CSS Anchor Positioning Module Level 1

Editor's Draft, 22 December 2025

▼ **More details about this document**

This version:
https://drafts.csswg.org/css-anchor-position-1/

Latest published version:
https://www.w3.org/TR/css-anchor-position-1/

Feedback:
CSSWG Issues Repository
Inline In Spec

Editors:
Tab Atkins-Bittner (Google)
Elika J. Etemad / fantasai (Apple)
Ian Kilpatrick (Google)

Former Editor:
Jhey Tompkins (Google)

Suggest an Edit for this Spec:
GitHub Editor

Abstract

This specification defines anchor positioning, where a positioned element can size and position itself relative to one or more "anchor elements" elsewhere on the page.

CSS is a language for describing the rendering of structured documents (such as HTML and XML) on screen, on paper, etc.

Status of this document

This is a public copy of the editors' draft. It is provided for discussion only and may change at any moment. Its publication here does not imply endorsement of its contents by W3C. Don't cite this document other than as work in progress.

Please send feedback by filing issues in GitHub (preferred), including the spec code "css-anchor-position" in the title, like this: "[css-anchor-position] *...summary of comment...*". All issues and comments are archived. Alternately, feedback can be sent to the (archived) public mailing list www-style@w3.org.

This document is governed by the 18 August 2025 W3C Process Document.

Table of Contents

1 Introduction
1.1 Value Definitions

2 Determining the Anchor
2.1 Creating an Anchor: the 'anchor-name' property
2.2 Scoping Anchor Names: the 'anchor-scope' property
2.3 Finding an Anchor
2.4 Default Anchors: the 'position-anchor' property
2.4.1 Implicit Anchor Elements
2.5 Anchor Relevance

3 Anchor-Based Positioning
3.1 The 'position-area' Property
3.1.1 Resolving the Position Area Grid
3.1.2 Syntax of <position-area> Values
3.1.3 Computed Value and Serialization of <position-area>
3.2 Anchor-relative Insets: the 'anchor()' function
3.2.1 Resolution of 'anchor()'
3.3 Taking Scroll Into Account

4 Anchor-Based Alignment
4.1 Area-specific Default Alignment
4.2 Centering on the Anchor: the 'anchor-center' alignment value

5 Anchor-Based Sizing
5.1 The 'anchor-size()' Function

5.1.1 Resolution of ‘anchor-size()’

6 Overflow Management
6.1 Giving Fallback Options: the ‘position-try-fallbacks’ property
6.2 Determining Fallback Order: the ‘position-try-order’ property
6.3 The ‘position-try’ Shorthand
6.4 The ‘@position-try’ Rule
6.5 Applying Position Fallback
6.5.1 Maintaining and Clearing Fallback Choices
6.5.1.1 Recording the last successful position option
6.5.1.2 Suspending Fallback During Transitions
6.5.1.3 Suspending Fallback During Animations
6.5.2 Applying Position Options
6.6 Conditional Hiding: the ‘position-visibility’ property

7 Accessibility Implications

8 DOM Interfaces
8.1 The CSSPositionTryRule interface

9 Appendix: Style & Layout Interleaving

10 Security Considerations

11 Privacy Considerations

12 Changes

Conformance
Document conventions
Conformance classes
Partial implementations
Implementations of Unstable and Proprietary Features
Non-experimental implementations

Index
Terms defined by this specification
Terms defined by reference

References

Normative References
Informative References

Property Index

IDL Index

Issues Index

1. Introduction

CSS absolute positioning allows authors to place boxes anywhere on the page, without regard to the layout of other boxes besides their containing block. This flexibility can be very useful, but also very limiting—often you want to position relative to *some* other box. ***Anchor positioning*** (via the 'position-anchor' and 'position-area' properties and/or the ***anchor functions*** 'anchor()' and 'anchor-size()') allows authors to achieve this, "anchoring" an absolutely positioned box to one or more other boxes on the page (its ***anchor references***, while also allowing them to try several possible positions to find the "best" one that avoids overlap/overflow.

EXAMPLE 1 ¶

For example, an author might want to position a tooltip centered and above the targeted element, unless that would place the tooltip offscreen, in which case it should be below the targeted element. This can be done with the following CSS:

```
.anchor {
  anchor-name: --tooltip;
}
.tooltip {
  /* Fixpos means we don't need to worry about
     containing block relationships;
     the tooltip can live anywhere in the DOM. */
  position: fixed;

  /* All the anchoring behavior will default to
     referring to the --tooltip anchor. */
  position-anchor: --tooltip;

  /* Align the tooltip's bottom to the top of the anchor;
     this also defaults to horizontally center-aligning
     the tooltip and the anchor (in horizontal writing modes). */
  position-area: block-start;

  /* Automatically swap if this overflows the window
     so the tooltip's top aligns to the anchor's bottom
     instead. */
  position-try: flip-block;

  /* Prevent getting too wide */
  max-inline-size: 20em;
}
```

Note that using the Popover API will automatically set ‘position’ and create the anchoring relationship without setting ‘anchor-name’ or ‘position-anchor’ value (by defining an implicit anchor element), so those properties wouldn’t need to be explicitly set again. So with the correct markup, this example can be simplified to:

```
.tooltip {
  /* Using the popover + popovertarget attributes sets 'position: fixed'
     and creates the necessary position-anchor relationship already. */
  position-area: block-start;
```

```
    position-try: flip-block;
    max-inline-size: 20em;
  }
```

§ 1.1. Value Definitions

This specification follows the CSS property definition conventions from [CSS2] using the value definition syntax from [CSS-VALUES-3]. Value types not defined in this specification are defined in CSS Values & Units [CSS-VALUES-3]. Combination with other CSS modules may expand the definitions of these value types.

In addition to the property-specific values listed in their definitions, all properties defined in this specification also accept the CSS-wide keywords as their property value. For readability they have not been repeated explicitly.

Like most operations in CSS besides selector matching, features in this specification operate over the flattened element tree.

§ 2. Determining the Anchor

Several features of this specification refer to the position and size of an ***anchor box***. Unless otherwise specified, this refers to the border box edge of the principal box of relevant anchor element. In most cases the relevant anchor element is specified as the default anchor element using the 'position-anchor' property, which can refer to an implicit anchor element defined by the host language or an anchor named via the CSS 'anchor-name' and 'anchor-scope' properties. (The 'anchor()' functions can also reference a named anchor directly.)

The anchor box's position and size is determined after layout. This position and size includes 'zoom' and 'position'-based adjustments (such as 'position: relative' or 'position: sticky') as well as transforms (such as 'transform' or 'offset-path'). In these cases, the axis-aligned bounding rectangle of the anchor box in the coordinate space of the absolutely positioned element's containing block is used instead. Transforms are often optimized onto a different thread, so transform-based updates to an anchor box's position may be delayed by a few frames. Authors can avoid this delay by using absolute or relative positioning instead where practical.

