Guile-Cairo

version 1.11.2, updated 19 November 2020

Carl Worth
Andy Wingo
(many others)

Short Contents

1 cairo_t

The cairo drawing context

1.1 Overview

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

`<cairo>`'s can be pushed to a stack via `cairo-save`. They may then safely be changed, without loosing 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 their 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 current destination surface for the context. This is either the original target surface as passed to `cairo-create` or 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 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`.

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.

The default source pattern is opaque black, (that is, it is equivalent to cairo_set_source_rgb(cr, 0.0, 0.0, 0.0)).

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>`) (*green* `<double>`) (*blue* `<double>`) (*alpha* `<double>`) [Function]

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.

The default source pattern is opaque black, (that is, it is equivalent to cairo_set_source_rgba(cr, 0.0, 0.0, 0.0, 1.0)).

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`.

The default source pattern is a solid pattern that is opaque black, (that is, it is equivalent to cairo_set_source_rgb(cr, 0.0, 0.0, 0.0)).

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

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* parameters 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 *cr* 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.

The default fill rule is '`CAIRO_FILL_RULE_WINDING`'.

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.

The default line cap style is '`CAIRO_LINE_CAP_BUTT`'.

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

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.

The default line join style is '`CAIRO_LINE_JOIN_MITER`'.

cr a cairo context

line-join a line join 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.

The default miter limit value is 10.0, which will convert joins with interior angles less than 11 degrees to bevels instead of miters. For reference, a miter limit of 2.0 makes the miter cutoff at 60 degrees, and a miter limit of 1.414 makes the cutoff at 90 degrees.

A miter limit for a desired angle can be computed as: miter limit = 1/sin(angle/2)

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.

The default operator is 'CAIRO_OPERATOR_OVER'.

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.) The accuracy of paths within Cairo is limited by the precision of its internal arithmetic, and the prescribed *tolerance* is restricted to the smallest representable internal value.

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

Tests whether the given point is inside the area that would be visible through the current clip, i.e. the area that would be filled by a `cairo-paint` operation.

See `cairo-clip`, and `cairo-clip-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.

Since 1.10

`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.

Note that code meant to be reusable should not call `cairo-reset-clip` as it will cause results unexpected by higher-level code which calls `cairo-clip`. Consider using `cairo-save` and `cairo-restore` around `cairo-clip` as a more robust means of temporarily restricting the clip region.

cr a cairo context

`cairo-copy-clip-rectangle-list` (*cr* `<cairo-t>`) [Function]
⇒ (*ret* `<cairo-rectangle-list-t >`)

Gets the current clip region as a list of rectangles in user coordinates. Never returns '`#f`'.

The status in the list may be '`CAIRO_STATUS_CLIP_NOT_REPRESENTABLE`' to indicate that the clip region cannot be represented as a list of user-space rectangles. The status may have other values to indicate other errors.

cr a cairo context

ret the current clip region as a list of rectangles in user coordinates, which should be destroyed using `cairo-rectangle-list-destroy`.

Since 1.4

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

A drawing operator that fills the current path according to the current fill rule, (each sub-path is implicitly closed before being filled). After `cairo-fill`, the current path will be cleared from the cairo context. See `cairo-set-fill-rule` and `cairo-fill-preserve`.

cr a cairo context

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

A drawing operator that fills the current path according to the current fill rule, (each sub-path is implicitly closed before being filled). Unlike `cairo-fill`, `cairo-fill-preserve` preserves the path within the cairo context.

See `cairo-set-fill-rule` and `cairo-fill`.

cr a cairo context

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

Computes a bounding box in user coordinates covering the area that would be affected, (the "inked" area), by a `cairo-fill` operation given the current path and fill parameters. If the current path is empty, returns an empty rectangle ((0,0), (0,0)). Surface dimensions and clipping are not taken into account.

Contrast with `cairo-path-extents`, which is similar, but returns non-zero extents for some paths with no inked area, (such as a simple line segment).

Note that `cairo-fill-extents` must necessarily do more work to compute the precise inked areas in light of the fill rule, so `cairo-path-extents` may be more desirable for sake of performance if the non-inked path extents are desired.

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

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

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>`) (*surface* `<cairo-surface-t>`) [Function]
(*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>) (*y1* <double>) (*x2* <double>) (*y2* <double>) [Function]

Computes a bounding box in user coordinates covering the area that would be affected, (the "inked" area), by a `cairo-stroke` operation given the current path and stroke parameters. If the current path is empty, returns an empty rectangle ((0,0), (0,0)). Surface dimensions and clipping are not taken into account.

Note that if the line width is set to exactly zero, then `cairo-stroke-extents` will return an empty rectangle. Contrast with `cairo-path-extents` which can be used to compute the non-empty bounds as the line width approaches zero.

Note that `cairo-stroke-extents` must necessarily do more work to compute the precise inked areas in light of the stroke parameters, so `cairo-path-extents` may be more desirable for sake of performance if non-inked path extents are desired.

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

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>) ⇒ (*ret* <cairo-bool-t>) [Function]

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.

This is a convenience function that simply calls `cairo-surface-copy-page` on *cr*'s target.

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.

This is a convenience function that simply calls `cairo-surface-show-page` on *cr*'s target.

cr a cairo context

2 Paths

Creating paths and manipulating path data

2.1 Overview

Paths are the most basic drawing tools and are primarily used to implicitly generate simple masks.

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==‘#f’`’ 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-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==‘#f’`’ 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-has-current-point` (*cr* `<cairo-t>`) [Function]
⇒ (*ret* `<cairo-bool-t>`)

Returns whether a current point is defined on the current path. See `cairo-get-current-point` for details on the current point.

cr a cairo context

ret whether a current point is defined.

Since 1.6

`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 or if *cr* is in an error status, *x* and *y* will both be set to 0.0. It is possible to check this in advance with `cairo-has-current-point`.

