattern matrix.

Other than the initial translation pattern matrix, as described above, all other pattern attributes, (such as its extend mode), are set to the default values as in `cairo-pattern-create-for-surface`. The resulting pattern can be queried with `cairo-get-source` so that these attributes can be modified if desired, (eg. to create a repeating pattern with `cairo-pattern-set-extend`).

cr a cairo context
surface a surface to be used to set the source pattern
x User-space X coordinate for surface origin
y User-space Y coordinate for surface origin

`cairo-get-source (cr <cairo-t>) ⇒ (ret <cairo-pattern-t>)` [Function]

Gets the current source pattern for *cr*.

cr a cairo context

ret the current source pattern. This object is owned by cairo. To keep a reference to it, you must call `cairo-pattern-reference`.

`cairo-set-antialias (cr <cairo-t>)` [Function]

`(antialias <cairo-antialias-t>)`

Set the antialiasing mode of the rasterizer used for drawing shapes. This value is a hint, and a particular backend may or may not support a particular value. At the current time, no backend supports 'CAIRO_ANTIALIAS_SUBPIXEL' when drawing shapes.

Note that this option does not affect text rendering, instead see `cairo-font-options-set-antialias`.

cr a <cairo-t>

antialias the new antialiasing mode

`cairo-get-antialias (cr <cairo-t>)` [Function]

`⇒ (ret <cairo-antialias-t>)`

Gets the current shape antialiasing mode, as set by `cairo-set-shape-antialias`.

cr a cairo context

ret the current shape antialiasing mode.

`cairo-set-dash (cr <cairo-t>) ⇒ (dashes <double>)` [Function]

`(num-dashes <int>) (offset <double>)`

Sets the dash pattern to be used by `cairo-stroke`. A dash pattern is specified by *dashes*, an array of positive values. Each value provides the length of alternate "on" and "off" portions of the stroke. The *offset* specifies an offset into the pattern at which the stroke begins.

Each "on" segment will have caps applied as if the segment were a separate sub-path. In particular, it is valid to use an "on" length of 0.0 with `CAIRO_LINE_CAP_ROUND` or `CAIRO_LINE_CAP_SQUARE` in order to distributed dots or squares along a path.

Note: The length values are in user-space units as evaluated at the time of stroking. This is not necessarily the same as the user space at the time of `cairo-set-dash`.

If *num-dashes* is 0 dashing is disabled.

If *num-dashes* is 1 a symmetric pattern is assumed with alternating on and off portions of the size specified by the single value in *dashes*.

If any value in *dashes* is negative, or if all values are 0, then *cairo-t* will be put into an error state with a status of `<cairo-status-invalid-dash>`.

cr a cairo context

dashes an array specifying alternate lengths of on and off stroke portions

num-dashes

the length of the dashes array

offset

an offset into the dash pattern at which the stroke should start

`cairo-get-dash-count (cr <cairo-t>) ⇒ (ret <int>)` [Function]

This function returns the length of the dash array in *cr* (0 if dashing is not currently in effect).

See also `cairo-set-dash` and `cairo-get-dash`.

cr a `<cairo-t>`

ret the length of the dash array, or 0 if no dash array set.

Since 1.4

`cairo-set-fill-rule (cr <cairo-t>)` [Function]

(*fill-rule* `<cairo-fill-rule-t>`)

Set the current fill rule within the cairo context. The fill rule is used to determine which regions are inside or outside a complex (potentially self-intersecting) path. The current fill rule affects both `cairo_fill` and `cairo_clip`. See `<cairo-fill-rule-t>` for details on the semantics of each available fill rule.

cr a `<cairo-t>`

fill-rule a fill rule, specified as a `<cairo-fill-rule-t>`

`cairo-get-fill-rule (cr <cairo-t>)` [Function]

⇒ (*ret* `<cairo-fill-rule-t>`)

Gets the current fill rule, as set by `cairo-set-fill-rule`.

cr a cairo context

ret the current fill rule.

`cairo-set-line-cap (cr <cairo-t>)` [Function]

(*line-cap* `<cairo-line-cap-t>`)

Sets the current line cap style within the cairo context. See `<cairo-line-cap-t>` for details about how the available line cap styles are drawn.

As with the other stroke parameters, the current line cap style is examined by `cairo-stroke`, `cairo-stroke-extents`, and `cairo-stroke-to-path`, but does not have any effect during path construction.

cr a cairo context

line-cap a line cap style

`cairo-get-line-cap (cr <cairo-t>) ⇒ (ret <cairo-line-cap-t>)` [Function]

Gets the current line cap style, as set by `cairo-set-line-cap`.

cr a cairo context

ret the current line cap style.

cairo-set-line-join (*cr* <cairo-t>) [*line-join* <cairo-line-join-t>] [Function]

Sets the current line join style within the cairo context. See <cairo-line-join-t> for details about how the available line join styles are drawn.

As with the other stroke parameters, the current line join style is examined by **cairo-stroke**, **cairo-stroke-extents**, and **cairo-stroke-to-path**, but does not have any effect during path construction.

cr a cairo context

line-join a line joint style

cairo-get-line-join (*cr* <cairo-t>) [Function]
 ⇒ (*ret* <cairo-line-join-t>)

Gets the current line join style, as set by **cairo-set-line-join**.

cr a cairo context

ret the current line join style.

cairo-set-line-width (*cr* <cairo-t>) (*width* <double>) [Function]

Sets the current line width within the cairo context. The line width value specifies the diameter of a pen that is circular in user space, (though device-space pen may be an ellipse in general due to scaling/shear/rotation of the CTM).

Note: When the description above refers to user space and CTM it refers to the user space and CTM in effect at the time of the stroking operation, not the user space and CTM in effect at the time of the call to **cairo-set-line-width**. The simplest usage makes both of these spaces identical. That is, if there is no change to the CTM between a call to **cairo-set-line-width** and the stroking operation, then one can just pass user-space values to **cairo-set-line-width** and ignore this note.

As with the other stroke parameters, the current line width is examined by **cairo-stroke**, **cairo-stroke-extents**, and **cairo-stroke-to-path**, but does not have any effect during path construction.

The default line width value is 2.0.

cr a <cairo-t>

width a line width

cairo-get-line-width (*cr* <cairo-t>) ⇒ (*ret* <double>) [Function]

This function returns the current line width value exactly as set by **cairo-set-line-width**. Note that the value is unchanged even if the CTM has changed between the calls to **cairo-set-line-width** and **cairo-get-line-width**.

cr a cairo context

ret the current line width.

cairo-set-miter-limit (*cr* <cairo-t>) (*limit* <double>) [Function]

Sets the current miter limit within the cairo context.

If the current line join style is set to 'CAIRO_LINE_JOIN_MITER' (see **cairo-set-line-join**), the miter limit is used to determine whether the lines should be joined with a

bevel instead of a miter. Cairo divides the length of the miter by the line width. If the result is greater than the miter limit, the style is converted to a bevel.

As with the other stroke parameters, the current line miter limit is examined by `cairo-stroke`, `cairo-stroke-extents`, and `cairo-stroke-to-path`, but does not have any effect during path construction.

cr a cairo context

limit miter limit to set

`cairo-get-miter-limit (cr <cairo-t>) ⇒ (ret <double>)` [Function]

Gets the current miter limit, as set by `cairo-set-miter-limit`.

cr a cairo context

ret the current miter limit.

`cairo-set-operator (cr <cairo-t>) (op <cairo-operator-t>)` [Function]

Sets the compositing operator to be used for all drawing operations. See `<cairo-operator-t>` for details on the semantics of each available compositing operator.

XXX: I'd also like to direct the reader's attention to some (not-yet-written) section on cairo's imaging model. How would I do that if such a section existed? (cworth).

cr a <cairo-t>

op a compositing operator, specified as a <cairo-operator-t>

`cairo-get-operator (cr <cairo-t>) ⇒ (ret <cairo-operator-t>)` [Function]

Gets the current compositing operator for a cairo context.

cr a cairo context

ret the current compositing operator.

`cairo-set-tolerance (cr <cairo-t>) (tolerance <double>)` [Function]

Sets the tolerance used when converting paths into trapezoids. Curved segments of the path will be subdivided until the maximum deviation between the original path and the polygonal approximation is less than *tolerance*. The default value is 0.1. A larger value will give better performance, a smaller value, better appearance. (Reducing the value from the default value of 0.1 is unlikely to improve appearance significantly.)

cr a <cairo-t>

tolerance the tolerance, in device units (typically pixels)

`cairo-get-tolerance (cr <cairo-t>) ⇒ (ret <double>)` [Function]

Gets the current tolerance value, as set by `cairo-set-tolerance`.

cr a cairo context

ret the current tolerance value.

cairo-clip (*cr* <cairo-t>) [Function]

Establishes a new clip region by intersecting the current clip region with the current path as it would be filled by **cairo-fill** and according to the current fill rule (see **cairo-set-fill-rule**).

After **cairo-clip**, the current path will be cleared from the cairo context.

The current clip region affects all drawing operations by effectively masking out any changes to the surface that are outside the current clip region.

Calling **cairo-clip** can only make the clip region smaller, never larger. But the current clip is part of the graphics state, so a temporary restriction of the clip region can be achieved by calling **cairo-clip** within a **cairo-save/cairo-restore** pair. The only other means of increasing the size of the clip region is **cairo-reset-clip**.

cr a cairo context

cairo-clip-preserve (*cr* <cairo-t>) [Function]

Establishes a new clip region by intersecting the current clip region with the current path as it would be filled by **cairo-fill** and according to the current fill rule (see **cairo-set-fill-rule**).

Unlike **cairo-clip**, **cairo-clip-preserve** preserves the path within the cairo context.

The current clip region affects all drawing operations by effectively masking out any changes to the surface that are outside the current clip region.

Calling **cairo-clip** can only make the clip region smaller, never larger. But the current clip is part of the graphics state, so a temporary restriction of the clip region can be achieved by calling **cairo-clip** within a **cairo-save/cairo-restore** pair. The only other means of increasing the size of the clip region is **cairo-reset-clip**.

cr a cairo context

cairo-clip-extents (*cr* <cairo-t>) ⇒ (*x1* <double>) [Function]
 (*y1* <double>) (*x2* <double>) (*y2* <double>)

Computes a bounding box in user coordinates covering the area inside the current clip.

cr a cairo context

x1 left of the resulting extents

y1 top of the resulting extents

x2 right of the resulting extents

y2 bottom of the resulting extents

Since 1.4

cairo-reset-clip (*cr* <cairo-t>) [Function]

Reset the current clip region to its original, unrestricted state. That is, set the clip region to an infinitely large shape containing the target surface. Equivalently, if infinity is too hard to grasp, one can imagine the clip region being reset to the exact bounds of the target surface.

cairo-in-fill (*cr* <cairo-t>) (*x* <double>) (*y* <double>) [Function]
 ⇒ (*ret* <cairo-bool-t>)

Tests whether the given point is inside the area that would be affected by a **cairo-fill** operation given the current path and filling parameters. Surface dimensions and clipping are not taken into account.