If the anchor box is fragmented, and the containing block of the absolutely positioned box referring to that anchor box is outside the relevant fragmentation context, the axis-aligned bounding rectangle of

its box fragments is used instead. (If the absolutely positioned box is inside the fragmentation context, it sees the anchor box as unfragmented—and can be itself fragmented by the fragmentation context.)

For performance reasons, scrolling is handled specially, see § 3.3 Taking Scroll Into Account. Other post-layout effects, such as filters, do not affect the anchor box's position.

§ 2.1. Creating an Anchor: the 'anchor-name' property

Name:	***'anchor-name'***
Value:	none | <dashed-ident>#
Initial:	none
Applies to:	all elements that generate a principal box
Inherited:	no
Percentages:	n/a
Computed value:	as specified
Canonical order:	per grammar
Animation type:	discrete

The 'anchor-name' property declares that an element is an ***anchor element***, whose principal box is an anchor box, and gives it a list of ***anchor names*** to be targeted by. Values are defined as follows:

'none'
: The property has no effect.

'<dashed-ident>#'
: If the element generates a principal box, the element is an anchor element, with a list of anchor names as specified. Each anchor name is a loosely matched tree-scoped name.

  Otherwise, the property has no effect.

Anchor names do not need to be unique. Not all elements are capable of being the target anchor

element of a given box. Thus a name can be reused in multiple places if the usages are scoped appropriately.

> NOTE: If multiple elements share an anchor name and are all visible to a given positioned box, the target anchor element will be the last one in DOM order. The 'anchor-scope' property can be used to further limit what names are visible to a given referencing box.

Anchor names are *not* scoped by containment by default; even if an element has style or layout containment (or any similar sort of containment), the anchor names of its descendants are visible to elements elsewhere in the page.

> NOTE: While an element is in the skipped contents of another element (due to 'content-visibility: hidden', for instance), it's not an acceptable anchor element, effectively acting as if it had no names.

> NOTE: Positioned elements in shadow trees can reference anchor names defined in "higher" trees. Currently, they cannot reference anchor names defined in "lower" shadow trees, though.

§ 2.2. Scoping Anchor Names: the 'anchor-scope' property

Name:	***'anchor-scope'***
Value:	none ¦ all	<dashed-ident>#
Initial:	none
Applies to:	all elements
Inherited:	no
Percentages:	n/a
Computed value:	as specified
Canonical order:	per grammar

Animation type:	discrete

This property scopes the specified anchor names, and lookups for these anchor names, to this element's subtree. See § 2 Determining the Anchor.

Values have the following meanings:

'none'
: No changes in anchor name scope.

'all'
: Specifies that all anchor names defined by this element or its descendants—whose scope is not already limited by a descendant using 'anchor-scope'—to be in scope only for this element's descendants; and limits descendants to only match anchor names to anchor elements within this subtree.

  This value only affects anchor names in the same tree scope, as if it were a strictly matched tree-scoped name. (That is, 'anchor-scope: all' acts identically to 'anchor-scope: --foo, --bar, ...', listing all relevant anchor names.)

'<dashed-ident>'
: Specifies that a matching anchor name defined by this element or its descendants—whose scope is not already limited by a descendant using 'anchor-scope'—to be in scope only for this element's descendants; and limits descendants to only match these anchor names to anchor elements within this subtree.

  The <dashed-ident> represents a strictly matched tree-scoped name, i.e. it can only match against anchor names in the same shadow tree.[CSS-SCOPING-1]

This property has no effect on implicit anchor elements.

EXAMPLE 2 ¶

When a design pattern is re-used, ‘anchor-scope’ can prevent naming clashes across identical components. For example, if a list contains positioned elements within each list item, which want to position themselves relative to the list item they’re in,

```
li {
  anchor-name: --list-item;
  anchor-scope: --list-item;
}
li .positioned {
  position: absolute;
  position-anchor: --list-item;
  position-area: inline-start;
}
```

Without ‘anchor-scope’, all of the `<li>` elements would be visible to all of the positioned elements, and so they’d all positioned themselves relative to the *final* `<li>`, stacking up on top of each other.

§ 2.3. Finding an Anchor

Several things in this specification find a target anchor element, given an ***anchor specifier***, which is either a <dashed-ident> (and a tree-scoped reference) that should match an ‘anchor-name’ value elsewhere on the page, or the keyword ‘auto’, or nothing (a missing specifier).

NOTE: The general rule captured by these conditions is that an element can only be a positioned box’s target anchor element if its own box is fully laid out before the positioned box that wants to reference it is laid out. CSS’s layout rules provide some useful guarantees about this, depending on the anchor and positioned box’s relationship with each other and their containing blocks. The list of conditions below exactly rephrases the stacking context rules into just what’s relevant for this purpose, ensuring there is no possibility of circularity in anchor positioning.

To determine the ***target anchor element*** given a querying element *query el* and an optional anchor specifier *anchor spec*:

1. If *anchor spec* was not passed, return the default anchor element if it exists, otherwise return nothing.
2. If *anchor spec* is ‘auto’:

1. If *query el* has an implicit anchor element that is an acceptable anchor element, return that element.
2. Otherwise, return nothing.

NOTE: Future APIs might also define implicit anchor elements. When they do, they'll be explicitly handled in this algorithm, to ensure coordination.

3. Otherwise, *anchor spec* is a <dashed-ident>. Return the last element *el* in tree order that satisfies the following conditions:
   - *el* is an anchor element with an anchor name of *anchor spec*.
   - *el*'s anchor name loosely matches *anchor spec*.

     NOTE: The anchor name is a tree-scoped name, while *anchor spec* is a tree-scoped reference.
   - *el* is an acceptable anchor element for *query el*.

   If no element satisfies these conditions, return nothing.

NOTE: 'anchor-scope' can restrict the visibility of certain anchor names, which can affect what elements can be anchor elements for a given lookup.

NOTE: An 'anchor-name' defined by styles in one shadow tree won't be seen by anchor functions in styles in a different shadow tree, preserving encapsulation. However, *elements* in different shadow trees can still anchor to each other, so long as both the 'anchor-name' and anchor function come from styles in the same tree, such as by using '::part()' to style an element inside a shadow. (Implicit anchor elements also aren't intrinsically limited to a single tree, but the details of that will depend on the API assigning them.)

An element *possible anchor* is an ***acceptable anchor element*** for an absolutely positioned element *positioned el* if all of the following are true:

- *possible anchor* is either an element or a fully styleable tree-abiding pseudo-element.
- *possible anchor* is in scope for *positioned el*, per the effects of 'anchor-scope' on *possible anchor* or its ancestors.
- *possible anchor* is laid out strictly before *positioned el*, aka one of the following is true:

- *possible anchor* and *positioned el* have the same original containing block and either
  - *possible anchor* is in a lower top layer than *positioned el*, or
  - they both exist in the same top layer, but *possible anchor* is either not absolutely positioned or occurs earlier in the flat tree order than *positioned el*
- The element generating *possible anchor*'s containing block (if one exists) is an acceptable anchor element for *positioned el*

- If *possible anchor* is in the skipped contents of another element, then *positioned el* is in the skipped contents of that same element.