Most path construction functions alter the current point. See the following for details on how they affect the current point: `cairo-new-path`, `cairo-new-sub-path`, `cairo-append-path`, `cairo-close-path`, `cairo-move-to`, `cairo-line-to`, `cairo-curve-to`, `cairo-rel-move-to`, `cairo-rel-line-to`, `cairo-rel-curve-to`, `cairo-arc`, `cairo-arc-negative`, `cairo-rectangle`, `cairo-text-path`, `cairo-glyph-path`, `cairo-stroke-to-path`.

Some functions use and alter the current point but do not otherwise change current path: `cairo-show-text`.

Some functions unset the current path and as a result, current point: `cairo-fill`, `cairo-stroke`.

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. If this initial line is undesired, it can be avoided by calling `cairo-new-sub-path` before calling `cairo-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>`) (*radius* `<double>`) (*angle1* `<double>`) (*angle2* `<double>`) [Function]

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>`) (*x2* `<double>`) (*y2* `<double>`) (*x3* `<double>`) (*y3* `<double>`) [Function]

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>`) (*num-glyphs* `<int>`) [Function]

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 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 NUL-terminated string of text encoded in UTF-8, or '`#f`'

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

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

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

Computes a bounding box in user-space coordinates covering the points on the current path. If the current path is empty, returns an empty rectangle ((0,0), (0,0)). Stroke parameters, fill rule, surface dimensions and clipping are not taken into account.

Contrast with `cairo-fill-extents` and `cairo-stroke-extents` which return the extents of only the area that would be "inked" by the corresponding drawing operations.

The result of `cairo-path-extents` is defined as equivalent to the limit of `cairo-stroke-extents` with '`CAIRO_LINE_CAP_ROUND`' as the line width approaches 0.0, (but never reaching the empty-rectangle returned by `cairo-stroke-extents` for a line width of 0.0).

Specifically, this means that zero-area sub-paths such as `cairo-move-to`;`cairo-line-to` segments, (even degenerate cases where the coordinates to both calls are identical), will be considered as contributing to the extents. However, a lone `cairo-move-to` will not contribute to the results of `cairo-path-extents`.

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 1.6

3 Patterns

Sources for drawing

3.1 Overview

<cairo-pattern> is the paint with which cairo draws. The primary use of patterns is as the source for all cairo drawing operations, although they can also be used as masks, that is, as the brush too.

A cairo pattern is created by using one of the many constructors, of the form cairo_pattern_create_*type*() or implicitly through cairo_set_source_*type*() functions.

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.

If two (or more) stops are specified with identical offset values, they will be sorted according to the order in which the stops are added, (stops added earlier will compare less than stops added later). This can be useful for reliably making sharp color transitions instead of the typical blend.

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.

If two (or more) stops are specified with identical offset values, they will be sorted according to the order in which the stops are added, (stops added earlier will compare less than stops added later). This can be useful for reliably making sharp color transitions instead of the typical blend.

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>`) (*blue* `<double>`) (*alpha* `<double>`) ⇒ (*ret* `<cairo-pattern-t >`) [Function]

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

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

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 (cx0, cy0, radius0) and (cx1, cy1, radius1). 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.

The default extend mode is '`CAIRO_EXTEND_NONE`' for surface patterns and '`CAIRO_EXTEND_PAD`' for gradient patterns.

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.

* Note that you might want to control filtering even when you do not have an explicit `<cairo-pattern-t>` object, (for example when using `cairo-set-source-surface`). In these cases, it is convenient to use `cairo-get-source` to get access to the pattern that cairo creates implicitly. For example:

```
cairo_set_source_surface (cr, image, x, y);
cairo_pattern_set_filter (cairo_get_source (cr), CAIRO_FILTER_NEAREST);
```

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

The current transformation matrix, *ctm*, is a two-dimensional affine transformation that maps all coordinates and other drawing instruments from the *user space* into the surface's canonical coordinate system, also known as the *device space*.

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>`) ⇒ (*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]
⇒ (*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>`) ⇒ (*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]
⇒ (*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 Regions

Representing a pixel-aligned area

5.1 Overview

Regions are a simple graphical data type representing an area of integer-aligned rectangles. They are often used on raster surfaces to track areas of interest, such as change or clip areas.

5.2 Usage

`cairo-region-create` ⇒ (*ret* `<cairo-region-t >`) [Function]

Allocates a new empty region object.

ret A newly allocated `<cairo-region-t>`. Free with `cairo-region-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-region-status`.

Since 1.10

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

Allocates a new region object copying the area from *original*.

original a `<cairo-region-t>`

ret A newly allocated `<cairo-region-t>`. Free with `cairo-region-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-region-status`.

Since 1.10

`cairo-region-get-extents` (*region* `<cairo-region-t>`) [Function]
(*extents* `<cairo-rectangle-int-t>`)

Gets the bounding rectangle of *region* as a `<cairo-rectangle-int-t>`

region a `<cairo-region-t>`

extents rectangle into which to store the extents

Since 1.10

`cairo-region-is-empty` (*region* `<cairo-region-t>`) [Function]
⇒ (*ret* `<cairo-bool-t>`)

Checks whether *region* is empty.

region a `<cairo-region-t>`

ret '`#t`' if *region* is empty, '`#f`' if it isn't.

Since 1.10

`cairo-region-contains-point` (*region* `<cairo-region-t>`) (*x* `<int>`) (*y* `<int>`) ⇒ (*ret* `<cairo-bool-t>`) [Function]

Checks whether (*x*, *y*) is contained in *region*.

region a `<cairo-region-t>`

x the x coordinate of a point

y the y coordinate of a point

ret '`#t`' if (*x*, *y*) is contained in *region*, '`#f`' if it is not.