See **cairo-fill**, **cairo-set-fill-rule** and **cairo-fill-preserve**.

cr a cairo context
x X coordinate of the point to test
y Y coordinate of the point to test
ret A non-zero value if the point is inside, or zero if outside.

cairo-mask (*cr* <cairo-t>) (*pattern* <cairo-pattern-t>) [Function]

A drawing operator that paints the current source using the alpha channel of *pattern* as a mask. (Opaque areas of *pattern* are painted with the source, transparent areas are not painted.)

cr a cairo context
pattern a <cairo-pattern-t>

cairo-mask-surface (*cr* <cairo-t>) [Function]
 (*surface* <cairo-surface-t>) (*surface-x* <double>)
 (*surface-y* <double>)

A drawing operator that paints the current source using the alpha channel of *surface* as a mask. (Opaque areas of *surface* are painted with the source, transparent areas are not painted.)

cr a cairo context
surface a <cairo-surface-t>
surface-x X coordinate at which to place the origin of *surface*
surface-y Y coordinate at which to place the origin of *surface*

cairo-paint (*cr* <cairo-t>) [Function]

A drawing operator that paints the current source everywhere within the current clip region.

cr a cairo context

cairo-paint-with-alpha (*cr* <cairo-t>) (*alpha* <double>) [Function]

A drawing operator that paints the current source everywhere within the current clip region using a mask of constant alpha value *alpha*. The effect is similar to **cairo-paint**, but the drawing is faded out using the alpha value.

cr a cairo context
alpha alpha value, between 0 (transparent) and 1 (opaque)

cairo-stroke (*cr* <cairo-t>) [Function]

A drawing operator that strokes the current path according to the current line width, line join, line cap, and dash settings. After `cairo-stroke`, the current path will be cleared from the cairo context. See `cairo-set-line-width`, `cairo-set-line-join`, `cairo-set-line-cap`, `cairo-set-dash`, and `cairo-stroke-preserve`.

Note: Degenerate segments and sub-paths are treated specially and provide a useful result. These can result in two different situations:

1. Zero-length "on" segments set in `cairo-set-dash`. If the cap style is `CAIRO_LINE_CAP_ROUND` or `CAIRO_LINE_CAP_SQUARE` then these segments will be drawn as circular dots or squares respectively. In the case of `CAIRO_LINE_CAP_SQUARE`, the orientation of the squares is determined by the direction of the underlying path.

2. A sub-path created by `cairo-move-to` followed by either a `cairo-close-path` or one or more calls to `cairo-line-to` to the same coordinate as the `cairo-move-to`. If the cap style is `CAIRO_LINE_CAP_ROUND` then these sub-paths will be drawn as circular dots. Note that in the case of `CAIRO_LINE_CAP_SQUARE` a degenerate sub-path will not be drawn at all, (since the correct orientation is indeterminate).

In no case will a cap style of `CAIRO_LINE_CAP_BUTT` cause anything to be drawn in the case of either degenerate segments or sub-paths.

cr a cairo context

cairo-stroke-preserve (*cr* <cairo-t>) [Function]

A drawing operator that strokes the current path according to the current line width, line join, line cap, and dash settings. Unlike `cairo-stroke`, `cairo-stroke-preserve` preserves the path within the cairo context.

See `cairo-set-line-width`, `cairo-set-line-join`, `cairo-set-line-cap`, `cairo-set-dash`, and `cairo-stroke-preserve`.

cr a cairo context

cairo-stroke-extents (*cr* <cairo-t>) ⇒ (*x1* <double>) [Function]
(*y1* <double>) (*x2* <double>) (*y2* <double>)

Computes a bounding box in user coordinates covering the area that would be affected by a `cairo-stroke` operation operation given the current path and stroke parameters. If the current path is empty,

cr a cairo context

x1 left of the resulting extents

y1 top of the resulting extents

x2 right of the resulting extents

y2 bottom of the resulting extents

cairo-in-stroke (*cr* <cairo-t>) (*x* <double>) (*y* <double>) [Function]
⇒ (*ret* <cairo-bool-t>)

Tests whether the given point is inside the area that would be affected by a `cairo-stroke` operation given the current path and stroking parameters. Surface dimensions and clipping are not taken into account.

See `cairo-stroke`, `cairo-set-line-width`, `cairo-set-line-join`, `cairo-set-line-cap`, `cairo-set-dash`, and `cairo-stroke-preserve`.

cr a cairo context

x X coordinate of the point to test

y Y coordinate of the point to test

ret A non-zero value if the point is inside, or zero if outside.

`cairo-copy-page (cr <cairo-t>)` [Function]

Emits the current page for backends that support multiple pages, but doesn't clear it, so, the contents of the current page will be retained for the next page too. Use `cairo-show-page` if you want to get an empty page after the emission.

cr a cairo context

`cairo-show-page (cr <cairo-t>)` [Function]

Emits and clears the current page for backends that support multiple pages. Use `cairo-copy-page` if you don't want to clear the page.

cr a cairo context

2 Paths

Creating paths and manipulating path data

2.1 Overview

2.2 Usage

`cairo-copy-path (cr <cairo-t>) ⇒ (ret <cairo-path-t>)` [Function]

Creates a copy of the current path and returns it to the user as a `<cairo-path-t>`. See `<cairo-path-data-t>` for hints on how to iterate over the returned data structure.

This function will always return a valid pointer, but the result will have no data (`'data==NULL'` and `'num_data==0'`), if either of the following conditions hold:

1. If there is insufficient memory to copy the path. In this case `'path->status'` will be set to `'CAIRO_STATUS_NO_MEMORY'`.
2. If `cr` is already in an error state. In this case `'path->status'` will contain the same status that would be returned by `cairo-status`.

In either case, `'path->status'` will be set to `'CAIRO_STATUS_NO_MEMORY'` (regardless of what the error status in `cr` might have been).

`cr` a cairo context

`ret` the copy of the current path. The caller owns the returned object and should call `cairo-path-destroy` when finished with it.

`cairo-copy-path-flat (cr <cairo-t>) ⇒ (ret <cairo-path-t>)` [Function]

Gets a flattened copy of the current path and returns it to the user as a `<cairo-path-t>`. See `<cairo-path-data-t>` for hints on how to iterate over the returned data structure.

This function is like `cairo-copy-path` except that any curves in the path will be approximated with piecewise-linear approximations, (accurate to within the current tolerance value). That is, the result is guaranteed to not have any elements of type `'CAIRO_PATH_CURVE_TO'` which will instead be replaced by a series of `'CAIRO_PATH_LINE_TO'` elements.

This function will always return a valid pointer, but the result will have no data (`'data==NULL'` and `'num_data==0'`), if either of the following conditions hold:

1. If there is insufficient memory to copy the path. In this case `'path->status'` will be set to `'CAIRO_STATUS_NO_MEMORY'`.
2. If `cr` is already in an error state. In this case `'path->status'` will contain the same status that would be returned by `cairo-status`.

`cr` a cairo context

`ret` the copy of the current path. The caller owns the returned object and should call `cairo-path-destroy` when finished with it.

cairo-append-path (*cr* <cairo-t>) (*path* <cairo-path-t>) [Function]

Append the *path* onto the current path. The *path* may be either the return value from one of `cairo-copy-path` or `cairo-copy-path-flat` or it may be constructed manually. See <cairo-path-t> for details on how the path data structure should be initialized, and note that ‘`path->status`’ must be initialized to ‘`CAIRO_STATUS_SUCCESS`’.

cr a cairo context

path path to be appended

cairo-get-current-point (*cr* <cairo-t>) ⇒ (*x* <double>) [Function]
(*y* <double>)

Gets the current point of the current path, which is conceptually the final point reached by the path so far.

The current point is returned in the user-space coordinate system. If there is no defined current point then *x* and *y* will both be set to 0.0.

Most path construction functions alter the current point. See the following for details on how they affect the current point:

`cairo-new-path`, `cairo-move-to`, `cairo-line-to`, `cairo-curve-to`, `cairo-arc`,
`cairo-rel-move-to`, `cairo-rel-line-to`, `cairo-rel-curve-to`, `cairo-arc`,
`cairo-text-path`, `cairo-stroke-to-path`

cr a cairo context

x return value for X coordinate of the current point

y return value for Y coordinate of the current point

cairo-new-path (*cr* <cairo-t>) [Function]

Clears the current path. After this call there will be no path and no current point.

cr a cairo context

cairo-new-sub-path (*cr* <cairo-t>) [Function]

Begin a new sub-path. Note that the existing path is not affected. After this call there will be no current point.

In many cases, this call is not needed since new sub-paths are frequently started with `cairo-move-to`.

A call to `cairo-new-sub-path` is particularly useful when beginning a new sub-path with one of the `cairo-arc` calls. This makes things easier as it is no longer necessary to manually compute the arc’s initial coordinates for a call to `cairo-move-to`.

cr a cairo context

Since 1.2

cairo-close-path (*cr* <cairo-t>) [Function]

Adds a line segment to the path from the current point to the beginning of the current sub-path, (the most recent point passed to `cairo-move-to`), and closes this sub-path. After this call the current point will be at the joined endpoint of the sub-path.

The behavior of `cairo-close-path` is distinct from simply calling `cairo-line-to` with the equivalent coordinate in the case of stroking. When a closed sub-path is

stroked, there are no caps on the ends of the sub-path. Instead, there is a line join connecting the final and initial segments of the sub-path.

If there is no current point before the call to `cairo_close_path`, this function will have no effect.

Note: As of cairo version 1.2.4 any call to `cairo_close_path` will place an explicit `MOVE_TO` element into the path immediately after the `CLOSE_PATH` element, (which can be seen in `cairo-copy-path` for example). This can simplify path processing in some cases as it may not be necessary to save the "last move_to point" during processing as the `MOVE_TO` immediately after the `CLOSE_PATH` will provide that point.

cr a cairo context

`cairo_arc (cr <cairo-t>) (xc <double>) (yc <double>) [Function]
 (radius <double>) (angle1 <double>) (angle2 <double>)`

Adds a circular arc of the given *radius* to the current path. The arc is centered at (*xc*, *yc*), begins at *angle1* and proceeds in the direction of increasing angles to end at *angle2*. If *angle2* is less than *angle1* it will be progressively increased by $2 * M_PI$ until it is greater than *angle1*.

If there is a current point, an initial line segment will be added to the path to connect the current point to the beginning of the arc.

Angles are measured in radians. An angle of 0.0 is in the direction of the positive X axis (in user space). An angle of $M_PI / 2.0$ radians (90 degrees) is in the direction of the positive Y axis (in user space). Angles increase in the direction from the positive X axis toward the positive Y axis. So with the default transformation matrix, angles increase in a clockwise direction.

(To convert from degrees to radians, use `'degrees * (M_PI / 180.)'`.)

This function gives the arc in the direction of increasing angles; see `cairo_arc_negative` to get the arc in the direction of decreasing angles.

The arc is circular in user space. To achieve an elliptical arc, you can scale the current transformation matrix by different amounts in the X and Y directions. For example, to draw an ellipse in the box given by *x*, *y*, *width*, *height*:

```
cairo_save (cr);
cairo_translate (cr, x + width / 2., y + height / 2.);
cairo_scale (cr, width / 2., height / 2.);
cairo_arc (cr, 0., 0., 1., 0., 2 * M_PI);
cairo_restore (cr);
```

cr a cairo context

xc X position of the center of the arc

yc Y position of the center of the arc

radius the radius of the arc

angle1 the start angle, in radians

angle2 the end angle, in radians

cairo-arc-negative (*cr* <cairo-t>) (*xc* <double>) (*yc* <double>) [Function]
 (*radius* <double>) (*angle1* <double>) (*angle2* <double>)

Adds a circular arc of the given *radius* to the current path. The arc is centered at (*xc*, *yc*), begins at *angle1* and proceeds in the direction of decreasing angles to end at *angle2*. If *angle2* is greater than *angle1* it will be progressively decreased by $2 * M_PI$ until it is less than *angle1*.