> NOTE: In other words, *positioned el* can anchor to *possible anchor* if they're both in the same skipped "leaf", but it can't anchor "across" leafs. This means skipping an element that contains both of them won't suddenly cause the *positioned el* to move to another anchor, but still prevents positioned elements *elsewhere* in the page from anchoring to the skipped element.

§ 2.4. Default Anchors: the 'position-anchor' property

Name:	***'position-anchor'***
Value:	none ¦ auto	<anchor-name>
Initial:	none
Applies to:	absolutely positioned boxes
Inherited:	no
Percentages:	n/a
Computed value:	as specified
Canonical order:	per grammar
Animation type:	discrete

The 'position-anchor' property specifies the ***default anchor element***, which is used by 'position-area', 'position-try', and (by default) all anchor functions applied to this element. 'position-anchor' is a reset-only sub-property of 'position'.

'none'
: The box has no default anchor element.

'auto'
: Use the implicit anchor element if it exists; otherwise the box has no default anchor element.

'<anchor-name>'
: The target anchor element selected by the specified <anchor-name> is the box's default anchor element.

ISSUE 1 We might want to change the initial value to be slightly more magical, auto-choosing between 'none' and 'auto' based on 'position-area' being used or not. [Issue #13067] ¶

The principal box of the default anchor element is the box's ***default anchor box***.

EXAMPLE 3 ¶

For example, in the following code both '.foo' and '.bar' elements can use the same positioning properties, just changing the anchor element they're referring to:

```
.anchored {
  position: absolute;
  top: calc(.5em + anchor(outside));
  /* Since no anchor name was specified,
     this automatically refers to the
     default anchor box. */
}

.foo.anchored {
  position-anchor: --foo;
}
.bar.anchored {
  position-anchor: --bar;
}
```

§ 2.4.1. Implicit Anchor Elements

Some specifications can define that, in certain circumstances, a particular element is an ***implicit anchor element*** for another element.

EXAMPLE 4 ¶

TODO: Fill in an example new popover-related details (once that finally lands in the HTML spec).

Implicit anchor elements can be referenced with the 'auto' keyword in 'position-anchor', or by omitting the anchor reference in anchor functions.

The implicit anchor element of a pseudo-element is its originating element, unless otherwise specified.

§ 2.5. Anchor Relevance

When determining whether an element *el* is relevant to the user, if a descendant of *el* is a target anchor element for a positioned box (which itself is not skipped and whose containing block is not *el* or a descendant of *el*), then *el* must be considered relevant to the user.

NOTE: This means that, for example, an anchor in a 'content-visibility: auto' subtree will prevent its subtree from skipping its contents as long as the positioned box relying on it is also not skipped. (Unless the anchor and the positioned box are both under the same 'content-visibility: auto' element; they can't cyclicly keep each other visible.)

§ 3. Anchor-Based Positioning

An absolutely positioned box can position itself relative to one or more anchor boxes on the page.

The 'position-area' property offers a convenient grid-based concept for positioning relative to the default anchor box; for more complex positioning or positioning relative to multiple boxes, the 'anchor()' function can be used in the inset properties to explicitly refer to edges of an anchor box.

§ 3.1. The 'position-area' Property

Name:	***'position-area'***
Value:	none | <position-area>
Initial:	none
Applies to:	positioned boxes with a default anchor box
Inherited:	no
Percentages:	n/a
Computed value:	the keyword 'none' or a pair of keywords, see § 3.1.3 Computed Value and Serialization of <position-area>
Canonical order:	per grammar
Animation type:	TBD

Most common use-cases of anchor positioning are only concerned with the edges of the positioned box's containing block and the edges of the default anchor box. These lines can be thought of as

defining a 3×3 grid; ‘position-area’ lets you easily specify what area of this position-area grid to lay out the positioned box in.

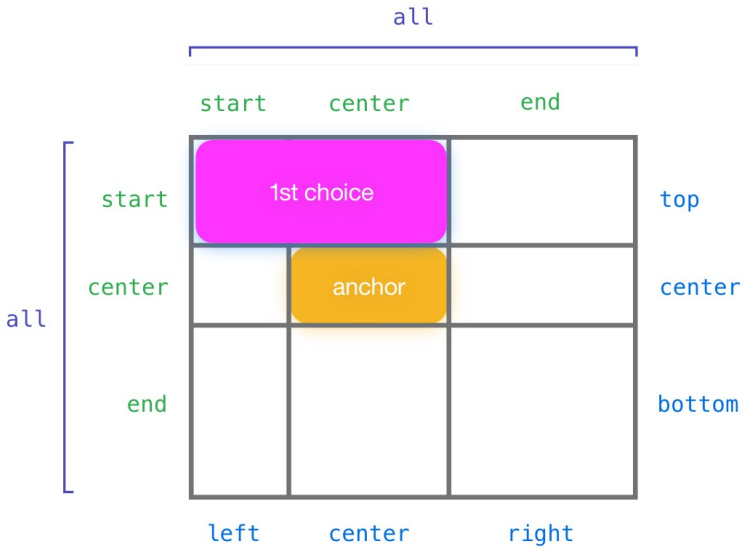

Figure 1 *An example of ‘position-area: top left’ positioning in a ‘horizontal-tb’ ‘ltr’ writing mode.*

‘none’

The property has no effect.

‘<position-area>’

If the box does not have a default anchor box, or is not an absolutely positioned box, this value has no effect.

Otherwise, selects a region of the position-area grid, and makes that the box’s containing block.

> NOTE: This means that the inset properties specify offsets from the position-area, and some property values, like ‘max-height: 100%’, will be relative to the position-area as well.

Values other than ‘none’ have the following additional effects:

- The scrollable containing block is used in place of the local containing block when the absolute-position containing block is generated by a scroll container, so that the entire scrollable overflow area (typically) is available for positioning.
- The used value of any ‘auto’ inset properties and ‘auto’ margin properties resolves to ‘0’.
- The ‘normal’ value for the self-alignment properties resolves to a corresponding value, see § 4.1 Area-specific Default Alignment.

§ 3.1.1. Resolving the Position Area Grid

The ***position-area grid*** is a 3×3 grid, composed of four grid lines in each axis. In order (using the writing mode of the containing block):

- the start edge of the box's pre-modification containing block, or the start edge of the default anchor box if that is more start-ward
- the start edge of the default anchor box
- the end edge of the default anchor box
- the end edge of the box's pre-modification containing block, or the end edge of the default anchor box if that is more end-ward.

> NOTE: When the default anchor box is partially or completely outside of the pre-modified containing block, some of the position-area grid's rows or columns can be zero-sized.