Since 1.10

`cairo-region-contains-rectangle` (*region* `<cairo-region-t>`) (*rectangle* `<cairo-rectangle-int-t>`) ⇒ (*ret* `<cairo-region-overlap-t>`) [Function]

Checks whether *rectangle* is inside, outside or partially contained in *region*

region a `<cairo-region-t>`

rectangle a `<cairo-rectangle-int-t>`

ret '`CAIRO_REGION_OVERLAP_IN`' if *rectangle* is entirely inside *region*, '`CAIRO_REGION_OVERLAP_OUT`' if *rectangle* is entirely outside *region*, or '`CAIRO_REGION_OVERLAP_PART`' if *rectangle* is partially inside and partially outside *region*.

Since 1.10

`cairo-region-translate` (*region* `<cairo-region-t>`) (*dx* `<int>`) (*dy* `<int>`) [Function]

Translates *region* by (*dx*, *dy*).

region a `<cairo-region-t>`

dx Amount to translate in the x direction

dy Amount to translate in the y direction

Since 1.10

`cairo-region-intersect` (*dst* `<cairo-region-t>`) (*other* `<cairo-region-t>`) ⇒ (*ret* `<cairo-status-t>`) [Function]

Computes the intersection of *dst* with *other* and places the result in *dst*

dst a `<cairo-region-t>`

other another `<cairo-region-t>`

ret '`CAIRO_STATUS_SUCCESS`' or '`CAIRO_STATUS_NO_MEMORY`'

Since 1.10

`cairo-region-subtract` (*dst* `<cairo-region-t>`) (*other* `<cairo-region-t>`) ⇒ (*ret* `<cairo-status-t>`) [Function]

Subtracts *other* from *dst* and places the result in *dst*

dst a `<cairo-region-t>`

other another `<cairo-region-t>`

ret '`CAIRO_STATUS_SUCCESS`' or '`CAIRO_STATUS_NO_MEMORY`'

Since 1.10

`cairo-region-union` (*dst* `<cairo-region-t>`) [Function]
(*other* `<cairo-region-t>`) ⇒ (*ret* `<cairo-status-t>`)

Computes the union of *dst* with *other* and places the result in *dst*

dst a `<cairo-region-t>`

other another `<cairo-region-t>`

ret '`CAIRO_STATUS_SUCCESS`' or '`CAIRO_STATUS_NO_MEMORY`'

Since 1.10

`cairo-region-xor` (*dst* `<cairo-region-t>`) [Function]
(*other* `<cairo-region-t>`) ⇒ (*ret* `<cairo-status-t>`)

Computes the exclusive difference of *dst* with *other* and places the result in *dst*. That is, *dst* will be set to contain all areas that are either in *dst* or in *other*, but not in both.

dst a `<cairo-region-t>`

other another `<cairo-region-t>`

ret '`CAIRO_STATUS_SUCCESS`' or '`CAIRO_STATUS_NO_MEMORY`'

Since 1.10

6 Text

Rendering text and glyphs

6.1 Overview

The functions with *text* in their name form cairo's *toy* text API. The toy API takes UTF-8 encoded text and is limited in its functionality to rendering simple left-to-right text with no advanced features. That means for example that most complex scripts like Hebrew, Arabic, and Indic scripts are out of question. No kerning or correct positioning of diacritical marks either. The font selection is pretty limited too and doesn't handle the case that the selected font does not cover the characters in the text. This set of functions are really that, a toy text API, for testing and demonstration purposes. Any serious application should avoid them.

The functions with *glyphs* in their name form cairo's *low-level* text API. The low-level API relies on the user to convert text to a set of glyph indexes and positions. This is a very hard problem and is best handled by external libraries, like the pangocairo that is part of the Pango text layout and rendering library. Pango is available from http://www.pango.org/ (`http://www.pango.org/`).

6.2 Usage

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

Note: The `cairo-select-font-face` 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.

Selects a family and style of font from a simplified description as a family name, slant and weight. Cairo provides no operation to list available family names on the system (this is a "toy", remember), but the standard CSS2 generic family names, ("serif", "sans-serif", "cursive", "fantasy", "monospace"), are likely to work as expected.

If *family* starts with the string "*cairo*:", or if no native font backends are compiled in, cairo will use an internal font family. The internal font family recognizes many modifiers in the *family* string, most notably, it recognizes the string "monospace". That is, the family name "*cairo*:monospace" will use the monospace version of the internal font family.

For "real" font selection, see the font-backend-specific font_face_create functions for the font backend you are using. (For example, if you are using the freetype-based cairo-ft font backend, see `cairo-ft-font-face-create-for-ft-face` or `cairo-ft-font-face-create-for-pattern`.) The resulting font face could then be used with `cairo-scaled-font-create` and `cairo-set-scaled-font`.

Similarly, when using the "real" font support, you can call directly into the underlying font system, (such as fontconfig or freetype), for operations such as listing available fonts, etc.

It is expected that most applications will need to use a more comprehensive font handling and text layout library, (for example, pango), in conjunction with cairo.

If text is drawn without a call to `cairo-select-font-face`, (nor `cairo-set-font-face` nor `cairo-set-scaled-font`), the default family is platform-specific, but is essentially `"sans-serif"`. Default slant is '`CAIRO_FONT_SLANT_NORMAL`', and default weight is '`CAIRO_FONT_WEIGHT_NORMAL`'.

This function is equivalent to a call to `cairo-toy-font-face-create` followed by `cairo-set-font-face`.

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.)

If text is drawn without a call to `cairo-set-font-size`, (nor `cairo-set-font-matrix` nor `cairo-set-scaled-font`), the default font size is 10.0.

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 NUL-terminated string of text encoded in UTF-8, or '`#f`'

`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-show-text-glyphs` (*cr* `<cairo-t>`) (*utf8* `<char>`) [Function]
(*utf8-len* `<int>`) (*glyphs* `<cairo-glyph-t>`) (*num-glyphs* `<int>`)
(*clusters* `<cairo-text-cluster-t>`) (*num-clusters* `<int>`)
(*cluster-flags* `<cairo-text-cluster-flags-t>`)

This operation has rendering effects similar to `cairo-show-glyphs` but, if the target surface supports it, uses the provided text and cluster mapping to embed the text for the glyphs shown in the output. If the target does not support the extended attributes, this function acts like the basic `cairo-show-glyphs` as if it had been passed *glyphs* and *num-glyphs*.