See **cairo-arc** for more details. This function differs only in the direction of the arc between the two angles.

cr a cairo context
xc X position of the center of the arc
yc Y position of the center of the arc
radius the radius of the arc
angle1 the start angle, in radians
angle2 the end angle, in radians

cairo-curve-to (*cr* <cairo-t>) (*x1* <double>) (*y1* <double>) [Function]
 (*x2* <double>) (*y2* <double>) (*x3* <double>) (*y3* <double>)

Adds a cubic Bzier spline to the path from the current point to position (*x3*, *y3*) in user-space coordinates, using (*x1*, *y1*) and (*x2*, *y2*) as the control points. After this call the current point will be (*x3*, *y3*).

If there is no current point before the call to **cairo-curve-to** this function will behave as if preceded by a call to **cairo_move_to** (*cr*, *x1*, *y1*).

cr a cairo context
x1 the X coordinate of the first control point
y1 the Y coordinate of the first control point
x2 the X coordinate of the second control point
y2 the Y coordinate of the second control point
x3 the X coordinate of the end of the curve
y3 the Y coordinate of the end of the curve

cairo-line-to (*cr* <cairo-t>) (*x* <double>) (*y* <double>) [Function]

Adds a line to the path from the current point to position (*x*, *y*) in user-space coordinates. After this call the current point will be (*x*, *y*).

If there is no current point before the call to **cairo-line-to** this function will behave as **cairo_move_to** (*cr*, *x*, *y*).

cr a cairo context
x the X coordinate of the end of the new line
y the Y coordinate of the end of the new line

cairo-move-to (*cr* <cairo-t>) (*x* <double>) (*y* <double>) [Function]

Begin a new sub-path. After this call the current point will be (*x*, *y*).

cr a cairo context
x the X coordinate of the new position
y the Y coordinate of the new position

cairo-rectangle (*cr* <cairo-t>) (*x* <double>) (*y* <double>) [Function]
(*width* <double>) (*height* <double>)

Adds a closed sub-path rectangle of the given size to the current path at position (*x*, *y*) in user-space coordinates.

This function is logically equivalent to:

```
cairo_move_to (cr, x, y);
cairo_rel_line_to (cr, width, 0);
cairo_rel_line_to (cr, 0, height);
cairo_rel_line_to (cr, -width, 0);
cairo_close_path (cr);
```

cr a cairo context
x the X coordinate of the top left corner of the rectangle
y the Y coordinate to the top left corner of the rectangle
width the width of the rectangle
height the height of the rectangle

cairo-glyph-path (*cr* <cairo-t>) (*glyphs* <cairo-glyph-t>) [Function]
(*num-glyphs* <int>)

Adds closed paths for the glyphs to the current path. The generated path if filled, achieves an effect similar to that of **cairo-show-glyphs**.

cr a cairo context
glyphs array of glyphs to show
num-glyphs number of glyphs to show

cairo-text-path (*cr* <cairo-t>) (*utf8* <char>) [Function]

Adds closed paths for text to the current path. The generated path if filled, achieves an effect similar to that of **cairo-show-text**.

Text conversion and positioning is done similar to **cairo-show-text**.

Like **cairo-show-text**, After this call the current point is moved to the origin of where the next glyph would be placed in this same progression. That is, the current point will be at the origin of the final glyph offset by its advance values. This allows for chaining multiple calls to **cairo-text-path** without having to set current point in between.

NOTE: The `cairo-text-path` function call is part of what the cairo designers call the "toy" text API. It is convenient for short demos and simple programs, but it is not expected to be adequate for serious text-using applications. See `cairo-glyph-path` for the "real" text path API in cairo.

cr a cairo context
utf8 a string of text encoded in UTF-8

`cairo-rel-curve-to` (*cr* <cairo-t>) (*dx1* <double>) [Function]
 (*dy1* <double>) (*dx2* <double>) (*dy2* <double>) (*dx3* <double>)
 (*dy3* <double>)

Relative-coordinate version of `cairo-curve-to`. All offsets are relative to the current point. Adds a cubic Bzier spline to the path from the current point to a point offset from the current point by (*dx3*, *dy3*), using points offset by (*dx1*, *dy1*) and (*dx2*, *dy2*) as the control points. After this call the current point will be offset by (*dx3*, *dy3*).

Given a current point of (*x*, *y*), `cairo_rel_curve_to` (*cr*, *dx1*, *dy1*, *dx2*, *dy2*, *dx3*, *dy3*) is logically equivalent to `cairo_curve_to` (*cr*, *x* + *dx1*, *y* + *dy1*, *x* + *dx2*, *y* + *dy2*, *x* + *dx3*, *y* + *dy3*).

It is an error to call this function with no current point. Doing so will cause *cr* to shutdown with a status of `CAIRO_STATUS_NO_CURRENT_POINT`.

cr a cairo context
dx1 the X offset to the first control point
dy1 the Y offset to the first control point
dx2 the X offset to the second control point
dy2 the Y offset to the second control point
dx3 the X offset to the end of the curve
dy3 the Y offset to the end of the curve

`cairo-rel-line-to` (*cr* <cairo-t>) (*dx* <double>) (*dy* <double>) [Function]

Relative-coordinate version of `cairo-line-to`. Adds a line to the path from the current point to a point that is offset from the current point by (*dx*, *dy*) in user space. After this call the current point will be offset by (*dx*, *dy*).

Given a current point of (*x*, *y*), `cairo_rel_line_to`(*cr*, *dx*, *dy*) is logically equivalent to `cairo_line_to` (*cr*, *x* + *dx*, *y* + *dy*).

It is an error to call this function with no current point. Doing so will cause *cr* to shutdown with a status of `CAIRO_STATUS_NO_CURRENT_POINT`.

cr a cairo context
dx the X offset to the end of the new line
dy the Y offset to the end of the new line

`cairo-rel-move-to (cr <cairo-t>) (dx <double>) (dy <double>)` [Function]

Begin a new sub-path. After this call the current point will offset by (x, y) .

Given a current point of (x, y) , `cairo_rel_move_to(cr, dx, dy)` is logically equivalent to `cairo_move_to (cr, x + dx, y + dy)`.

It is an error to call this function with no current point. Doing so will cause `cr` to shutdown with a status of `CAIRO_STATUS_NO_CURRENT_POINT`.

`cr` a cairo context

`dx` the X offset

`dy` the Y offset

3 Patterns

Gradients and filtered sources

3.1 Overview

3.2 Usage

`cairo-pattern-add-color-stop-rgb` [Function]
 (*pattern* <cairo-pattern-t>) (*offset* <double>) (*red* <double>)
 (*green* <double>) (*blue* <double>)

Adds an opaque color stop to a gradient pattern. The offset specifies the location along the gradient's control vector. For example, a linear gradient's control vector is from (x0,y0) to (x1,y1) while a radial gradient's control vector is from any point on the start circle to the corresponding point on the end circle.

The color is specified in the same way as in `cairo-set-source-rgb`.

Note: If the pattern is not a gradient pattern, (eg. a linear or radial pattern), then the pattern will be put into an error status with a status of 'CAIRO_STATUS_PATTERN_TYPE_MISMATCH'.

pattern a <cairo-pattern-t>
offset an offset in the range [0.0 .. 1.0]
red red component of color
green green component of color
blue blue component of color

`cairo-pattern-add-color-stop-rgba` [Function]
 (*pattern* <cairo-pattern-t>) (*offset* <double>) (*red* <double>)
 (*green* <double>) (*blue* <double>) (*alpha* <double>)

Adds a translucent color stop to a gradient pattern. The offset specifies the location along the gradient's control vector. For example, a linear gradient's control vector is from (x0,y0) to (x1,y1) while a radial gradient's control vector is from any point on the start circle to the corresponding point on the end circle.

The color is specified in the same way as in `cairo-set-source-rgba`.

Note: If the pattern is not a gradient pattern, (eg. a linear or radial pattern), then the pattern will be put into an error status with a status of 'CAIRO_STATUS_PATTERN_TYPE_MISMATCH'.

pattern a <cairo-pattern-t>
offset an offset in the range [0.0 .. 1.0]
red red component of color
green green component of color
blue blue component of color
alpha alpha component of color

`cairo-pattern-get-color-stop-rgba` [Function]
 (`pattern` <cairo-pattern-t>) (`index` <int>)
 ⇒ (`ret` <cairo-status-t>) (`offset` <double>) (`red` <double>)
 (`green` <double>) (`blue` <double>) (`alpha` <double>)

Gets the color and offset information at the given `index` for a gradient pattern. Values of `index` are 0 to 1 less than the number returned by `cairo-pattern-get-color-stop-count`.

`pattern` a <cairo-pattern-t>
`index` index of the stop to return data for
`offset` return value for the offset of the stop, or '#f'
`red` return value for red component of color, or '#f'
`green` return value for green component of color, or '#f'
`blue` return value for blue component of color, or '#f'
`alpha` return value for alpha component of color, or '#f'
`ret` 'CAIRO_STATUS_SUCCESS', or 'CAIRO_STATUS_INVALID_INDEX' if `index` is not valid for the given pattern. If the pattern is not a gradient pattern, 'CAIRO_STATUS_PATTERN_TYPE_MISMATCH' is returned.

Since 1.4

`cairo-pattern-create-rgb` (`red` <double>) (`green` <double>) [Function]
 (`blue` <double>) ⇒ (`ret` <cairo-pattern-t>)

Creates a new `cairo_pattern_t` corresponding to an opaque color. The color components are floating point numbers in the range 0 to 1. If the values passed in are outside that range, they will be clamped.

`red` red component of the color
`green` green component of the color
`blue` blue component of the color
`ret` the newly created <cairo-pattern-t> if successful, or an error pattern in case of no memory. The caller owns the returned object and should call `cairo-pattern-destroy` when finished with it. This function will always return a valid pointer, but if an error occurred the pattern status will be set to an error. To inspect the status of a pattern use `cairo-pattern-status`.

`cairo-pattern-create-rgba` (`red` <double>) (`green` <double>) [Function]
 (`blue` <double>) (`alpha` <double>) ⇒ (`ret` <cairo-pattern-t>)

Creates a new `cairo_pattern_t` corresponding to a translucent color. The color components are floating point numbers in the range 0 to 1. If the values passed in are outside that range, they will be clamped.

`red` red component of the color
`green` green component of the color

blue blue component of the color
alpha alpha component of the color
ret the newly created `<cairo-pattern-t>` if successful, or an error pattern in case of no memory. The caller owns the returned object and should call `cairo-pattern-destroy` when finished with it. This function will always return a valid pointer, but if an error occurred the pattern status will be set to an error. To inspect the status of a pattern use `cairo-pattern-status`.

`cairo-pattern-get-rgba` (*pattern* `<cairo-pattern-t>`) [Function]
 ⇒ (*ret* `<cairo-status-t>`) (*red* `<double>`) (*green* `<double>`)
 (*blue* `<double>`) (*alpha* `<double>`)

Gets the solid color for a solid color pattern.

pattern a `<cairo-pattern-t>`
red return value for red component of color, or `'#f'`
green return value for green component of color, or `'#f'`
blue return value for blue component of color, or `'#f'`
alpha return value for alpha component of color, or `'#f'`
ret `'CAIRO_STATUS_SUCCESS'`, or `'CAIRO_STATUS_PATTERN_TYPE_MISMATCH'` if the pattern is not a solid color pattern.

Since 1.4

`cairo-pattern-create-for-surface` [Function]
 (*surface* `<cairo-surface-t>`) ⇒ (*ret* `<cairo-pattern-t>`)

Create a new `cairo_pattern_t` for the given surface.

surface the surface
ret the newly created `<cairo-pattern-t>` if successful, or an error pattern in case of no memory. The caller owns the returned object and should call `cairo-pattern-destroy` when finished with it. This function will always return a valid pointer, but if an error occurred the pattern status will be set to an error. To inspect the status of a pattern use `cairo-pattern-status`.