§ 3.1.2. Syntax of <position-area> Values

Positions are specified as a pair of values, which can be expressed in flow-relative or physical terms. The allowed syntax of a <position-area> value is:

```
<position-area> = [
  [ left | center | right | span-left | span-right
  | x-start | x-end | span-x-start | span-x-end
  | self-x-start | self-x-end | span-self-x-start | span-self-x-end
  | span-all ]
  ||
  [ top | center | bottom | span-top | span-bottom
  | y-start | y-end | span-y-start | span-y-end
  | self-y-start | self-y-end | span-self-y-start | span-self-y-end
  | span-all ]
|
  [ block-start | center | block-end | span-block-start | span-block-end | span-all
  ||
  [ inline-start | center | inline-end | span-inline-start | span-inline-end
  | span-all ]
|
  [ self-block-start | center | self-block-end | span-self-block-start
  | span-self-block-end | span-all ]
  ||
  [ self-inline-start | center | self-inline-end | span-self-inline-start
```

```
  | span-self-inline-end | span-all ]
|
  [ start | center | end | span-start | span-end | span-all ]{1,2}
|
  [ self-start | center | self-end | span-self-start | span-self-end | span-all ]{1
]
```

The <position-area> value selects a region of the position-area grid by specifying the rows and columns the region occupies as follows:

'start', ***'end'***, ***'self-start'***, ***'self-end'***
'top', ***'bottom'***, ***'left'***, ***'right'***
'y-start', ***'y-end'***, ***'self-y-start'***, ***'self-y-end'***
'x-start', ***'x-end'***, ***'self-x-start'***, ***'self-x-end'***
'block-start', ***'block-end'***, ***'self-block-start'***, ***'self-block-end'***
'inline-start', ***'inline-end'***, ***'self-inline-start'***, ***'self-inline-end'***
'center'

The single corresponding row or column, depending on which axis this keyword is specifying.

Like in 'anchor()', the plain logical keywords ('start', 'end', etc) refer to the writing mode of the box's containing block. The 'x-start'/etc determine their direction in the same way, but in the specified physical axis.

The 'self-*' logical keywords ('self-start', 'self-x-end', etc) are identical, but refer to the box's own writing mode.

'span-start', ***'span-end'***, ***'span-self-start'***, ***'span-self-end'***
'span-top', ***'span-bottom'***, ***'span-left'***, ***'span-right'***
'span-y-start', ***'span-y-end'***, ***'span-self-y-start'***, ***'span-self-y-end'***
'span-x-start', ***'span-x-end'***, ***'span-self-x-start'***, ***'span-self-x-end'***
'span-block-start', ***'span-block-end'***, ***'span-self-block-start'***, ***'span-self-block-end'***
'span-inline-start', ***'span-inline-end'***, ***'span-self-inline-start'***, ***'span-self-inline-end'***

Two adjacent rows or columns, depending on which axis this keyword is specifying: the center row/column, and the row/column corresponding to the other half of the keyword as per the single-track keywords.

(For example, 'span-top' spans the first two rows—the center row and the top row.)

'span-all'

All three rows or columns, depending on which axis this keyword is specifying.

Some keywords are ambiguous about what axis they refer to: 'center', 'span-all', and the 'start'/etc keywords that don't specify the block or inline axis explicitly. If the other keyword is unambiguous about its axis, then the ambiguous keyword is referring to the opposite axis. (For example, in 'block-start center', the 'center' keyword is referring to the inline axis.) If both keywords are ambiguous, however, then the first refers to the block axis of the box's containing block, and the second to the

inline axis. (For example, 'span-all start' is equivalent to 'span-all inline-start'.)

If only a single keyword is given, it behaves as if the second keyword is 'span-all' if the given keyword is unambigous about its axis; otherwise, it behaves as if the given keyword was repeated. (For example, 'top' is equivalent to 'top span-all', but 'center' is equivalent to 'center center'.)

3.1.3. Computed Value and Serialization of <position-area>

The computed value of a <position-area> value is the two keywords indicating the selected tracks in each axis, with the long ('block-start') and short ('start') logical keywords treated as equivalent. It serializes in the order given in the grammar (above), with the logical keywords serialized in their short forms (e.g. 'start start' instead of 'block-start inline-start').

3.2. Anchor-relative Insets: the 'anchor()' function

An absolutely positioned box can use the ***'anchor()'*** function as a value in its inset properties to refer to the position of one or more anchor boxes. The 'anchor()' function resolves to a <length>. It is only allowed in the inset properties (and is otherwise invalid).

Name:	'top', 'left', 'right', 'bottom'
New values:	<anchor()>

```
<anchor()> = anchor( <anchor-name>? && <anchor-side>, <length-percentage>? )
<anchor-name> = <dashed-ident>
<anchor-side> = inside | outside
      | top | left | right | bottom
      | start | end | self-start | self-end
      | <percentage> | center
```

The 'anchor()' function has three arguments:

- the <anchor-name> value specifies how to find the anchor element it will be drawing positioning information from. Its possible values are:

  '<dashed-ident>'
  : Specifies the anchor name it will look for. This name is a tree-scoped reference.

  omitted

Selects the default anchor element defined for the box, if possible.

See target anchor element for details.

- the <anchor-side> value refers to the position of the corresponding side of the target anchor element. Its possible values are:

  'inside'
  'outside'
  Resolves to one of the anchor box's sides, depending on which inset property it's used in. 'inside' refers to the same side as the inset property (attaching the positioned box to the "inside" of the anchor box), while 'outside' refers to the opposite.

  'top'
  'right'
  'bottom'
  'left'
  Refers to the specified side of the anchor box.

  > NOTE: These are only usable in the inset properties in the matching axis. For example, 'left' is usable in 'left', 'right', or the logical inset properties that refer to the horizontal axis.

  'start'
  'end'
  'self-start'
  'self-end'
  Refers to one of the sides of the anchor box in the same axis as the inset property it's used in, by resolving the keyword against the writing mode of either the positioned box (for 'self-start' and 'self-end') or the positioned box's containing block (for 'start' and 'end').

  '<percentage>'
  'center'
  Refers to a position a corresponding percentage between the 'start' and 'end' sides, with '0%' being equivalent to 'start' and '100%' being equivalent to 'end'.

  'center' is equivalent to '50%'.

- the optional <length-percentage> final argument is a fallback value, specifying what the function should compute to if it's an unresolvable anchor function.

An 'anchor()' function representing a resolvable anchor function resolves at computed value time (using style & layout interleaving) to the <length> that would align the edge of the positioned boxes' inset-modified containing block corresponding to the property the function appears in with the specified edge of the target anchor element's anchor box.

NOTE: This means that transitions or animations of a property using an anchor function will work "as expected" for all sorts of possible changes: the anchor box moving, anchor elements being added or removed from the document, the 'anchor-name' property being changed on anchors, etc.

EXAMPLE 5 ¶

For example, in '.bar { inset-block-start: anchor(--foo block-start); }', the 'anchor()' will resolve to the length that'll line up the `.bar` element's block-start edge with the '--foo' anchor's block-start edge.

On the other hand, in '.bar { inset-block-end: anchor(--foo block-start); }', it will instead resolve to the length that'll line up the `.bar` element's *block-end* edge with the '--foo' anchor's block-start edge.