The mapping between *utf8* and *glyphs* is provided by an array of *clusters*. Each cluster covers a number of text bytes and glyphs, and neighboring clusters cover neighboring areas of *utf8* and *glyphs*. The clusters should collectively cover *utf8* and *glyphs* in entirety.

The first cluster always covers bytes from the beginning of *utf8*. If *cluster-flags* do not have the '`CAIRO_TEXT_CLUSTER_FLAG_BACKWARD`' set, the first cluster also covers the beginning of *glyphs*, otherwise it covers the end of the *glyphs* array and following clusters move backward.

See `<cairo-text-cluster-t>` for constraints on valid clusters.

cr a cairo context

utf8 a string of text encoded in UTF-8

utf8-len length of *utf8* in bytes, or -1 if it is NUL-terminated

glyphs array of glyphs to show

num-glyphs
number of glyphs to show

clusters array of cluster mapping information

num-clusters
number of clusters in the mapping

cluster-flags
cluster mapping flags

Since 1.8

`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 NUL-terminated string of text encoded in UTF-8, or '`#f`'

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

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

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

`cairo-toy-font-face-create` (*family* `<char>`) (*slant* `<cairo-font-slant-t>`) (*weight* `<cairo-font-weight-t>`) ⇒ (*ret* `<cairo-font-face-t >`) [Function]

Creates a font face from a triplet of family, slant, and weight. These font faces are used in implementation of the the `<cairo-t>` "toy" font API.

If *family* is the zero-length string `""`, the platform-specific default family is assumed. The default family then can be queried using `cairo-toy-font-face-get-family`.

The `cairo-select-font-face` function uses this to create font faces. See that function for limitations and other details of toy font faces.

family a font family name, encoded in UTF-8

slant the slant for the font

weight the weight for the font

ret a newly created `<cairo-font-face-t>`. Free with `cairo-font-face-destroy` when you are done using it.

Since 1.8

`cairo-toy-font-face-get-family` [Function]
`(font-face <cairo-font-face-t>) ⇒ (ret <char >)`

Gets the familly name of a toy font.

font-face A toy font face

ret The family name. This string is owned by the font face and remains valid as long as the font face is alive (referenced).

Since 1.8

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

Gets the slant a toy font.

font-face A toy font face

ret The slant value

Since 1.8

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

Gets the weight a toy font.

font-face A toy font face

ret The weight value

Since 1.8

7 Font Faces

Base class for font faces

7.1 Overview

`<cairo-font-face>` represents a particular font at a particular weight, slant, and other characteristic but no size, transformation, or size.

Font faces are created using *font-backend*-specific constructors, typically of the form cairo_*backend*`-font-face-create`, or implicitly using the *toy* text API by way of `cairo-select-font-face`. The resulting face can be accessed using `cairo-get-font-face`.

7.2 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

8 Scaled Fonts

Font face at particular size and options

8.1 Overview

`<cairo-scaled-font>` represents a realization of a font face at a particular size and transformation and a certain set of font options.

8.2 Usage

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

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

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

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 NUL-terminated 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-text-to-glyphs` [Function]
(*scaled-font* `<cairo-scaled-font-t>`) (*x* `<double>`) (*y* `<double>`) (*utf8* `<char>`) (*utf8-len* `<int>`) ⇒ (*ret* `<cairo-status-t>`) (*glyphs* `<cairo-glyph-t*>`) (*num-glyphs* `<int>`) (*clusters* `<cairo-text-cluster-t*>`) (*num-clusters* `<int>`) (*cluster-flags* `<cairo-text-cluster-flags-t>`)

Converts UTF-8 text to an array of glyphs, optionally with cluster mapping, that can be used to render later using *scaled-font*.

If *glyphs* initially points to a non-'`#f`' value, that array is used as a glyph buffer, and *num-glyphs* should point to the number of glyph entries available there. If the provided glyph array is too short for the conversion, a new glyph array is allocated using `cairo-glyph-allocate` and placed in *glyphs*. Upon return, *num-glyphs* always contains the number of generated glyphs. If the value *glyphs* points to has changed after the call, the user is responsible for freeing the allocated glyph array

using `cairo-glyph-free`. This may happen even if the provided array was large enough.

If *clusters* is not '`#f`', *num-clusters* and *cluster-flags* should not be '`#f`', and cluster mapping will be computed. The semantics of how cluster array allocation works is similar to the glyph array. That is, if *clusters* initially points to a non-'`#f`' value, that array is used as a cluster buffer, and *num-clusters* should point to the number of cluster entries available there. If the provided cluster array is too short for the conversion, a new cluster array is allocated using `cairo-text-cluster-allocate` and placed in *clusters*. Upon return, *num-clusters* always contains the number of generated clusters. If the value *clusters* points at has changed after the call, the user is responsible for freeing the allocated cluster array using `cairo-text-cluster-free`. This may happen even if the provided array was large enough.