`cairo-pattern-get-surface` (*pattern* `<cairo-pattern-t>`) [Function]
 ⇒ (*ret* `<cairo-status-t>`) (*surface* `<cairo-surface-t*>`)

Gets the surface of a surface pattern. The reference returned in *surface* is owned by the pattern; the caller should call `cairo-surface-reference` if the surface is to be retained.

pattern a `<cairo-pattern-t>`
surface return value for surface of pattern, or `'#f'`
ret `'CAIRO_STATUS_SUCCESS'`, or `'CAIRO_STATUS_PATTERN_TYPE_MISMATCH'` if the pattern is not a surface pattern.

Since 1.4

`cairo-pattern-create-linear` (*x0* <double>) (*y0* <double>) [Function]
 (*x1* <double>) (*y1* <double>) ⇒ (*ret* <cairo-pattern-t>)

Create a new linear gradient `cairo_pattern_t` along the line defined by (*x0*, *y0*) and (*x1*, *y1*). Before using the gradient pattern, a number of color stops should be defined using `cairo-pattern-add-color-stop-rgb` or `cairo-pattern-add-color-stop-rgba`.

Note: The coordinates here are in pattern space. For a new pattern, pattern space is identical to user space, but the relationship between the spaces can be changed with `cairo-pattern-set-matrix`.

x0 x coordinate of the start point
y0 y coordinate of the start point
x1 x coordinate of the end point
y1 y coordinate of the end point
ret the newly created <cairo-pattern-t> if successful, or an error pattern in case of no memory. The caller owns the returned object and should call `cairo-pattern-destroy` when finished with it. This function will always return a valid pointer, but if an error occurred the pattern status will be set to an error. To inspect the status of a pattern use `cairo-pattern-status`.

`cairo-pattern-get-linear-points` (*pattern* <cairo-pattern-t>) [Function]
 ⇒ (*ret* <cairo-status-t>) (*x0* <double>) (*y0* <double>)
 (*x1* <double>) (*y1* <double>)

Gets the gradient endpoints for a linear gradient.

pattern a <cairo-pattern-t>
x0 return value for the x coordinate of the first point, or '#f'
y0 return value for the y coordinate of the first point, or '#f'
x1 return value for the x coordinate of the second point, or '#f'
y1 return value for the y coordinate of the second point, or '#f'
ret 'CAIRO_STATUS_SUCCESS', or 'CAIRO_STATUS_PATTERN_TYPE_MISMATCH' if *pattern* is not a linear gradient pattern.

Since 1.4

`cairo-pattern-create-radial` (*cx0* <double>) (*cy0* <double>) [Function]
 (*radius0* <double>) (*cx1* <double>) (*cy1* <double>)
 (*radius1* <double>) ⇒ (*ret* <cairo-pattern-t>)

Creates a new radial gradient `cairo_pattern_t` between the two circles defined by (*x0*, *y0*, *c0*) and (*x1*, *y1*, *c0*). Before using the gradient pattern, a number of color stops should be defined using `cairo-pattern-add-color-stop-rgb` or `cairo-pattern-add-color-stop-rgba`.

Note: The coordinates here are in pattern space. For a new pattern, pattern space is identical to user space, but the relationship between the spaces can be changed with `cairo-pattern-set-matrix`.

cx0 x coordinate for the center of the start circle
cy0 y coordinate for the center of the start circle
radius0 radius of the start circle
cx1 x coordinate for the center of the end circle
cy1 y coordinate for the center of the end circle
radius1 radius of the end circle
ret the newly created `<cairo-pattern-t>` if successful, or an error pattern in case of no memory. The caller owns the returned object and should call `cairo-pattern-destroy` when finished with it. This function will always return a valid pointer, but if an error occurred the pattern status will be set to an error. To inspect the status of a pattern use `cairo-pattern-status`.

cairo-pattern-get-radial-circles [Function]
 (*pattern* `<cairo-pattern-t>`) ⇒ (*ret* `<cairo-status-t>`)
 (*x0* `<double>`) (*y0* `<double>`) (*r0* `<double>`) (*x1* `<double>`)
 (*y1* `<double>`) (*r1* `<double>`)

Gets the gradient endpoint circles for a radial gradient, each specified as a center coordinate and a radius.

pattern a `<cairo-pattern-t>`
x0 return value for the x coordinate of the center of the first circle, or '#f'
y0 return value for the y coordinate of the center of the first circle, or '#f'
r0 return value for the radius of the first circle, or '#f'
x1 return value for the x coordinate of the center of the second circle, or '#f'
y1 return value for the y coordinate of the center of the second circle, or '#f'
r1 return value for the radius of the second circle, or '#f'
ret 'CAIRO_STATUS_SUCCESS', or 'CAIRO_STATUS_PATTERN_TYPE_MISMATCH' if *pattern* is not a radial gradient pattern.

Since 1.4

cairo-pattern-set-extend (*pattern* `<cairo-pattern-t>`) [Function]
 (*extend* `<cairo-extend-t>`)

Sets the mode to be used for drawing outside the area of a pattern. See `<cairo-extend-t>` for details on the semantics of each extend strategy.

pattern a `<cairo-pattern-t>`
extend a `<cairo-extend-t>` describing how the area outside of the pattern will be drawn

`cairo-pattern-get-extend` (*pattern* <cairo-pattern-t>) [Function]
 ⇒ (*ret* <cairo-extend-t>)

Gets the current extend mode for a pattern. See <cairo-extend-t> for details on the semantics of each extend strategy.

pattern a <cairo-pattern-t>

ret the current extend strategy used for drawing the pattern.

`cairo-pattern-set-filter` (*pattern* <cairo-pattern-t>) [Function]
 (*filter* <cairo-filter-t>)

Sets the filter to be used for resizing when using this pattern. See <cairo-filter-t> for details on each filter.

pattern a <cairo-pattern-t>

filter a <cairo-filter-t> describing the filter to use for resizing the pattern

`cairo-pattern-get-filter` (*pattern* <cairo-pattern-t>) [Function]
 ⇒ (*ret* <cairo-filter-t>)

Gets the current filter for a pattern. See <cairo-filter-t> for details on each filter.

pattern a <cairo-pattern-t>

ret the current filter used for resizing the pattern.

`cairo-pattern-set-matrix` (*pattern* <cairo-pattern-t>) [Function]
 (*matrix* <cairo-matrix-t>)

Sets the pattern's transformation matrix to *matrix*. This matrix is a transformation from user space to pattern space.

When a pattern is first created it always has the identity matrix for its transformation matrix, which means that pattern space is initially identical to user space.

Important: Please note that the direction of this transformation matrix is from user space to pattern space. This means that if you imagine the flow from a pattern to user space (and on to device space), then coordinates in that flow will be transformed by the inverse of the pattern matrix.

For example, if you want to make a pattern appear twice as large as it does by default the correct code to use is:

```
cairo_matrix_init_scale (&matrix, 0.5, 0.5);
cairo_pattern_set_matrix (pattern, &matrix);
```

Meanwhile, using values of 2.0 rather than 0.5 in the code above would cause the pattern to appear at half of its default size.

Also, please note the discussion of the user-space locking semantics of `cairo-set-source`.

pattern a <cairo-pattern-t>

matrix a <cairo-matrix-t>

`cairo-pattern-get-matrix` (*pattern* <cairo-pattern-t>) [Function]
(*matrix* <cairo-matrix-t>)

Stores the pattern's transformation matrix into *matrix*.

pattern a <cairo-pattern-t>

matrix return value for the matrix

`cairo-pattern-get-type` (*pattern* <cairo-pattern-t>) [Function]
⇒ (*ret* <cairo-pattern-type-t>)

This function returns the type a pattern. See <cairo-pattern-type-t> for available types.

pattern a <cairo-pattern-t>

ret The type of *pattern*.

Since 1.2

4 Transformations

Manipulating the current transformation matrix

4.1 Overview

4.2 Usage

`cairo-translate` (*cr* <cairo-t>) (*tx* <double>) (*ty* <double>) [Function]

Modifies the current transformation matrix (CTM) by translating the user-space origin by (*tx*, *ty*). This offset is interpreted as a user-space coordinate according to the CTM in place before the new call to `cairo_translate`. In other words, the translation of the user-space origin takes place after any existing transformation.

cr a cairo context
tx amount to translate in the X direction
ty amount to translate in the Y direction

`cairo-scale` (*cr* <cairo-t>) (*sx* <double>) (*sy* <double>) [Function]

Modifies the current transformation matrix (CTM) by scaling the X and Y user-space axes by *sx* and *sy* respectively. The scaling of the axes takes place after any existing transformation of user space.

cr a cairo context
sx scale factor for the X dimension
sy scale factor for the Y dimension

`cairo-rotate` (*cr* <cairo-t>) (*angle* <double>) [Function]

Modifies the current transformation matrix (CTM) by rotating the user-space axes by *angle* radians. The rotation of the axes takes places after any existing transformation of user space. The rotation direction for positive angles is from the positive X axis toward the positive Y axis.

cr a cairo context
angle angle (in radians) by which the user-space axes will be rotated

`cairo-transform` (*cr* <cairo-t>) (*matrix* <cairo-matrix-t>) [Function]

Modifies the current transformation matrix (CTM) by applying *matrix* as an additional transformation. The new transformation of user space takes place after any existing transformation.

cr a cairo context
matrix a transformation to be applied to the user-space axes

`cairo-set-matrix` (*cr* <cairo-t>) (*matrix* <cairo-matrix-t>) [Function]

Modifies the current transformation matrix (CTM) by setting it equal to *matrix*.

cr a cairo context
matrix a transformation matrix from user space to device space

cairo-get-matrix (*cr* <cairo-t>) (*matrix* <cairo-matrix-t>) [Function]
Stores the current transformation matrix (CTM) into *matrix*.

cr a cairo context
matrix return value for the matrix

cairo-identity-matrix (*cr* <cairo-t>) [Function]
Resets the current transformation matrix (CTM) by setting it equal to the identity matrix. That is, the user-space and device-space axes will be aligned and one user-space unit will transform to one device-space unit.

cr a cairo context

cairo-user-to-device (*cr* <cairo-t>) \Rightarrow (*x* <double>) [Function]
(*y* <double>)

Transform a coordinate from user space to device space by multiplying the given point by the current transformation matrix (CTM).

cr a cairo context
x X value of coordinate (in/out parameter)
y Y value of coordinate (in/out parameter)

cairo-user-to-device-distance (*cr* <cairo-t>) [Function]
 \Rightarrow (*dx* <double>) (*dy* <double>)

Transform a distance vector from user space to device space. This function is similar to **cairo-user-to-device** except that the translation components of the CTM will be ignored when transforming (*dx,dy*).

cr a cairo context
dx X component of a distance vector (in/out parameter)
dy Y component of a distance vector (in/out parameter)

cairo-device-to-user (*cr* <cairo-t>) \Rightarrow (*x* <double>) [Function]
(*y* <double>)

Transform a coordinate from device space to user space by multiplying the given point by the inverse of the current transformation matrix (CTM).

cr a cairo
x X value of coordinate (in/out parameter)
y Y value of coordinate (in/out parameter)

cairo-device-to-user-distance (*cr* <cairo-t>) [Function]
 \Rightarrow (*dx* <double>) (*dy* <double>)

Transform a distance vector from device space to user space. This function is similar to **cairo-device-to-user** except that the translation components of the inverse CTM will be ignored when transforming (*dx,dy*).

cr a cairo context
dx X component of a distance vector (in/out parameter)
dy Y component of a distance vector (in/out parameter)

5 Text

Rendering text and sets of glyphs

5.1 Overview

5.2 Usage

`cairo-select-font-face` (*cr* <cairo-t>) (*family* <char>) [Function]
 (*slant* <cairo-font-slant-t>) (*weight* <cairo-font-weight-t>)

Selects a family and style of font from a simplified description as a family name, slant and weight. This function is meant to be used only for applications with simple font needs: Cairo doesn't provide for operations such as listing all available fonts on the system, and it is expected that most applications will need to use a more comprehensive font handling and text layout library in addition to cairo.

cr a <cairo-t>
family a font family name, encoded in UTF-8
slant the slant for the font
weight the weight for the font

`cairo-set-font-size` (*cr* <cairo-t>) (*size* <double>) [Function]

Sets the current font matrix to a scale by a factor of *size*, replacing any font matrix previously set with `cairo-set-font-size` or `cairo-set-font-matrix`. This results in a font size of *size* user space units. (More precisely, this matrix will result in the font's em-square being a *size* by *size* square in user space.)

cr a <cairo-t>
size the new font size, in user space units

`cairo-set-font-matrix` (*cr* <cairo-t>) [Function]
 (*matrix* <cairo-matrix-t>)

Sets the current font matrix to *matrix*. The font matrix gives a transformation from the design space of the font (in this space, the em-square is 1 unit by 1 unit) to user space. Normally, a simple scale is used (see `cairo-set-font-size`), but a more complex font matrix can be used to shear the font or stretch it unequally along the two axes

cr a <cairo-t>
matrix a <cairo-matrix-t> describing a transform to be applied to the current font.