Since 'inset-block-start' and 'inset-block-end' values specify insets from different edges (the block-start and block-end of the element's containing block, respectively), the same 'anchor()' will usually resolve to different lengths in each.

EXAMPLE 6 ¶

Because the 'anchor()' function resolves to a <length>, it can be used in math functions like any other length.

ISSUE 2 Add a better example; this one can be accomplished easily with 'anchor-center'. [Issue #10776] ¶

For example, the following will set up the element so that its inset-modified containing block is centered on the anchor box and as wide as possible without overflowing the containing block:

```
.centered-message {
  position: fixed;
  max-width: max-content;
  justify-self: center;

  --center: anchor(--x 50%);
  --half-distance: min(
    abs(0% - var(--center)),
    abs(100% - var(--center))
  );
  left: calc(var(--center) - var(--half-distance));
  right: calc(var(--center) - var(--half-distance));
  bottom: anchor(--x top);
}
```

This might be appropriate for an error message on an `<input>` element, for example, as the centering will make it easier to discover which input is being referred to.

§ 3.2.1. Resolution of 'anchor()'

An 'anchor()' function is a ***resolvable anchor function*** only if all the following conditions are true:

- It's applied to an absolutely positioned box.
- If its <anchor-side> specifies a physical keyword, it's specified in an inset property applicable to that axis. (For example, 'left' can only be used in 'left', 'right', or a logical inset property in the horizontal axis.)
- There is a target anchor element for the box it's used on, and the <anchor-name> value specified

in the function.

If any of these conditions are false, the ‘anchor()’ function computes to its specified fallback value. If no fallback value is specified, it makes the declaration referencing it invalid at computed-value time.

§ 3.3. Taking Scroll Into Account

For performance reasons, implementations usually perform scrolling on a separate scrolling/"compositing" thread, which has very limited capabilities (simple movement/transforms/etc., but no layout or similar expensive operations) and thus can be relied upon to respond to scrolling fast enough to be considered "instant" to human perception.

If scrolling just causes an anchor-positioned element to *move*, there is in theory no issue; the movement can be performed on the scrolling thread so the positioned element moves smoothly with the scrolling content. However, anchor positioning allows an element to make the positions of its own opposite edges depend on things in *different* scrolling contexts, which means scrolling could move just *one* edge and cause a size change, and thus perform layout. This can’t be performed on the scrolling thread!

To compensate for this, while still allowing as much freedom to anchor to various elements as possible, anchor positioning uses a combination of remembered scroll offsets and compensating for scroll.

The details here are technical, but the gist is:

- When a positioned element is first displayed, or when it changes fallbacks, its position is correctly calculated according to the up-to-date position of all anchor references.

  If these anchor references are in a different scroll context, their total scroll offsets are *memorized*, and layout will continue using those memorized offsets, even if those elements are scrolled later. (Only the scroll offsets are memorized; their actual laid-out positions are freshly calculated each time and remain accurate.) They'll only recalculate if the positioned element stops being displayed and starts again, or changes fallbacks.

- The one exception to this is the default anchor element; if it's scrolled away from its remembered scroll offset, the positioned element moves with it. Because this is *purely* a shift in position, the positioned element can't change size or otherwise require layout in response.

The end result is that anchor positioning should generally "just work", regardless of what the element is anchoring to, but it might be limited in how it can respond to scrolling.

An ***anchor recalculation point*** occurs for an absolutely positioned element whenever that element begins generating boxes (aka switches from 'display:none' or 'display:contents' to any other 'display' value), identical to when it starts running CSS animations.

An anchor recalculation point also occurs for an element when determining position fallback styles for that element; if it changes fallback styles as a result, it uses the result of the anchor recalculation point associated with the chosen set of fallback styles.

When an anchor recalculation point occurs for an element *abspos*, then for every element *anchor* referenced by one of *abspos*'s anchor references, it associates a ***remembered scroll offset*** equal to the *current* sum of the scroll offsets of all scroll container ancestors of *anchor*, up to but not including *abspos*'s containing block. The remembered scroll offset also accounts for other scroll-dependent positioning changes, such as 'position: sticky'. If *abspos* has a default anchor element, it always calculates a remembered scroll offset for it, even if *abspos* doesn't actually have an anchor reference to it.

All anchor references are calculated as if all scroll containers were at their initial scroll position, and then have their associated remembered scroll offset added to them.

ISSUE 3 Transforms have the same issue as scrolling, so Anchor Positioning similarly doesn't pay attention to them normally. Can we go ahead and incorporate the effects of transforms here?

The above allows a positioned element to respond to the scroll positions of its anchor references *once*, but if any of them are scrolled, the positioned element will no longer appear to be anchored to them (tho it will continue to respond to their non-scrolling movement). While this problem can't be solved *in general*, we *can* respond to the scrolling of *one* anchor reference; specifically, the default anchor element:

An absolutely positioned box *abspos* ***compensates for scroll*** in the horizontal or vertical axis if both of the following conditions are true:

- *abspos* has a default anchor box.
- *abspos* has an anchor reference to its default anchor box or at least to something in the same scrolling context, aka at least one of:
  - *abspos*'s used self-alignment property value in that axis is 'anchor-center';
  - *abspos* has a non-'none' value for 'position-area'
  - at least one 'anchor()' function on *abspos*'s used inset properties in the axis refers to a target anchor element with the same nearest scroll container ancestor as *abspos*'s default anchor box.

NOTE: If *abspos* has a position options list, then whether it compensates for scroll in an axis is also affected by the applied fallback style.

abspos's ***default scroll shift*** is a pair of lengths for the horizontal and vertical axises, respectively. Each length is calculated as:

- If *abspos* is compensating for scroll in the axis, then the length is the difference between the remembered scroll offset of the default anchor element and what its current remembered scroll offset would be if it were recalculated.
- Otherwise, the length is 0.

After layout has been performed for *abspos*, it is additionally shifted by the default scroll shift, as if affected by a transform (before any other transforms).

ISSUE 4 Define the precise timing of the snapshot: updated each frame, before style recalc. ¶

ISSUE 5 Similar to remembered scroll offset, can we pay attention to transforms on the default anchor element? ¶

NOTE: While remembered scroll offsets affect the value of 'anchor()' functions, default scroll shift directly shifts the element, *after* determining the value of its inset properties, applying alignment, etc. This is *usually* indistinguishable, but cases like 'round(anchor(outside), 50px)', which transform the default anchor element's position in a non-linear fashion, will expose the difference in behavior.

4. Anchor-Based Alignment

4.1. Area-specific Default Alignment

When 'position-area' is not 'none', the used value of 'normal' self-alignment changes depending on the <position-area> value, to align the box towards the anchor:

- If the only the center track in an axis is selected, the default alignment in that axis is 'center'.
- If all three tracks are selected, the default alignment in that axis is 'anchor-center'.
- Otherwise, the default alignment in that axis is toward the non-specified side track: if it's specifying the "start" track of its axis, the default alignment in that axis is 'end'; etc.