In the simplest case, *glyphs* and *clusters* can point to '`#f`' initially and a suitable array will be allocated. In code:

```
cairo_status_t status;

cairo_glyph_t *glyphs = NULL;
int num_glyphs;
cairo_text_cluster_t *clusters = NULL;
int num_clusters;
cairo_text_cluster_flags_t cluster_flags;

status = cairo_scaled_font_text_to_glyphs (scaled_font,
                                           x, y,
                                           utf8, utf8_len,
                                           &glyphs, &num_glyphs,
                                           &clusters, &num_clusters, &cluster_fla

if (status == CAIRO_STATUS_SUCCESS) {
    cairo_show_text_glyphs (cr,
                            utf8, utf8_len,
                            glyphs, num_glyphs,
                            clusters, num_clusters, cluster_flags);

    cairo_glyph_free (glyphs);
    cairo_text_cluster_free (clusters);
}
```

If no cluster mapping is needed:

```
cairo_status_t status;

cairo_glyph_t *glyphs = NULL;
int num_glyphs;

status = cairo_scaled_font_text_to_glyphs (scaled_font,
```

```
                                                x, y,
                                                utf8, utf8_len,
                                                &glyphs, &num_glyphs,
                                                NULL, NULL,
                                                NULL);

    if (status == CAIRO_STATUS_SUCCESS) {
        cairo_show_glyphs (cr, glyphs, num_glyphs);
        cairo_glyph_free (glyphs);
    }
```

If stack-based glyph and cluster arrays are to be used for small arrays:

```
    cairo_status_t status;

    cairo_glyph_t stack_glyphs[40];
    cairo_glyph_t *glyphs = stack_glyphs;
    int num_glyphs = sizeof (stack_glyphs) / sizeof (stack_glyphs[0]);
    cairo_text_cluster_t stack_clusters[40];
    cairo_text_cluster_t *clusters = stack_clusters;
    int num_clusters = sizeof (stack_clusters) / sizeof (stack_clusters[0]);
    cairo_text_cluster_flags_t cluster_flags;

    status = cairo_scaled_font_text_to_glyphs (scaled_font,
                                                x, y,
                                                utf8, utf8_len,
                                                &glyphs, &num_glyphs,
                                                &clusters, &num_clusters, &cluster_fla

    if (status == CAIRO_STATUS_SUCCESS) {
        cairo_show_text_glyphs (cr,
                                utf8, utf8_len,
                                glyphs, num_glyphs,
                                clusters, num_clusters, cluster_flags);

        if (glyphs != stack_glyphs)
            cairo_glyph_free (glyphs);
        if (clusters != stack_clusters)
            cairo_text_cluster_free (clusters);
    }
```

For details of how *clusters*, *num-clusters*, and *cluster-flags* map input UTF-8 text to the output glyphs see `cairo-show-text-glyphs`.

The output values can be readily passed to `cairo-show-text-glyphscairo-show-glyphs`, or related functions, assuming that the exact same *scaled-font* is used for the operation.

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

x X position to place first glyph

y Y position to place first glyph

utf8 a string of text encoded in UTF-8

utf8-len length of *utf8* in bytes, or -1 if it is NUL-terminated

glyphs pointer to array of glyphs to fill

num-glyphs
pointer to number of glyphs

clusters pointer to array of cluster mapping information to fill, or '`#f`'

num-clusters
pointer to number of clusters, or '`#f`'

cluster-flags
pointer to location to store cluster flags corresponding to the output *clusters*, or '`#f`'

ret '`CAIRO_STATUS_SUCCESS`' upon success, or an error status if the input values are wrong or if conversion failed. If the input values are correct but the conversion failed, the error status is also set on *scaled-font*.

Since 1.8

`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 uses. This is the font face passed to `cairo-scaled-font-create`.

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*. Note that the translation offsets (x0, y0) of the CTM are ignored by `cairo-scaled-font-create`. So, the matrix this function returns always has 0,0 as x0,y0.

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. However, this function never returns 'CAIRO_FONT_TYPE_TOY'.

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

ret
: The type of *scaled-font*.

Since 1.2

9 Font Options

How a font should be rendered

9.1 Overview

The font options specify how fonts should be rendered. Most of the time the font options implied by a surface are just right and do not need any changes, but for pixel-based targets tweaking font options may result in superior output on a particular display.

9.2 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

10 FreeType Fonts

Font support for FreeType

10.1 Overview

The FreeType font backend is primarily used to render text on GNU/Linux systems, but can be used on other platforms too.

10.2 Usage

11 Win32 Fonts

Font support for Microsoft Windows

11.1 Overview

The Microsoft Windows font backend is primarily used to render text on Microsoft Windows systems.

11.2 Usage

12 Quartz Fonts

Font support via CGFont on OS X

12.1 Overview

The Quartz font backend is primarily used to render text on Apple MacOS X systems. The CGFont API is used for the internal implementation of the font backend methods.

12.2 Usage

13 User Fonts

Font support with font data provided by the user

13.1 Overview

The user-font feature allows the cairo user to provide drawings for glyphs in a font. This is most useful in implementing fonts in non-standard formats, like SVG fonts and Flash fonts, but can also be used by games and other application to draw "funky" fonts.

13.2 Usage

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

Creates a new user font-face.

Use the setter functions to associate callbacks with the returned user font. The only mandatory callback is render_glyph.

After the font-face is created, the user can attach arbitrary data (the actual font data) to it using `cairo-font-face-set-user-data` and access it from the user-font callbacks by using `cairo-scaled-font-get-font-face` followed by `cairo-font-face-get-user-data`.

ret a newly created `<cairo-font-face-t>`. Free with `cairo-font-face-destroy` when you are done using it.

Since 1.8

`cairo-user-font-face-set-init-func` [Function]
(*font-face* `<cairo-font-face-t>`)
(*init-func* `<cairo-user-scaled-font-init-func-t>`)

Sets the scaled-font initialization function of a user-font. See `<cairo-user-scaled-font-init-func-t>` for details of how the callback works.

The font-face should not be immutable or a '`CAIRO_STATUS_USER_FONT_IMMUTABLE`' error will occur. A user font-face is immutable as soon as a scaled-font is created from it.

font-face A user font face

init-func The init callback, or '`#f`'

Since 1.8

14 cairo_device_t

interface to underlying rendering system

14.1 Overview

Devices are the abstraction Cairo employs for the rendering system used by a `<cairo-surface>`. You can get the device of a surface using `cairo-surface-get-device`.

Devices are created using custom functions specific to the rendering system you want to use. See the documentation for the surface types for those functions.