`cairo-get-font-matrix` (*cr* <cairo-t>) [Function]
 (*matrix* <cairo-matrix-t>)

Stores the current font matrix into *matrix*. See `cairo-set-font-matrix`.

cr a <cairo-t>
matrix return value for the matrix

`cairo-set-font-options` (*cr* <cairo-t>) [Function]
 (*options* <cairo-font-options-t>)

Sets a set of custom font rendering options for the <cairo-t>. Rendering options are derived by merging these options with the options derived from underlying surface; if the value in *options* has a default value (like ‘CAIRO_ANTIALIAS_DEFAULT’), then the value from the surface is used.

cr a <cairo-t>
options font options to use

`cairo-get-font-options` (*cr* <cairo-t>) [Function]
 (*options* <cairo-font-options-t>)

Retrieves font rendering options set via <cairo-set-font-options>. Note that the returned options do not include any options derived from the underlying surface; they are literally the options passed to `cairo-set-font-options`.

cr a <cairo-t>
options a <cairo-font-options-t> object into which to store the retrieved options. All existing values are overwritten

`cairo-set-font-face` (*cr* <cairo-t>) [Function]
 (*font-face* <cairo-font-face-t>)

Replaces the current <cairo-font-face-t> object in the <cairo-t> with *font-face*. The replaced font face in the <cairo-t> will be destroyed if there are no other references to it.

cr a <cairo-t>
font-face a <cairo-font-face-t>, or ‘#f’ to restore to the default font

`cairo-get-font-face` (*cr* <cairo-t>) [Function]
 ⇒ (*ret* <cairo-font-face-t>)

Gets the current font face for a <cairo-t>.

cr a <cairo-t>
ret the current font face. This object is owned by cairo. To keep a reference to it, you must call `cairo_font_face_reference`. This function never returns ‘#f’. If memory cannot be allocated, a special "nil" <cairo-font-face-t> object will be returned on which `cairo-font-face-status` returns ‘CAIRO_STATUS_NO_MEMORY’. Using this nil object will cause its error state to propagate to other objects it is passed to, (for example, calling `cairo-set-font-face` with a nil font will trigger an error that will shutdown the `cairo_t` object).

`cairo-set-scaled-font` (*cr* <cairo-t>) [Function]
 (*scaled-font* <cairo-scaled-font-t>)

Replaces the current font face, font matrix, and font options in the <cairo-t> with those of the <cairo-scaled-font-t>. Except for some translation, the current CTM of the <cairo-t> should be the same as that of the <cairo-scaled-font-t>, which can be accessed using `cairo-scaled-font-get-ctm`.

```

cr          a <cairo-t>
scaled-font
              a <cairo-scaled-font-t>

```

Since 1.2

```

cairo-get-scaled-font (cr <cairo-t>) [Function]
  ⇒ (ret <cairo-scaled-font-t>)

```

Gets the current scaled font for a <cairo-t>.

```

cr          a <cairo-t>
ret         the current scaled font. This object is owned by cairo. To keep a reference
              to it, you must call cairo-scaled-font-reference. This function never
              returns '#f'. If memory cannot be allocated, a special "nil" <cairo-
              scaled-font-t> object will be returned on which cairo-scaled-font-
              status returns 'CAIRO_STATUS_NO_MEMORY'. Using this nil object will
              cause its error state to propagate to other objects it is passed to, (for
              example, calling cairo-set-scaled-font with a nil font will trigger an
              error that will shutdown the cairo_t object).

```

Since 1.4

```

cairo-show-text (cr <cairo-t>) (utf8 <char>) [Function]

```

A drawing operator that generates the shape from a string of UTF-8 characters, rendered according to the current *font_face*, *font_size* (*font_matrix*), and *font_options*. This function first computes a set of glyphs for the string of text. The first glyph is placed so that its origin is at the current point. The origin of each subsequent glyph is offset from that of the previous glyph by the advance values of the previous glyph. After this call the current point is moved to the origin of where the next glyph would be placed in this same progression. That is, the current point will be at the origin of the final glyph offset by its advance values. This allows for easy display of a single logical string with multiple calls to **cairo-show-text**.

NOTE: The **cairo-show-text** function call is part of what the cairo designers call the "toy" text API. It is convenient for short demos and simple programs, but it is not expected to be adequate for serious text-using applications. See **cairo-show-glyphs** for the "real" text display API in cairo.

```

cr          a cairo context
utf8        a string of text encoded in UTF-8

```

```

cairo-show-glyphs (cr <cairo-t>) (glyphs <cairo-glyph-t>) [Function]
  (num-glyphs <int>)

```

A drawing operator that generates the shape from an array of glyphs, rendered according to the current *font_face*, *font_size* (*font_matrix*), and *font_options*.

```

cr          a cairo context
glyphs     array of glyphs to show
num-glyphs
              number of glyphs to show

```

`cairo-font-extents` (*cr* <cairo-t>) [Function]
 (*extents* <cairo-font-extents-t>)

Gets the font extents for the currently selected font.

cr a <cairo-t>

extents a <cairo-font-extents-t> object into which the results will be stored.

`cairo-text-extents` (*cr* <cairo-t>) (*utf8* <char>) [Function]
 (*extents* <cairo-text-extents-t>)

Gets the extents for a string of text. The extents describe a user-space rectangle that encloses the "inked" portion of the text, (as it would be drawn by `cairo-show-text`). Additionally, the `x_advance` and `y_advance` values indicate the amount by which the current point would be advanced by `cairo-show-text`.

Note that whitespace characters do not directly contribute to the size of the rectangle (`extents.width` and `extents.height`). They do contribute indirectly by changing the position of non-whitespace characters. In particular, trailing whitespace characters are likely to not affect the size of the rectangle, though they will affect the `x_advance` and `y_advance` values.

cr a <cairo-t>

utf8 a string of text, encoded in UTF-8

extents a <cairo-text-extents-t> object into which the results will be stored

`cairo-glyph-extents` (*cr* <cairo-t>) (*glyphs* <cairo-glyph-t>) [Function]
 (*num-glyphs* <int>) (*extents* <cairo-text-extents-t>)

Gets the extents for an array of glyphs. The extents describe a user-space rectangle that encloses the "inked" portion of the glyphs, (as they would be drawn by `cairo-show-glyphs`). Additionally, the `x_advance` and `y_advance` values indicate the amount by which the current point would be advanced by `cairo-show-glyphs`.

Note that whitespace glyphs do not contribute to the size of the rectangle (`extents.width` and `extents.height`).

cr a <cairo-t>

glyphs an array of <cairo-glyph-t> objects

num-glyphs
 the number of elements in *glyphs*

extents a <cairo-text-extents-t> object into which the results will be stored

6 `cairo_font_face_t`

Base class for fonts

6.1 Overview

6.2 Usage

`cairo-font-face-get-type` (*font-face* `<cairo-font-face-t>`) [Function]

⇒ (*ret* `<cairo-font-type-t>`)

This function returns the type of the backend used to create a font face. See `<cairo-font-type-t>` for available types.

font-face a font face

ret The type of *font-face*.

Since 1.2

7 Scaled Fonts

Caching metrics for a particular font size

7.1 Overview

7.2 Usage

```
cairo-scaled-font-create (font-face <cairo-font-face-t>) [Function]
                        (font-matrix <cairo-matrix-t>) (ctm <cairo-matrix-t>)
                        (options <cairo-font-options-t>) ⇒ (ret <cairo-scaled-font-t>)
```

Creates a <cairo-scaled-font-t> object from a font face and matrices that describe the size of the font and the environment in which it will be used.

font-face a <cairo-font-face-t>

font-matrix

font space to user space transformation matrix for the font. In the simplest case of a N point font, this matrix is just a scale by N, but it can also be used to shear the font or stretch it unequally along the two axes. See `cairo-set-font-matrix`.

ctm user to device transformation matrix with which the font will be used.

options options to use when getting metrics for the font and rendering with it.

ret a newly created <cairo-scaled-font-t>. Destroy with `cairo-scaled-font-destroy`

```
cairo-scaled-font-extents [Function]
                        (scaled-font <cairo-scaled-font-t>)
                        (extents <cairo-font-extents-t>)
```

Gets the metrics for a <cairo-scaled-font-t>.

scaled-font

a <cairo-scaled-font-t>

extents a <cairo-font-extents-t> which to store the retrieved extents.

```
cairo-scaled-font-text-extents [Function]
                        (scaled-font <cairo-scaled-font-t>) (utf8 <char>)
                        (extents <cairo-text-extents-t>)
```

Gets the extents for a string of text. The extents describe a user-space rectangle that encloses the "inked" portion of the text drawn at the origin (0,0) (as it would be drawn by `cairo-show-text` if the cairo graphics state were set to the same `font-face`, `font-matrix`, `ctm`, and `font-options` as *scaled-font*). Additionally, the `x_advance` and `y_advance` values indicate the amount by which the current point would be advanced by `cairo-show-text`.

Note that whitespace characters do not directly contribute to the size of the rectangle (`extents.width` and `extents.height`). They do contribute indirectly by changing the position of non-whitespace characters. In particular, trailing whitespace characters

are likely to not affect the size of the rectangle, though they will affect the `x_advance` and `y_advance` values.

scaled-font

a `<cairo-scaled-font-t>`

utf8 a string of text, encoded in UTF-8

extents a `<cairo-text-extents-t>` which to store the retrieved extents.

Since 1.2

`cairo-scaled-font-glyph-extents` [Function]

(*scaled-font* `<cairo-scaled-font-t>`) (*glyphs* `<cairo-glyph-t>`)

(*num-glyphs* `<int>`) (*extents* `<cairo-text-extents-t>`)

Gets the extents for an array of glyphs. The extents describe a user-space rectangle that encloses the "inked" portion of the glyphs, (as they would be drawn by `cairo-show-glyphs` if the cairo graphics state were set to the same `font_face`, `font_matrix`, `ctm`, and `font_options` as *scaled-font*). Additionally, the `x_advance` and `y_advance` values indicate the amount by which the current point would be advanced by `cairo-show-glyphs`.