However, if only one inset property in the relevant axis is 'auto', the default alignment is instead towards the edge with the non-'auto' inset; and this is an 'unsafe' alignment.

NOTE: This single-'auto' behavior preserves the way a single specified inset controls the position of an absolutely positioned box.

EXAMPLE 7

For example, assuming an English-equivalent writing mode (horizontal-tb, ltr), then the value 'span-x-start top' resolves to the 'start' region of the vertical axis, and the 'start' and 'center' regions of the horizontal axis, so the default alignments will be 'align-self: end' (making the box's bottom margin edge flush with the bottom of the 'top' region) and 'justify-self: end' (making the box's end-side margin edge flush with the end side of the 'span-start' region).

If the box overflows its inset-modified containing block, but would still fit within its original containing block, by default it will "shift" to stay within its original containing block, even if that violates its normal alignment. See *CSS Box Alignment 3* § 4.4 Overflow Alignment: the safe and unsafe keywords and scroll safety limits for details.

This behavior makes it more likely that positioned boxes remain visible and within their intended bounds, even when their containing block ends up smaller than anticipated.

For example, a 'position-area: bottom span-right' value lets the positioned box stretch from its anchor's left edge to its containing block's right edge, and left-aligns it in that space by default. But if the positioned box is larger than that space (such as if the anchor is very close to the right edge of the screen), it will shift leftwards to stay visible.

§ 4.2. Centering on the Anchor: the 'anchor-center' alignment value

Name:	***'justify-self'***, ***'align-self'***, ***'justify-items'***, ***'align-items'***
New values:	anchor-center

The self-alignment properties allow an absolutely positioned box to align itself within the inset-modified containing block. The existing values, plus carefully chosen inset properties, are usually enough for useful alignment, but a common case for anchored positioning—centering over the anchor box—requires careful and somewhat complex set-up to achieve.

The new ***'anchor-center'*** value makes this case extremely simple: if the positioned box has a default anchor box, then it is centered (insofar as possible) over the default anchor box in the relevant axis. Additionally:

- The scrollable containing block is used in place of the local containing block where applicable, so that the entire scrollable overflow area (typically) is available for positioning.
- The used value of any 'auto' inset properties and 'auto' margin properties resolves to '0'.

If the box is not absolutely positioned, or does not have a default anchor box, this value behaves as 'center' and has no additional effect on how inset properties resolve.

NOTE: When using ‘anchor-center’, by default if the anchor is too close to the edge of the box’s original containing block, it will “shift” from being purely centered, in order to remain within the original containing block. See *CSS Box Alignment 3* § 4.4 Overflow Alignment: the safe and unsafe keywords and scroll safety limits for more details.

§ 5. Anchor-Based Sizing

An absolutely positioned box can use the ***‘anchor-size()’*** function in its sizing properties to refer to the size of one or more anchor boxes. The ‘anchor-size()’ function resolves to a <length>. It is only allowed in the accepted @position-try properties (and is otherwise invalid).

§ 5.1. The ‘anchor-size()’ Function

Name:	‘width’, ‘height’, ‘min-width’, ‘min-height’, ‘max-width’, ‘max-height’, ‘top’, ‘left’, ‘right’, ‘bottom’, ‘margin-top’, ‘margin-left’, ‘margin-right’, ‘margin-bottom’
New values:	<anchor-size()>

```
anchor-size() = anchor-size( [ <anchor-name> || <anchor-size> ]? , <length-percenta
<anchor-size> = width | height | block | inline | self-block | self-inline
```

The ‘anchor-size()’ function is similar to ‘anchor()’, and takes the same arguments, save that the <anchor-side> keywords are replaced with <anchor-size>, referring to the distance between two opposing sides.

The physical <anchor-size> keywords (***‘width’*** and ***‘height’***) refer to the width and height, respectively, of the target anchor element. Unlike ‘anchor()’, there is no restriction on having to match axises; for example, ‘width: anchor-size(--foo height);’ is valid.

The logical <anchor-size> keywords (***‘block’***, ***‘inline’***, ***‘self-block’***, and ***‘self-inline’***) map to one of the physical keywords according to either the writing mode of the box (for ‘self-block’ and ‘self-inline’) or the writing mode of the box’s containing block (for ‘block’ and ‘inline’).

If the <anchor-size> keyword is omitted, it defaults to behaving as whatever keyword matches the axis of the property that ‘anchor-size()’ is used in. (For example, ‘width: anchor-size()’ is equivalent to

'width: anchor-size(width)'.)

An 'anchor-size()' function representing a resolvable anchor-size function resolves at computed value time (via style & layout interleaving) to the <length> separating the relevant edges (either left and right, or top and bottom, whichever is in the specified axis) of the target anchor element's anchor box.

5.1.1. Resolution of 'anchor-size()'

An 'anchor-size()' function is a ***resolvable anchor-size function*** only if all the following conditions are true:

- It's applied to an absolutely positioned box.
- There is a target anchor element for the box it's used on, and the <anchor-name> value specified in the function.

If any of these conditions are false, the 'anchor-size()' function resolves to its specified fallback value. If no fallback value is specified, it makes the declaration referencing it invalid at computed-value time.

6. Overflow Management

Anchor positioning, while powerful, can also be unpredictable. The anchor box might be anywhere on the page, so positioning a box in any particular fashion (such as above the anchor, or the right of the anchor) might result in the positioned box overflowing its containing block or being positioned partially off screen.

To ameliorate this, an absolutely positioned box can use the 'position-try-fallbacks' property to specify additional position options (variant sets of positioning/alignment properties generated from the box's existing styles, or specified in '@position-try' rules) that the UA can try if the box overflows its initial position. Each is applied to the box, one by one in the order specified by 'position-try-order', and the first that doesn't cause the box to overflow its containing block is taken as the winner.

Once an option has been chosen, the element keeps those styles until it overflows again, even if an earlier (and presumably more desirable) option again becomes available without causing overflow. (See remember or forget the last successful position option.)

EXAMPLE 8 ¶

For example, the following CSS will first attempt to position a "popover" below the element, but if it doesn't fit on-screen will switch to being above. It defaults to start-aligning with the anchor, but will switch to end-aligning if that doesn't fit. If it doesn't fit on *either* side, it will take the whole horizontal space, while centering on the anchor as much as possible (thanks to § 4.1 Area-specific Default Alignment).

```
#myPopover {
  position: fixed;
  position-anchor: --button;
  position-area: bottom span-x-end;
  position-try-fallbacks: flip-x, flip-y, flip-x flip-y, bottom, top;

  /* The popover is at least as wide as the button */
  min-width: anchor-size(width);

  /* The popover is at least as tall as 2 menu items */
  min-height: 6em;
}
```

ISSUE 6 Add a picture! ¶

§ 6.1. Giving Fallback Options: the 'position-try-fallbacks' property

Name:	***'position-try-fallbacks'***
Value:	none ¦ [[<dashed-ident> || <try-tactic>] ¦ <position-area>]#
Initial:	none
Applies to:	absolutely positioned boxes
Inherited:	no
Percentages:	n/a
Computed value:	as specified

Canonical order:	per grammar
Animation type:	discrete

This property provides a list of alternate positioning styles to try when the absolutely positioned box overflows its inset-modified containing block. This ***position options list*** initially contains a single position option generated from the element's ***fallback base styles***, i.e. the computed styles without applying 'position-try-fallbacks'.