An important function that devices fulfill is sharing access to the rendering system between Cairo and your application. If you want to access a device directly that you used to draw to with Cairo, you must first call `cairo-device-flush` to ensure that Cairo finishes all operations on the device and resets it to a clean state.

Cairo also provides the functions `cairo-device-acquire` and `cairo-device-release` to synchronize access to the rendering system in a multithreaded environment. This is done internally, but can also be used by applications.

Putting this all together, a function that works with devices should look something like this:

```
void
my_device_modifying_function (cairo_device_t *device)
{
  cairo_status_t status;

  // Ensure the device is properly reset
  cairo_device_flush (device);
  // Try to acquire the device
  status = cairo_device_acquire (device);
  if (status != CAIRO_STATUS_SUCCESS) {
    printf ("Failed to acquire the device: %s\n", cairo_status_to_string (status));
    return;
  }

  // Do the custom operations on the device here.
  // But do not call any Cairo functions that might acquire devices.

  // Release the device when done.
  cairo_device_release (device);
}
```

Please refer to the documentation of each backend for additional usage requirements, guarantees provided, and interactions with existing surface API of the device functions for surfaces of that type.

14.2 Usage

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

This function finishes the device and drops all references to external resources. All surfaces, fonts and other objects created for this *device* will be finished, too. Further operations on the *device* will not affect the *device* but will instead trigger a '`CAIRO_STATUS_DEVICE_FINISHED`' error.

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

This function may acquire devices.

device the `<cairo-device-t>` to finish

Since 1.10

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

Finish any pending operations for the device and also restore any temporary modifications cairo has made to the device's state. This function must be called before switching from using the device with Cairo to operating on it directly with native APIs. If the device doesn't support direct access, then this function does nothing.

This function may acquire devices.

device a `<cairo-device-t>`

Since 1.10

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

This function returns the type of the device. See `<cairo-device-type-t>` for available types.

device a `<cairo-device-t>`

ret The type of *device*.

Since 1.10

`cairo-device-acquire` (*device* `<cairo-device-t>`) [Function]
⇒ (*ret* `<cairo-status-t>`)

Acquires the *device* for the current thread. This function will block until no other thread has acquired the device.

If the return value is '`CAIRO_STATUS_SUCCESS`', you successfully acquired the device. From now on your thread owns the device and no other thread will be able to acquire it until a matching call to `cairo-device-release`. It is allowed to recursively acquire the device multiple times from the same thread.

You must never acquire two different devices at the same time unless this is explicitly allowed. Otherwise the possibility of deadlocks exist.

As various Cairo functions can acquire devices when called, these functions may also cause deadlocks when you call them with an acquired device. So you must not have a device acquired when calling them. These functions are marked in the documentation.

device a `<cairo-device-t>`

ret ‘CAIRO_STATUS_SUCCESS’ on success or an error code if the device is in an error state and could not be acquired. After a successful call to `cairo-device-acquire`, a matching call to `cairo-device-release` is required.

Since 1.10

`cairo-device-release` (*device* `<cairo-device-t>`) [Function]

Releases a *device* previously acquired using `cairo-device-acquire`. See that function for details.

device a `<cairo-device-t>`

Since 1.10

15 cairo_surface_t

Base class for surfaces

15.1 Overview

<cairo-surface> is the abstract type representing all different drawing targets that cairo can render to. The actual drawings are performed using a cairo *context*.

A cairo surface is created by using *backend*-specific constructors, typically of the form cairo_*backend*-surface-create.

Most surface types allow accessing the surface without using Cairo functions. If you do this, keep in mind that it is mandatory that you call cairo-surface-flush before reading from or writing to the surface and that you must use cairo-surface-mark-dirty after modifying it. Note that for other surface types it might be necessary to acquire the surface's device first. See cairo-device-acquire for a discussion of devices.

```
void
modify_image_surface (cairo_surface_t *surface)
{
  unsigned char *data;
  int width, height, stride;

  // flush to ensure all writing to the image was done
  cairo_surface_flush (surface);

  // modify the image
  data = cairo_image_surface_get_data (surface);
  width = cairo_image_surface_get_width (surface);
  height = cairo_image_surface_get_height (surface);
  stride = cairo_image_surface_get_stride (surface);
  modify_image_data (data, width, height, stride);

  // mark the image dirty so Cairo clears its caches.
  cairo_surface_mark_dirty (surface);
}
```

15.2 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-create-for-rectangle` [Function]
(*target* <cairo-surface-t>) (*x* <double>) (*y* <double>)
(*width* <double>) (*height* <double>) ⇒ (*ret* <cairo-surface-t >)

Create a new surface that is a rectangle within the target surface. All operations drawn to this surface are then clipped and translated onto the target surface. Nothing drawn via this sub-surface outside of its bounds is drawn onto the target surface, making this a useful method for passing constrained child surfaces to library routines that draw directly onto the parent surface, i.e. with no further backend allocations, double buffering or copies.

The semantics of subsurfaces have not been finalized yet unless the rectangle is in full device units, is contained within the extents of the target surface, and the target or subsurface's device transforms are not changed.

target an existing surface for which the sub-surface will point to

x the x-origin of the sub-surface from the top-left of the target surface (in device-space units)

y the y-origin of the sub-surface from the top-left of the target surface (in device-space units)

width width of the sub-surface (in device-space units)

height height of the sub-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.

Since 1.10

`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, referencing and destroying, and flushing and

finishing 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 modifications 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-device` (*surface* `<cairo-surface-t>`) [Function]
⇒ (*ret* `<cairo-device-t >`)

This function returns the device for a *surface*. See `<cairo-device-t>`.

surface a `<cairo-surface-t>`

ret The device for *surface* or '`#f`' if the surface does not have an associated device.

Since 1.10

`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]
⇒ (*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]
⇒ (*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

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

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

There is a convenience function for this that takes a `<cairo-t>`, namely `cairo-copy-page`.

surface a `<cairo-surface-t>`

Since 1.6

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

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

There is a convenience function for this that takes a `<cairo-t>`, namely `cairo-show-page`.