Note that whitespace glyphs do not contribute to the size of the rectangle (`extents.width` and `extents.height`).

scaled-font

a `<cairo-scaled-font-t>`

glyphs an array of glyph IDs with X and Y offsets.

num-glyphs

the number of glyphs in the *glyphs* array

extents a `<cairo-text-extents-t>` which to store the retrieved extents.

`cairo-scaled-font-get-font-face` [Function]

(*scaled-font* `<cairo-scaled-font-t>`)

⇒ (*ret* `<cairo-font-face-t>`)

Gets the font face that this scaled font was created for.

scaled-font

a `<cairo-scaled-font-t>`

ret The `<cairo-font-face-t>` with which *scaled-font* was created.

Since 1.2

`cairo-scaled-font-get-font-options` [Function]

(*scaled-font* `<cairo-scaled-font-t>`)

(*options* `<cairo-font-options-t>`)

Stores the font options with which *scaled-font* was created into *options*.

scaled-font

a `<cairo-scaled-font-t>`

options return value for the font options

Since 1.2

`cairo-scaled-font-get-font-matrix` [Function]

```
(scaled-font <cairo-scaled-font-t>)
(font-matrix <cairo-matrix-t>)
```

Stores the font matrix with which *scaled-font* was created into *matrix*.

scaled-font

a <cairo-scaled-font-t>

font-matrix

return value for the matrix

Since 1.2

`cairo-scaled-font-get-ctm` [Function]

```
(scaled-font <cairo-scaled-font-t>) (ctm <cairo-matrix-t>)
```

Stores the CTM with which *scaled-font* was created into *ctm*.

scaled-font

a <cairo-scaled-font-t>

ctm

return value for the CTM

Since 1.2

`cairo-scaled-font-get-type` [Function]

```
(scaled-font <cairo-scaled-font-t>)
⇒ (ret <cairo-font-type-t>)
```

This function returns the type of the backend used to create a scaled font. See <cairo-font-type-t> for available types.

scaled-font

a <cairo-scaled-font-t>

ret

The type of *scaled-font*.

Since 1.2

8 Font Options

How a font should be rendered

8.1 Overview

8.2 Usage

`cairo-font-options-create` ⇒ (`ret` <cairo-font-options-t>) [Function]

Allocates a new font options object with all options initialized to default values.

ret a newly allocated <cairo-font-options-t>. Free with `cairo-font-options-destroy`. This function always returns a valid pointer; if memory cannot be allocated, then a special error object is returned where all operations on the object do nothing. You can check for this with `cairo-font-options-status`.

`cairo-font-options-copy` (*original* <cairo-font-options-t>) [Function]
⇒ (`ret` <cairo-font-options-t>)

Allocates a new font options object copying the option values from *original*.

original a <cairo-font-options-t>

ret a newly allocated <cairo-font-options-t>. Free with `cairo-font-options-destroy`. This function always returns a valid pointer; if memory cannot be allocated, then a special error object is returned where all operations on the object do nothing. You can check for this with `cairo-font-options-status`.

`cairo-font-options-merge` (*options* <cairo-font-options-t>) [Function]
(*other* <cairo-font-options-t>)

Merges non-default options from *other* into *options*, replacing existing values. This operation can be thought of as somewhat similar to compositing *other* onto *options* with the operation of ‘CAIRO_OPERATION_OVER’.

options a <cairo-font-options-t>

other another <cairo-font-options-t>

`cairo-font-options-hash` (*options* <cairo-font-options-t>) [Function]
⇒ (`ret` <unsigned long>)

Compute a hash for the font options object; this value will be useful when storing an object containing a `cairo_font_options_t` in a hash table.

options a <cairo-font-options-t>

ret the hash value for the font options object. The return value can be cast to a 32-bit type if a 32-bit hash value is needed.

- `cairo-font-options-set-antialias` [Function]
 (`options` <cairo-font-options-t>)
 (`antialias` <cairo-antialias-t>)
 Sets the antialiasing mode for the font options object. This specifies the type of antialiasing to do when rendering text.
- `options` a <cairo-font-options-t>
`antialias` the new antialiasing mode
- `cairo-font-options-get-antialias` [Function]
 (`options` <cairo-font-options-t>) ⇒ (`ret` <cairo-antialias-t>)
 Gets the antialiasing mode for the font options object.
- `options` a <cairo-font-options-t>
`ret` the antialiasing mode
- `cairo-font-options-set-hint-style` [Function]
 (`options` <cairo-font-options-t>)
 (`hint-style` <cairo-hint-style-t>)
 Sets the hint style for font outlines for the font options object. This controls whether to fit font outlines to the pixel grid, and if so, whether to optimize for fidelity or contrast. See the documentation for <cairo-hint-style-t> for full details.
- `options` a <cairo-font-options-t>
`hint-style` the new hint style
- `cairo-font-options-get-hint-style` [Function]
 (`options` <cairo-font-options-t>) ⇒ (`ret` <cairo-hint-style-t>)
 Gets the hint style for font outlines for the font options object. See the documentation for <cairo-hint-style-t> for full details.
- `options` a <cairo-font-options-t>
`ret` the hint style for the font options object
- `cairo-font-options-set-hint-metrics` [Function]
 (`options` <cairo-font-options-t>)
 (`hint-metrics` <cairo-hint-metrics-t>)
 Sets the metrics hinting mode for the font options object. This controls whether metrics are quantized to integer values in device units. See the documentation for <cairo-hint-metrics-t> for full details.
- `options` a <cairo-font-options-t>
`hint-metrics`
 the new metrics hinting mode

9 FreeType Fonts

Font support for FreeType

9.1 Overview

9.2 Usage

10 Win32 Fonts

Font support for Microsoft Windows

10.1 Overview

10.2 Usage

11 `cairo_surface_t`

Base class for surfaces

11.1 Overview

11.2 Usage

`cairo-surface-create-similar` (*other* <cairo-surface-t>) [Function]
 (*content* <cairo-content-t>) (*width* <int>) (*height* <int>)
 ⇒ (*ret* <cairo-surface-t>)

Create a new surface that is as compatible as possible with an existing surface. For example the new surface will have the same fallback resolution and font options as *other*. Generally, the new surface will also use the same backend as *other*, unless that is not possible for some reason. The type of the returned surface may be examined with `cairo-surface-get-type`.

Initially the surface contents are all 0 (transparent if contents have transparency, black otherwise.)

other an existing surface used to select the backend of the new surface
content the content for the new surface
width width of the new surface, (in device-space units)
height height of the new surface (in device-space units)
ret a pointer to the newly allocated surface. The caller owns the surface and should call `cairo_surface_destroy` when done with it. This function always returns a valid pointer, but it will return a pointer to a "nil" surface if *other* is already in an error state or any other error occurs.

`cairo-surface-finish` (*surface* <cairo-surface-t>) [Function]

This function finishes the surface and drops all references to external resources. For example, for the Xlib backend it means that cairo will no longer access the drawable, which can be freed. After calling `cairo-surface-finish` the only valid operations on a surface are getting and setting user data and referencing and destroying it. Further drawing to the surface will not affect the surface but will instead trigger a `CAIRO_STATUS_SURFACE_FINISHED` error.

When the last call to `cairo-surface-destroy` decreases the reference count to zero, cairo will call `cairo-surface-finish` if it hasn't been called already, before freeing the resources associated with the surface.

surface the <cairo-surface-t> to finish

`cairo-surface-flush` (*surface* <cairo-surface-t>) [Function]

Do any pending drawing for the surface and also restore any temporary modification's cairo has made to the surface's state. This function must be called before switching from drawing on the surface with cairo to drawing on it directly with native APIs. If the surface doesn't support direct access, then this function does nothing.

surface a <cairo-surface-t>

`cairo_surface_get_font_options` (*surface* <cairo-surface-t>) [Function]
 (*options* <cairo-font-options-t>)

Retrieves the default font rendering options for the surface. This allows display surfaces to report the correct subpixel order for rendering on them, print surfaces to disable hinting of metrics and so forth. The result can then be used with `cairo_scaled_font_create`.

surface a <cairo-surface-t>

options a <cairo-font-options-t> object into which to store the retrieved options. All existing values are overwritten

`cairo_surface_get_content` (*surface* <cairo-surface-t>) [Function]
 ⇒ (*ret* <cairo-content-t>)

This function returns the content type of *surface* which indicates whether the surface contains color and/or alpha information. See <cairo-content-t>.

surface a <cairo-surface-t>

ret The content type of *surface*.

Since 1.2

`cairo_surface_mark_dirty` (*surface* <cairo-surface-t>) [Function]

Tells cairo that drawing has been done to surface using means other than cairo, and that cairo should reread any cached areas. Note that you must call `cairo_surface_flush` before doing such drawing.

surface a <cairo-surface-t>

`cairo_surface_mark_dirty_rectangle` [Function]
 (*surface* <cairo-surface-t>) (*x* <int>) (*y* <int>) (*width* <int>)
 (*height* <int>)

Like `cairo_surface_mark_dirty`, but drawing has been done only to the specified rectangle, so that cairo can retain cached contents for other parts of the surface.

Any cached clip set on the surface will be reset by this function, to make sure that future cairo calls have the clip set that they expect.

surface a <cairo-surface-t>

x X coordinate of dirty rectangle

y Y coordinate of dirty rectangle

width width of dirty rectangle

height height of dirty rectangle

`cairo_surface_set_device_offset` (*surface* <cairo-surface-t>) [Function]
 (*x-offset* <double>) (*y-offset* <double>)

Sets an offset that is added to the device coordinates determined by the CTM when drawing to *surface*. One use case for this function is when we want to create a <cairo-surface-t> that redirects drawing for a portion of an onscreen surface to an offscreen surface in a way that is completely invisible to the user of the cairo API. Setting a

transformation via `cairo-translate` isn't sufficient to do this, since functions like `cairo-device-to-user` will expose the hidden offset.

Note that the offset affects drawing to the surface as well as using the surface in a source pattern.

surface a `<cairo-surface-t>`
x-offset the offset in the X direction, in device units
y-offset the offset in the Y direction, in device units

`cairo-surface-get-device-offset` (*surface* `<cairo-surface-t>`) [Function]
 \Rightarrow (*x-offset* `<double>`) (*y-offset* `<double>`)

This function returns the previous device offset set by `cairo-surface-set-device-offset`.

surface a `<cairo-surface-t>`
x-offset the offset in the X direction, in device units
y-offset the offset in the Y direction, in device units

Since 1.2

`cairo-surface-get-type` (*surface* `<cairo-surface-t>`) [Function]
 \Rightarrow (*ret* `<cairo-surface-type-t>`)

This function returns the type of the backend used to create a surface. See `<cairo-surface-type-t>` for available types.

surface a `<cairo-surface-t>`
ret The type of *surface*.

Since 1.2

12 Image Surfaces

Rendering to memory buffers

12.1 Overview

Image surfaces provide the ability to render to memory buffers either allocated by cairo or by the calling code. The supported image formats are those defined in `<cairo-format-t>`.

12.2 Usage

`cairo-image-surface-create` (*format* `<cairo-format-t>`) [Function]
 (*width* `<int>`) (*height* `<int>`) ⇒ (*ret* `<cairo-surface-t>`)

Creates an image surface of the specified format and dimensions. Initially the surface contents are all 0. (Specifically, within each pixel, each color or alpha channel belonging to format will be 0. The contents of bits within a pixel, but not belonging to the given format are undefined).

format format of pixels in the surface to create

width width of the surface, in pixels

height height of the surface, in pixels

ret a pointer to the newly created surface. The caller owns the surface and should call `cairo_surface_destroy` when done with it. This function always returns a valid pointer, but it will return a pointer to a "nil" surface if an error such as out of memory occurs. You can use `cairo_surface_status` to check for this.