Each comma-separated entry in the list is a separate option: either the name of a '@position-try' block, or a <try-tactic> representing an automatic transformation of the box's existing computed style.

Values have the following meanings:

'none'
: The property has no effect; the box's position options list is empty.

'<dashed-ident>'
: If there is a '@position-try' rule with the given name, its associated position option is added to the position options list.

  Otherwise, this value has no effect.

'<try-tactic>'
: Automatically creates a position option by executing the specified try tactic to the box's base styles, then adding the constructed position option to the box's position options list.

  <try-tactic> = flip-block || flip-inline || flip-start || flip-x || flip-y

  'flip-block'
  : swaps the values in the block axis (between, for example, 'margin-block-start' and 'margin-block-end'), essentially mirroring across an inline-axis line.

  'flip-inline'
  : swaps the values in the inline axis, essentially mirroring across a block-axis line.

  'flip-x'
  : swaps the values in the horizontal axis (between, for example, 'margin-left' and 'margin-right'), essentially mirroring across a vertical-axis line.

  'flip-y'
  : swaps the values in the vertical axis, essentially mirroring across a horizontal-axis line.

  'flip-start'

swaps the values of the start properties with each other, and the end properties with each other (between, for example, 'margin-block-start' and 'margin-inline-start'), essentially mirroring across a diagonal drawn from the start-start corner to the end-end corner.

If multiple keywords are given, the transformations are composed in order to produce a single position option. Logical directions are resolved against the writing mode of the containing block.

'<dashed-ident> || <try-tactic>'

Combines the effects of the previous two options: if there is a '@position-try' rule with the given name, applies its position option to the base style, then transforms it according to the specified <try-tactic> and adds the result to the box's position options list.

Otherwise, does nothing.

'<position-area>'

Automatically creates a position option composed solely of a 'position-area' property with the given value.

§ 6.2. Determining Fallback Order: the 'position-try-order' property

Name:	***'position-try-order'***
Value:	normal | <try-size>
Initial:	normal
Applies to:	absolutely positioned boxes
Inherited:	no
Percentages:	n/a
Computed value:	as specified
Canonical order:	per grammar
Animation type:	discrete

This property allows an element to sort its position options by the available space they define, if it's

more important for the box to have as much space as possible rather than strictly following the order declared in ‘position-try-fallbacks’.

<try-size> = most-width | most-height | most-block-size | most-inline-size

‘normal’
: Try the position options in the order specified by ‘position-try-fallbacks’.

‘most-width’
‘most-height’
‘most-block-size’
‘most-inline-size’
: For each entry in the position options list, apply that position option to the box, and find the inset-modified containing block size that results from those styles (treating ‘auto’ ‘inset’ values as zero). Stably sort the position options list according to this size, with the largest coming first.

  Logical directions are resolved against the writing mode of the containing block.

EXAMPLE 9 ¶

For example, the following styles will initially position the popup list either above or below its anchoring button, depending on which option gives it the most space.

```
.anchor { anchor-name: --foo; }
.list {
  position: fixed;
  position-anchor: --foo;
  position-area: block-end span-inline-end;
  position-try-fallbacks: --bottom-scrollable, flip-block, --top-scrollable;
  position-try-order: most-height;
}
@position-try --bottom-scrollable {
  align-self: stretch;
}
@position-try --top-scrollable {
  position-area: block-start span-inline-end;
  align-self: stretch;
}
```

The base styles and the '--bottom-scrollable' option have the same available height, since in both cases the inset-modified containing block stretches from the anchor to the edge of the containing block. Likewise the 'flip-block' option and the '--top-scrollable' options have the same available height. Because 'position-try-order' uses a stable sort, these pairs will each retain their relative positions in the list, with the '*-scrollable' option coming later; and the pair that has the most space available will come first.

This causes the box to first try to align against the anchor at its natural height on whichever side is larger (using the base styles or 'flip-block' styles) but if that causes overflow, it'll fall back to just filling the same space and being scrollable instead (using the matching '*-scrollable' styles), thus never overflowing and trying to move to the smaller space.

§ 6.3. The 'position-try' Shorthand

Name:	***'position-try'***
Value:	<'position-try-order'>? <'position-try-fallbacks'>

Initial:	see individual properties
Applies to:	see individual properties
Inherited:	see individual properties
Percentages:	see individual properties
Computed value:	see individual properties
Animation type:	see individual properties
Canonical order:	per grammar

This shorthand sets both 'position-try-fallbacks' and 'position-try-order'. If <'position-try-order'> is omitted, it's set to the property's initial value.

§ 6.4. The '@position-try' Rule

The ***'@position-try'*** rule defines a ***position option*** with a given name, specifying one or more sets of positioning properties that can be applied to a box via 'position-try-fallbacks',

The syntax of the '@position-try' rule is:

```
@position-try <dashed-ident> {
  <declaration-list>
}
```

The <dashed-ident> specified in the prelude is the rule's name. If multiple '@position-try' rules are declared with the same name, they cascade the same as '@keyframe' rules do.

The '@position-try' rule only ***accepts*** the following properties:

- inset properties
- margin properties
- sizing properties
- self-alignment properties

- ‘position-anchor’
- ‘position-area’

It is invalid to use ‘!important’ on the properties in the <declaration-list>. Doing so causes the property it is used on to become invalid, but does not invalidate the ‘@property-try’ rule as a whole.

All of the properties in a ‘@position-try’ are applied to the box as part of the ***Position Fallback Origin***, a new cascade origin that lies between the Author Origin and the Animation Origin.

Similar to the Animation Origin, use of the ‘revert’ value acts as if the property was part of the Author Origin, so that it instead reverts back to the User Origin. (As with the Animation Origin, however, ‘revert-layer’ has no special behavior and acts as specified.)

> NOTE: The accepted @position-try properties are the smallest group of properties that affect just the size and position of the box itself, without otherwise changing its contents or styling. This significantly simplifies the implementation of position fallback while addressing the fundamental need to move an anchor-positioned box in response to available space. Since these rules override normal declarations in the Author Origin, this also limits the poor interactions of ‘@position-try’ declarations with the normal cascading and inheritance of other properties. It is expected that a future extension to container queries will allow querying an element based on the position fallback it’s using, enabling the sort of conditional styling not allowed by this restricted list.

> NOTE: If multiple elements want to use the same ‘@position-try’ rules, but relative to their own anchor elements, omit the <anchor-name> in ‘anchor()’ and specify each box’s anchor in ‘position-anchor’ instead.