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

Since 1.6

`cairo-surface-has-show-text-glyphs` [Function]
(*surface* `<cairo-surface-t>`) ⇒ (*ret* `<cairo-bool-t>`)

Returns whether the surface supports sophisticated `cairo-show-text-glyphs` operations. That is, whether it actually uses the provided text and cluster data to a `cairo-show-text-glyphs` call.

Note: Even if this function returns '`#f`', a `cairo-show-text-glyphs` operation targeted at *surface* will still succeed. It just will act like a `cairo-show-glyphs` operation. Users can use this function to avoid computing UTF-8 text and cluster mapping if the target surface does not use it.

surface a `<cairo-surface-t>`

ret '`#t`' if *surface* supports `cairo-show-text-glyphs`, '`#f`' otherwise

Since 1.8

`cairo-surface-get-mime-data` (*surface* `<cairo-surface-t>`) [Function]
(*mime-type* `<string>`) ⇒ (*ret* `<bytevector>`)

Return data previously associated with the surface with the given *mime-type*.

surface a `<cairo-surface-t>`

mime-type
a string denoting the MIME type, for example `"image/jpeg"`, `"image/png"`, `"image/jp2"`, or `"text/x-uri"`

ret `#f` if no data was associated with the surface, otherwise a fresh bytevector holding a copy of the data

Since 1.11

`cairo-surface-set-mime-data` (*surface* `<cairo-surface-t>`) (*mime-type* `<string>`) (*data* `<bytevector>`) [Function]

Associate image data with a surface with the given *mime-type*.

Attach an image in the format *mime_type* to *surface*. To remove the data from a surface, call this function with same mime type and `#f` for *data*.

The attached image (or filename) data can later be used by backends which support it (currently: PDF, PS, SVG and Win32 Printing surfaces) to emit this data instead of making a snapshot of the *surface*. This approach tends to be faster and requires less memory and disk space.

See corresponding backend surface docs for details about which MIME types it can handle. Caution: the associated MIME data will be discarded if you draw on the surface afterwards. Use this function with care.

Even if a backend supports a MIME type, that does not mean cairo will always be able to use the attached MIME data. For example, if the backend does not natively support the compositing operation used to apply the MIME data to the backend. In that case, the MIME data will be ignored. Therefore, to apply an image in all cases, it is best to create an image surface which contains the decoded image data and then attach the MIME data to that. This ensures the image will always be used while still allowing the MIME data to be used whenever possible.

surface a `<cairo-surface-t>`

mime-type
a string denoting the MIME type, for example `"image/jpeg"`, `"image/png"`, `"image/jp2"`, or `"text/x-uri"`

data a bytevector of data to associate with the surface, or `#f` to remove the association

Since 1.11

16 Image Surfaces

Rendering to memory buffers

16.1 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>`.

16.2 Usage

`cairo-format-stride-for-width` (*format* `<cairo-format-t>`) [Function]
(*width* `<int>`) ⇒ (*ret* `<int>`)

This function provides a stride value that will respect all alignment requirements of the accelerated image-rendering code within cairo. Typical usage will be of the form:

```
int stride;
unsigned char *data;
cairo_surface_t *surface;

stride = cairo_format_stride_for_width (format, width);
data = malloc (stride * height);
surface = cairo_image_surface_create_for_data (data, format,
  width, height,
  stride);
```

format A `<cairo-format-t>` value

width The desired width of an image surface to be created.

ret the appropriate stride to use given the desired format and width, or -1 if either the format is invalid or the width too large.

Since 1.6

`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

17 PDF Surfaces

Rendering PDF documents

17.1 Overview

The PDF surface is used to render cairo graphics to Adobe PDF files and is a multi-page vector surface backend.

17.2 Usage

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

Creates a PDF surface of the specified size in points to be written to *filename*. If *filename* is not given, the output is sent to the current output port.

filename a filename for the PDF output (must be writable), '`#f`' may be used to specify no output. This will generate a PDF surface that may be queried and used as a source, without generating a temporary file.

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-get-versions` ⇒ (*versions* <cairo-pdf-version-t const*>) (*num-versions* <int>) [Function]

Used to retrieve the list of supported versions. See `cairo-pdf-surface-restrict-to-version`.

versions supported version list

num-versions
list length

Since 1.10

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

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

18 PNG Support

Reading and writing PNG images

18.1 Overview

The PNG functions allow reading PNG images into image surfaces, and writing any surface to a PNG file.

18.2 Usage

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

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`' Alternatively, you can allow errors to propagate through the drawing operations and check the status on the context upon completion using `cairo-status`.

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

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.

19 PostScript Surfaces

Rendering PostScript documents

19.1 Overview

The PostScript surface is used to render cairo graphics to Adobe PostScript files and is a multi-page vector surface backend.

19.2 Usage

`cairo-ps-surface-create` (*width-in-points* `<double>`) [Function]
(*height-in-points* `<double>`) (*filename* `<char>`)
⇒ (*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), '`#f`' may be used to specify no output. This will generate a PS surface that may be queried and used as a source, without generating a temporary file.

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-restrict-to-level` [Function]
(*surface* `<cairo-surface-t>`) (*level* `<cairo-ps-level-t>`)

Restricts the generated PostSript file to *level*. See `cairo-ps-get-levels` for a list of available level values that can be used here.

This function should only be called before any drawing operations have been performed on the given surface. The simplest way to do this is to call this function immediately after creating the surface.

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

level PostScript level

Since 1.6

`cairo-ps-get-levels` ⇒ (*levels* `<cairo-ps-level-t const*>`) (*num-levels* `<int>`) [Function]

Used to retrieve the list of supported levels. See `cairo-ps-surface-restrict-to-level`.

levels supported level list

num-levels list length

Since 1.6

`cairo-ps-surface-set-eps` (*surface* `<cairo-surface-t>`) (*eps* `<cairo-bool-t>`) [Function]

If *eps* is '`#t`', the PostScript surface will output Encapsulated PostScript.