`cairo-image-surface-get-format` (*surface* `<cairo-surface-t>`) [Function]
 ⇒ (*ret* `<cairo-format-t>`)

Get the format of the surface.

surface a `<cairo-image-surface-t>`

ret the format of the surface

Since 1.2

`cairo-image-surface-get-width` (*surface* `<cairo-surface-t>`) [Function]
 ⇒ (*ret* `<int>`)

Get the width of the image surface in pixels.

surface a `<cairo-image-surface-t>`

ret the width of the surface in pixels.

`cairo-image-surface-get-height` (*surface* `<cairo-surface-t>`) [Function]
 ⇒ (*ret* `<int>`)

Get the height of the image surface in pixels.

surface a `<cairo-image-surface-t>`

ret the height of the surface in pixels.

`cairo-image-surface-get-stride` (*surface* <cairo-surface-t>) [Function]
⇒ (*ret* <int>)

Get the stride of the image surface in bytes

surface a <cairo-image-surface-t>

ret the stride of the image surface in bytes (or 0 if *surface* is not an image surface). The stride is the distance in bytes from the beginning of one row of the image data to the beginning of the next row.

Since 1.2

13 PDF Surfaces

Rendering PDF documents

13.1 Overview

13.2 Usage

`cairo-pdf-surface-create` (*filename* <char>) [Function]
 (*width-in-points* <double>) (*height-in-points* <double>)
 ⇒ (*ret* <cairo-surface-t>)

Creates a PDF surface of the specified size in points to be written to *filename*.

filename a filename for the PDF output (must be writable)

width-in-points

width of the surface, in points (1 point == 1/72.0 inch)

height-in-points

height of the surface, in points (1 point == 1/72.0 inch)

ret

a pointer to the newly created surface. The caller owns the surface and should call `cairo_surface_destroy` when done with it. This function always returns a valid pointer, but it will return a pointer to a "nil" surface if an error such as out of memory occurs. You can use `cairo-surface-status` to check for this.

Since 1.2

`cairo-pdf-surface-set-size` (*surface* <cairo-surface-t>) [Function]
 (*width-in-points* <double>) (*height-in-points* <double>)

Changes the size of a PDF surface for the current (and subsequent) pages.

This function should only be called before any drawing operations have been performed on the current page. The simplest way to do this is to call this function immediately after creating the surface or immediately after completing a page with either `cairo-show-page` or `cairo-copy-page`.

surface a PDF `cairo_surface_t`

width-in-points

new surface width, in points (1 point == 1/72.0 inch)

height-in-points

new surface height, in points (1 point == 1/72.0 inch)

Since 1.2

14 PNG Support

Reading and writing PNG images

14.1 Overview

14.2 Usage

`cairo-image-surface-create-from-png` (*filename* <char>) [Function]
 ⇒ (*ret* <cairo-surface-t>)

Creates a new image surface and initializes the contents to the given PNG file.

filename name of PNG file to load

ret a new <cairo-surface-t> initialized with the contents of the PNG file, or a "nil" surface if any error occurred. A nil surface can be checked for with `cairo_surface_status(surface)` which may return one of the following values: `CAIRO_STATUS_NO_MEMORY` `CAIRO_STATUS_FILE_NOT_FOUND` `CAIRO_STATUS_READ_ERROR`■

`cairo-surface-write-to-png` (*surface* <cairo-surface-t>) [Function]
 (*filename* <char>) ⇒ (*ret* <cairo-status-t>)

Writes the contents of *surface* to a new file *filename* as a PNG image.

surface a <cairo-surface-t> with pixel contents

filename the name of a file to write to

ret `CAIRO_STATUS_SUCCESS` if the PNG file was written successfully. Otherwise, `CAIRO_STATUS_NO_MEMORY` if memory could not be allocated for the operation or `CAIRO_STATUS_SURFACE_TYPE_MISMATCH` if the surface does not have pixel contents, or `CAIRO_STATUS_WRITE_ERROR` if an I/O error occurs while attempting to write the file.

15 PostScript Surfaces

Rendering PostScript documents

15.1 Overview

15.2 Usage

`cairo-ps-surface-create` (*filename* <char>) [Function]
 (*width-in-points* <double>) (*height-in-points* <double>)
 ⇒ (*ret* <cairo-surface-t>)

Creates a PostScript surface of the specified size in points to be written to *filename*. See `cairo-ps-surface-create-for-stream` for a more flexible mechanism for handling the PostScript output than simply writing it to a named file.

Note that the size of individual pages of the PostScript output can vary. See `cairo-ps-surface-set-size`.

filename a filename for the PS output (must be writable)

width-in-points
width of the surface, in points (1 point == 1/72.0 inch)

height-in-points
height of the surface, in points (1 point == 1/72.0 inch)

ret a pointer to the newly created surface. The caller owns the surface and should call `cairo_surface_destroy` when done with it. This function always returns a valid pointer, but it will return a pointer to a "nil" surface if an error such as out of memory occurs. You can use `cairo-surface-status` to check for this.

Since 1.2

`cairo-ps-surface-set-size` (*surface* <cairo-surface-t>) [Function]
 (*width-in-points* <double>) (*height-in-points* <double>)

Changes the size of a PostScript surface for the current (and subsequent) pages.

This function should only be called before any drawing operations have been performed on the current page. The simplest way to do this is to call this function immediately after creating the surface or immediately after completing a page with either `cairo-show-page` or `cairo-copy-page`.

surface a PostScript `cairo_surface_t`

width-in-points
new surface width, in points (1 point == 1/72.0 inch)

height-in-points
new surface height, in points (1 point == 1/72.0 inch)

Since 1.2

`cairo-ps-surface-dsc-comment` (*surface* <cairo-surface-t>) [Function]
 (*comment* <char>)

Emit a comment into the PostScript output for the given surface.

The comment is expected to conform to the PostScript Language Document Structuring Conventions (DSC). Please see that manual for details on the available comments and their meanings. In particular, the `%‘IncludeFeature’` comment allows a device-independent means of controlling printer device features. So the PostScript Printer Description Files Specification will also be a useful reference.

The comment string must begin with a percent character (`%`) and the total length of the string (including any initial percent characters) must not exceed 255 characters. Violating either of these conditions will place *surface* into an error state. But beyond these two conditions, this function will not enforce conformance of the comment with any particular specification.

The comment string should not have a trailing newline.

The DSC specifies different sections in which particular comments can appear. This function provides for comments to be emitted within three sections: the header, the Setup section, and the PageSetup section. Comments appearing in the first two sections apply to the entire document while comments in the `BeginPageSetup` section apply only to a single page.

For comments to appear in the header section, this function should be called after the surface is created, but before a call to `cairo-ps-surface-begin-setup`.

For comments to appear in the Setup section, this function should be called after a call to `cairo-ps-surface-begin-setup` but before a call to `cairo-ps-surface-begin-page-setup`.

For comments to appear in the PageSetup section, this function should be called after a call to `cairo-ps-surface-begin-page-setup`.

Note that it is only necessary to call `cairo-ps-surface-begin-page-setup` for the first page of any surface. After a call to `cairo-show-page` or `cairo-copy-page` comments are unambiguously directed to the PageSetup section of the current page. But it doesn't hurt to call this function at the beginning of every page as that consistency may make the calling code simpler.

As a final note, cairo automatically generates several comments on its own. As such, applications must not manually generate any of the following comments:

Header section: `%!PS-Adobe-3.0, %‘Creator’, %‘CreationDate’, %‘Pages’, %‘BoundingBox’, %‘DocumentData’, %‘LanguageLevel’, %‘EndComments’.`

Setup section: `%‘BeginSetup’, %‘EndSetup’`

PageSetup section: `%‘BeginPageSetup’, %‘PageBoundingBox’, %‘EndPageSetup’.`

Other sections: `%‘BeginProlog’, %‘EndProlog’, %‘Page’, %‘Trailer’, %‘EOF’`

Here is an example sequence showing how this function might be used:

```
cairo_surface_t *surface = cairo_ps_surface_create (filename, width, height);
...
cairo_ps_surface_dsc_comment (surface, "%Title: My excellent document");
```

```

cairo_ps_surface_dsc_comment (surface, "%%Copyright: Copyright (C) 2006 Cairo Lov
...
cairo_ps_surface_dsc_begin_setup (surface);
cairo_ps_surface_dsc_comment (surface, "%%IncludeFeature: *MediaColor White");█
...
cairo_ps_surface_dsc_begin_page_setup (surface);
cairo_ps_surface_dsc_comment (surface, "%%IncludeFeature: *PageSize A3");█
cairo_ps_surface_dsc_comment (surface, "%%IncludeFeature: *InputSlot LargeCapacit
cairo_ps_surface_dsc_comment (surface, "%%IncludeFeature: *MediaType Glossy");█
cairo_ps_surface_dsc_comment (surface, "%%IncludeFeature: *MediaColor Blue");█
... draw to first page here ..
cairo_show_page (cr);
...
cairo_ps_surface_dsc_comment (surface, "%%IncludeFeature: *PageSize A5");█
...

```

surface a PostScript cairo_surface_t

comment a comment string to be emitted into the PostScript output

Since 1.2

16 SVG Surfaces

Rendering SVG documents

16.1 Overview

16.2 Usage

```
cairo-svg-surface-create (filename <char>) [Function]  
    (width-in-points <double>) (height-in-points <double>)  
    ⇒ (ret <cairo-surface-t>)
```

Creates a SVG surface of the specified size in points to be written to *filename*.

filename a filename for the SVG output (must be writable)

width-in-points

width of the surface, in points (1 point == 1/72.0 inch)

height-in-points

height of the surface, in points (1 point == 1/72.0 inch)

ret

a pointer to the newly created surface. The caller owns the surface and should call `cairo_surface_destroy` when done with it. This function always returns a valid pointer, but it will return a pointer to a "nil" surface if an error such as out of memory occurs. You can use `cairo_surface_status` to check for this.

Since 1.2

17 `cairo_matrix_t`

Generic matrix operations

17.1 Overview

`<cairo-matrix-t>` is used throughout `cairo` to convert between different coordinate spaces. A `<cairo-matrix-t>` holds an affine transformation, such as a scale, rotation, shear, or a combination of these. The transformation of a point ('x','y') is given by:

$$\begin{aligned}x_{\text{new}} &= xx * x + xy * y + x0; \\y_{\text{new}} &= yx * x + yy * y + y0;\end{aligned}$$

The current transformation matrix of a `<cairo-t>`, represented as a `<cairo-matrix-t>`, defines the transformation from user-space coordinates to device-space coordinates. See `cairo-get-matrix` and `cairo-set-matrix`.

17.2 Usage

`cairo-matrix-translate` (*matrix* `<cairo-matrix-t>`) [Function]
(*tx* `<double>`) (*ty* `<double>`)

Applies a translation by *tx*, *ty* to the transformation in *matrix*. The effect of the new transformation is to first translate the coordinates by *tx* and *ty*, then apply the original transformation to the coordinates.

matrix a `cairo_matrix_t`
tx amount to translate in the X direction
ty amount to translate in the Y direction

`cairo-matrix-invert` (*matrix* `<cairo-matrix-t>`) [Function]
⇒ (*ret* `<cairo-status-t>`)

Changes *matrix* to be the inverse of it's original value. Not all transformation matrices have inverses; if the matrix collapses points together (it is *degenerate*), then it has no inverse and this function will fail.