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. An Encapsulated PostScript file should never contain more than one page.

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

eps '`#t`' to output EPS format PostScript

Since 1.6

`cairo-ps-surface-get-eps` (*surface* `<cairo-surface-t>`) ⇒ (*ret* `<cairo-bool-t>`) [Function]

Check whether the PostScript surface will output Encapsulated PostScript.

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

ret '`#t`' if the surface will output Encapsulated PostScript.

Since 1.6

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

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>) (*comment* <char>) [Function]

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

20 Recording Surfaces

Records all drawing operations

20.1 Overview

A recording surface is a surface that records all drawing operations at the highest level of the surface backend interface, (that is, the level of paint, mask, stroke, fill, and show_text_glyphs). The recording surface can then be "replayed" against any target surface by using it as a source surface.

If you want to replay a surface so that the results in target will be identical to the results that would have been obtained if the original operations applied to the recording surface had instead been applied to the target surface, you can use code like this:

```
        cairo_t *cr;

    cr = cairo_create (target);
    cairo_set_source_surface (cr, recording_surface, 0.0, 0.0);
    cairo_paint (cr);
    cairo_destroy (cr);
```

A recording surface is logically unbounded, i.e. it has no implicit constraint on the size of the drawing surface. However, in practice this is rarely useful as you wish to replay against a particular target surface with known bounds. For this case, it is more efficient to specify the target extents to the recording surface upon creation.

The recording phase of the recording surface is careful to snapshot all necessary objects (paths, patterns, etc.), in order to achieve accurate replay. The efficiency of the recording surface could be improved by improving the implementation of snapshot for the various objects. For example, it would be nice to have a copy-on-write implementation for _cairo_surface_snapshot.

20.2 Usage

`cairo-recording-surface-create` (*content* <cairo-content-t>) [Function]
(*extents* <cairo-rectangle-t>) ⇒ (*ret* <cairo-surface-t >)

Creates a recording-surface which can be used to record all drawing operations at the highest level (that is, the level of paint, mask, stroke, fill and show_text_glyphs). The recording surface can then be "replayed" against any target surface by using it as a source to drawing operations.

The recording phase of the recording surface is careful to snapshot all necessary objects (paths, patterns, etc.), in order to achieve accurate replay.

content the content of the recording surface

extents the extents to record in pixels, can be '`#f`' to record unbounded operations.

ret a pointer to the newly created surface. The caller owns the surface and should call `cairo-surface-destroy` when done with it.

Since 1.10

`cairo-recording-surface-ink-extents` [Function]

(*surface* <cairo-surface-t>) ⇒ (*x0* <double>) (*y0* <double>) (*width* <double>) (*height* <double>)

Measures the extents of the operations stored within the recording-surface. This is useful to compute the required size of an image surface (or equivalent) into which to replay the full sequence of drawing operations.

surface	a <cairo-recording-surface-t>
x0	the x-coordinate of the top-left of the ink bounding box
y0	the y-coordinate of the top-left of the ink bounding box
width	the width of the ink bounding box
height	the height of the ink bounding box

Since 1.10

21 SVG Surfaces

Rendering SVG documents

21.1 Overview

The SVG surface is used to render cairo graphics to SVG files and is a multi-page vector surface backend.

21.2 Usage

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

Creates a SVG surface of the specified size in points to be written to *filename*. If *filename* is not given, the output is sent to the current output port.

The SVG surface backend recognizes the following MIME types for the data attached to a surface (see `cairo-surface-set-mime-data`) when it is used as a source pattern for drawing on this surface: '`CAIRO_MIME_TYPE_JPEG`', '`CAIRO_MIME_TYPE_PNG`', '`CAIRO_MIME_TYPE_URI`'. If any of them is specified, the SVG backend emits a href with the content of MIME data instead of a surface snapshot (PNG, Base64-encoded) in the corresponding image tag.

The unofficial MIME type '`CAIRO_MIME_TYPE_URI`' is examined first. If present, the URI is emitted as is: assuring the correctness of URI is left to the client code.

If '`CAIRO_MIME_TYPE_URI`' is not present, but '`CAIRO_MIME_TYPE_JPEG`' or '`CAIRO_MIME_TYPE_PNG`' is specified, the corresponding data is Base64-encoded and emitted.

filename a filename for the SVG output (must be writable), '`#f`' may be used to specify no output. This will generate a SVG surface that may be queried and used as a source, without generating a temporary file.

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

22 cairo_matrix_t

Generic matrix operations

22.1 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:

```
x_new = xx * x + xy * y + x0;
y_new = yx * x + yy * y + y0;
```

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.

22.2 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 its 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.

matrix a <cairo-matrix-t>

ret If *matrix* has an inverse, modifies *matrix* to be the inverse matrix and returns ‘CAIRO_STATUS_SUCCESS’. Otherwise, returns ‘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:

```
dx2 = dx1 * a + dy1 * c;
dy2 = dx1 * b + dy1 * d;
```

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

23 Error handling

Decoding cairo's status

23.1 Overview

Cairo uses a single status type to represent all kinds of errors. A status value of '`CAIRO_STATUS_SUCCESS`' represents no error and has an integer value of zero. All other status values represent an error.

Cairo's error handling is designed to be easy to use and safe. All major cairo objects *retain* an error status internally which can be queried anytime by the users using cairo*_status() calls. In the mean time, it is safe to call all cairo functions normally even if the underlying object is in an error status. This means that no error handling code is required before or after each individual cairo function call.

23.2 Usage

24 Version Information

Compile-time and run-time version checks.

24.1 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)
```

24.2 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.

24.3 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 the 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 ());
```

24.4 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.

25 Types

Generic data types

25.1 Overview

This section lists generic data types used in the cairo API.

25.2 Usage

Concept Index

(Index is nonexistent)

Function Index