Returns: If *matrix* has an inverse, modifies *matrix* to be the inverse matrix and returns 'CAIRO_STATUS_SUCCESS'. Otherwise,

matrix a `<cairo-matrix-t>`
ret 'CAIRO_STATUS_INVALID_MATRIX'.

`cairo-matrix-multiply` (*result* `<cairo-matrix-t>`) [Function]
(*a* `<cairo-matrix-t>`) (*b* `<cairo-matrix-t>`)

Multiplies the affine transformations in *a* and *b* together and stores the result in *result*. The effect of the resulting transformation is to first apply the transformation in *a* to the coordinates and then apply the transformation in *b* to the coordinates.

It is allowable for *result* to be identical to either *a* or *b*.

result a <cairo-matrix-t> in which to store the result
a a <cairo-matrix-t>
b a <cairo-matrix-t>

cairo-matrix-transform-distance (*matrix* <cairo-matrix-t>) [Function]
 ⇒ (*dx* <double>) (*dy* <double>)

Transforms the distance vector (*dx,dy*) by *matrix*. This is similar to **cairo-matrix-transform-point** except that the translation components of the transformation are ignored. The calculation of the returned vector is as follows:

$$\begin{aligned} dx2 &= dx1 * a + dy1 * c; \\ dy2 &= dx1 * b + dy1 * d; \end{aligned}$$

Affine transformations are position invariant, so the same vector always transforms to the same vector. If (*x1,y1*) transforms to (*x2,y2*) then (*x1+dx1,y1+dy1*) will transform to (*x1+dx2,y1+dy2*) for all values of *x1* and *x2*.

matrix a <cairo-matrix-t>
dx X component of a distance vector. An in/out parameter
dy Y component of a distance vector. An in/out parameter

cairo-matrix-transform-point (*matrix* <cairo-matrix-t>) [Function]
 ⇒ (*x* <double>) (*y* <double>)

Transforms the point (*x, y*) by *matrix*.

matrix a <cairo-matrix-t>
x X position. An in/out parameter
y Y position. An in/out parameter

18 Error handling

Decoding cairo's status

18.1 Overview

18.2 Usage

19 Version Information

Compile-time and run-time version checks.

19.1 Overview

Cairo has a three-part version number scheme. In this scheme, we use even vs. odd numbers to distinguish fixed points in the software vs. in-progress development, (such as from git instead of a tar file, or as a "snapshot" tar file as opposed to a "release" tar file).

```

    _____ Major. Always 1, until we invent a new scheme.
  /   ___ Minor. Even/Odd = Release/Snapshot (tar files) or Branch/Head (git)
 | /  _ Micro. Even/Odd = Tar-file/git
 | | /
1.0.0

```

Here are a few examples of versions that one might see.

Releases

```

1.0.0 - A major release
1.0.2 - A subsequent maintenance release
1.2.0 - Another major release

```

Snapshots

```

1.1.2 - A snapshot (working toward the 1.2.0 release)

```

In-progress development (eg. from git)

```

1.0.1 - Development on a maintenance branch (toward 1.0.2 release)
1.1.1 - Development on head (toward 1.1.2 snapshot and 1.2.0 release)

```

19.2 Compatibility

The API/ABI compatibility guarantees for various versions are as follows. First, let's assume some cairo-using application code that is successfully using the API/ABI "from" one version of cairo. Then let's ask the question whether this same code can be moved "to" the API/ABI of another version of cairo.

Moving from a release to any later version (release, snapshot, development) is always guaranteed to provide compatibility.

Moving from a snapshot to any later version is not guaranteed to provide compatibility, since snapshots may introduce new API that ends up being removed before the next release.

Moving from an in-development version (odd micro component) to any later version is not guaranteed to provide compatibility. In fact, there's not even a guarantee that the code will even continue to work with the same in-development version number. This is because these numbers don't correspond to any fixed state of the software, but rather the many states between snapshots and releases.

19.3 Examining the version

Cairo provides the ability to examine the version at either compile-time or run-time and in both a human-readable form as well as an encoded form suitable for direct comparison. Cairo also provides a macro (`CAIRO_VERSION_ENCODE`) to perform the encoding.

```
Compile-time
-----
%CAIRO_VERSION_STRING Human-readable
%CAIRO_VERSION Encoded, suitable for comparison
```

```
Run-time
-----
cairo_version_string() Human-readable
cairo_version() Encoded, suitable for comparison
```

For example, checking that the cairo version is greater than or equal to 1.0.0 could be achieved at compile-time or run-time as follows:

```
##if %CAIRO_VERSION >= %CAIRO_VERSION_ENCODE(1, 0, 0)
printf ("Compiling with suitable cairo version: %s\n", CAIRO_VERSION_STRING);
##endif

if (cairo_version() >= %CAIRO_VERSION_ENCODE(1, 0, 0))
    printf ("Running with suitable cairo version: %s\n", cairo_version_string ());
```

19.4 Usage

`cairo-version` ⇒ (`ret` <int>) [Function]

Returns the version of the cairo library encoded in a single integer as per `CAIRO_VERSION_ENCODE`. The encoding ensures that later versions compare greater than earlier versions.

A run-time comparison to check that cairo's version is greater than or equal to version X.Y.Z could be performed as follows:

```
if (cairo_version() >= CAIRO_VERSION_ENCODE(X,Y,Z)) {...}
```

See also `cairo-version-string` as well as the compile-time equivalents `'CAIRO_VERSION'` and `'CAIRO_VERSION_STRING'`.

`ret` the encoded version.

`cairo-version-string` ⇒ (`ret` <char>) [Function]

Returns the version of the cairo library as a human-readable string of the form "X.Y.Z".

See also `cairo-version` as well as the compile-time equivalents `'CAIRO_VERSION_STRING'` and `'CAIRO_VERSION'`.

`ret` a string containing the version.

20 Types

Generic data types used in the cairo API

20.1 Overview

20.2 Usage

Concept Index

types, cairo_matrix 53

Function Index

cairo-append-path	15	cairo-image-surface-get-width	45
cairo-arc	16	cairo-in-fill	11
cairo-arc-negative	17	cairo-in-stroke	12
cairo-clip	9	cairo-line-to	17
cairo-clip- extents	9	cairo-mask	11
cairo-clip-preserve	9	cairo-mask-surface	11
cairo-close-path	15	cairo-matrix-invert	53
cairo-copy-clip-rectangle-list	10	cairo-matrix-multiply	53
cairo-copy-page	13	cairo-matrix-transform-distance	54
cairo-copy-path	14	cairo-matrix-transform-point	54
cairo-copy-path-flat	14	cairo-matrix-translate	53
cairo-create	1	cairo-move-to	18
cairo-curve-to	17	cairo-new-path	15
cairo-device-to-user	29	cairo-new-sub-path	15
cairo-device-to-user-distance	29	cairo-paint	11
cairo-fill	10	cairo-paint-with-alpha	11
cairo-fill- extents	10	cairo-pattern-add-color-stop-rgb	21
cairo-fill-preserve	10	cairo-pattern-add-color-stop-rgba	21
cairo-font- extents	33	cairo-pattern-create-for-surface	23
cairo-font-face-get-type	34	cairo-pattern-create-linear	24
cairo-font-options-copy	38	cairo-pattern-create-radial	24
cairo-font-options-create	38	cairo-pattern-create-rgb	22
cairo-font-options-get-antialias	39	cairo-pattern-create-rgba	22
cairo-font-options-get-hint-style	39	cairo-pattern-get-color-stop-rgba	22
cairo-font-options-hash	38	cairo-pattern-get-extend	26
cairo-font-options-merge	38	cairo-pattern-get-filter	26
cairo-font-options-set-antialias	39	cairo-pattern-get-linear-points	24
cairo-font-options-set-hint-metrics	39	cairo-pattern-get-matrix	27
cairo-font-options-set-hint-style	39	cairo-pattern-get-radial-circles	25
cairo-get-antialias	5	cairo-pattern-get-rgba	23
cairo-get-current-point	15	cairo-pattern-get-surface	23
cairo-get-dash-count	6	cairo-pattern-get-type	27
cairo-get-fill-rule	6	cairo-pattern-set-extend	25
cairo-get-font-face	31	cairo-pattern-set-filter	26
cairo-get-font-matrix	30	cairo-pattern-set-matrix	26
cairo-get-font-options	31	cairo-pdf-surface-create	47
cairo-get-group-target	3	cairo-pdf-surface-set-size	47
cairo-get-line-cap	6	cairo-pop-group	2
cairo-get-line-join	7	cairo-pop-group-to-source	3
cairo-get-line-width	7	cairo-ps-surface-create	49
cairo-get-matrix	29	cairo-ps-surface-dsc-comment	50
cairo-get-miter-limit	8	cairo-ps-surface-set-size	49
cairo-get-operator	8	cairo-push-group	2
cairo-get-scaled-font	32	cairo-rectangle	18
cairo-get-source	5	cairo-rel-curve-to	19
cairo-get-target	1	cairo-rel-line-to	19
cairo-get-tolerance	8	cairo-rel-move-to	20
cairo-glyph- extents	33	cairo-reset-clip	9
cairo-glyph-path	18	cairo-restore	1
cairo-identity-matrix	29	cairo-rotate	28
cairo-image-surface-create	45	cairo-save	1
cairo-image-surface-create-from-png	48	cairo-scale	28
cairo-image-surface-get-format	45	cairo-scaled-font-create	35
cairo-image-surface-get-height	45	cairo-scaled-font- extents	35
cairo-image-surface-get-stride	46	cairo-scaled-font-get-ctm	37

cairo-scaled-font-get-font-face.....	36	cairo-show-glyphs.....	32
cairo-scaled-font-get-font-matrix.....	37	cairo-show-page.....	13
cairo-scaled-font-get-font-options.....	36	cairo-show-text.....	32
cairo-scaled-font-get-type.....	37	cairo-stroke.....	12
cairo-scaled-font-glyph-extents.....	36	cairo-stroke-extents.....	12
cairo-scaled-font-text-extents.....	35	cairo-stroke-preserve.....	12
cairo-select-font-face.....	30	cairo-surface-create-similar.....	42
cairo-set-antialias.....	5	cairo-surface-finish.....	42
cairo-set-dash.....	5	cairo-surface-flush.....	42
cairo-set-fill-rule.....	6	cairo-surface-get-content.....	43
cairo-set-font-face.....	31	cairo-surface-get-device-offset.....	44
cairo-set-font-matrix.....	30	cairo-surface-get-font-options.....	43
cairo-set-font-options.....	31	cairo-surface-get-type.....	44
cairo-set-font-size.....	30	cairo-surface-mark-dirty.....	43
cairo-set-line-cap.....	6	cairo-surface-mark-dirty-rectangle.....	43
cairo-set-line-join.....	7	cairo-surface-set-device-offset.....	43
cairo-set-line-width.....	7	cairo-surface-write-to-png.....	48
cairo-set-matrix.....	28	cairo-svg-surface-create.....	52
cairo-set-miter-limit.....	7	cairo-text-extents.....	33
cairo-set-operator.....	8	cairo-text-path.....	18
cairo-set-scaled-font.....	31	cairo-transform.....	28
cairo-set-source.....	4	cairo-translate.....	28
cairo-set-source-rgb.....	3	cairo-user-to-device.....	29
cairo-set-source-rgba.....	4	cairo-user-to-device-distance.....	29
cairo-set-source-surface.....	4	cairo-version.....	57
cairo-set-tolerance.....	8	cairo-version-string.....	57