> NOTE: The most common types of fallback positioning (putting the positioned box on one side of the anchor normally, but flipping to the opposite side if needed) can be done automatically with keywords in ‘position-try-fallbacks’, without using ‘@position-try’ at all.

§ 6.5. Applying Position Fallback

When a positioned box (after applying any default scroll shift) overflows its inset-modified containing block, and has more than one position option in its position options list, it determines position fallback styles to attempt to find an option that avoids overflow. The resulting styles are applied to the element via interleaving, so they affect computed values (and can trigger transitions/etc) even though they depend on layout and used values.

Implementations may choose to impose an implementation-defined limit on the length of position options lists, to limit the amount of excess layout work that may be required. This limit must be *at least* five.

To ***determine position fallback styles*** for an element *abspos*:

1. Let *current styles* be the current used styles of *abspos* (which might be the result of earlier fallback).
2. For each *option* in the position options list:
    1. If *option* is currently *abspos*'s last successful position option, continue.
    2. Let *adjusted styles* be the result of applying a position option *option* to *abspos*.
    3. Let *el rect* be the size and position of *abspos*'s margin box, and *cb rect* be the size and position of *abspos*'s inset-modified containing block, when laid out with *adjusted styles*.
    4. If *cb rect* was negative-size in either axis and corrected into zero-size, continue.

        NOTE: This prevents a zero-size *el rect* from still being considered "inside" a negative-size *cb rect* and getting selected as a successful option.

    5. If *el rect* is not fully contained within *cb rect*, continue.
    6. Return *adjusted styles*, along with the associated set of remembered scroll offsets that were hypothetically calculated for them.
3. Assert: The previous step finished without finding a position option that avoids overflow.
4. Return *current styles*.

NOTE: Descendants overflowing *el* don't affect this calculation, only *el*'s own margin box.

NOTE: Because we purposely skip the position option currently in effect, it doesn't get its remembered scroll offsets updated; if none of the other fallbacks work and we stick with the current styles, all the remembered scroll offsets stay the same.

During a full layout pass, once a box has determined its fallback styles (or determined it's not using any), laying out later boxes cannot change this decision.

EXAMPLE 10 ¶

For example, say you have two positioned boxes, A and B, with A laid out before B. If B overflows and causes A's containing block to gain scrollbars, this *does not* cause A to go back and re-determine its fallback styles in an attempt to avoid overflowing. (At best, this can result in exponential layout costs; at worst, it's cyclic and will never settle.)

Layout does not "go backward", in other words.

§ 6.5.1. Maintaining and Clearing Fallback Choices

Some changes to a box have a particularly direct effect on determining position fallback styles and thus trigger special behavior. These ***fallback-sensitive changes*** include:

- Its computed 'position' value has changed, its containing block association has changed, or it no longer generates a box.
- Its computed value for any longhand of 'position-try' has changed.
- Its computed value for any accepted @position-try property has changed.
- Any of the '@position-try' rules referenced by it have been added, removed, or mutated.

§ *6.5.1.1. Recording the last successful position option*

In order to maintain layout stability as much as possible, determining position fallback styles prioritizes the last successful position option, which is determined as follows:

At the time that `ResizeObserver` events are determined and delivered, the box must ***record the last successful position option*** as follows:

- If *el* has a last successful position option remove its last successful position option if any fallback-sensitive changes have occurred. Then, determine position fallback styles for *el* and set its last successful position option to the set of accepted @position-try properties (and values) that it's now using.
- Otherwise, if a box *el* is absolutely positioned, set its ***last successful position option*** to the set of accepted @position-try properties (and values) that it's currently using.

NOTE: The timing of this recording/removal is intentionally identical to the treatment of last remembered sizes.

ISSUE 7 The following sections attempt to clarify the interaction with transitions and animations. [Issue #13048]

§ *6.5.1.2. Suspending Fallback During Transitions*

The UA must determine position fallback styles for both the start and end states of a transition (see [CSS-TRANSITIONS-1]) that includes properties that could cause a fallback-sensitive change.

During a transition for properties that could cause a fallback-sensitive change, however, determining position fallback styles and recording the last successful position option are suspended.

§ *6.5.1.3. Suspending Fallback During Animations*

If an animation (see [CSS-ANIMATIONS-1] and [WEB-ANIMATIONS-1]) affects any properties that could cause a fallback-sensitive change, then the UA must determine position fallback styles for the keyframes that contain those properties (only). As fallback determination is order-sensitive, later keyframes must take into account the result of earlier keyframes.

While animating between these keyframes, however, determining position fallback styles and recording the last successful position option are suspended.

§ 6.5.2. Applying Position Options

To ***apply a position option*** to a box's element *el*, given a position option *new styles*:

1. With *new styles* inserted into the cascade in the position fallback origin, resolve the cascade, and perform enough layout to determine *el*'s used styles.

   For the purpose of calculating these styles, a *hypothetical* anchor recalculation point is calculated, and the resulting hypothetical remembered scroll offsets are used to determine *el*'s styles.

2. Return *el*'s used styles.

To ***execute a try-tactic*** to a set of *styles* of a box's element *el*, between two directions *directions*, returning a set of transformed styles:

0. If *directions* are opposites along the same axis, they are "opposing". Otherwise (when they are specifying different axises), they are "perpendicular".
1. Determine the specified values of the accepted @position-try properties on *el*, and let *styles* be the result.
2. Substitute variables, 'env()' functions, and similar arbitrary substitution functions in *styles*.

   For 'env()' functions, if the referenced environment variable is associated with a direction or axis (such as 'safe-area-inset-top'), switch the referenced environment variable corresponding to *directions*.

   EXAMPLE 11 ¶
   For example, if 'top: env(safe-area-inset-top);' is specified, and *directions* are up and left, the 'env()' will resolve as if 'env(safe-area-inset-left)' had been specified instead. (And then, in the next step, will actually swap into the 'left' property.)

3. Swap the values of the *styles* between the associated properties corresponding to *directions*.

   EXAMPLE 12 ¶
   For example, if "top" and "left" are being swapped, then the values of 'margin-top' and 'margin-left' are swapped, 'width' and 'height' are swapped, etc.

   NOTE: If the directions are opposites along the same axis, some properties (like 'width' or 'align-self') wont' swap, since they're associated with themselves across the two directions, but their values might be changed by the next step.

4. Modify the values of the properties as they swap to match the new directions, as follows:

   - For inset properties, change the specified side in 'anchor()' functions to maintain the same relative relationship to the new direction that they had to the old.

     If a <percentage> is used, and *directions* are opposing, change it to '100%' minus the original percentage.

> EXAMPLE 13 ¶
> For example, if "top" and "left" are being swapped, then ‘margin-top: anchor(bottom)’ will become ‘margin-left: anchor(right)’.
>
> If "top" and "bottom" are being swapped, then ‘margin-top: anchor(20%)’ will become ‘margin-bottom: anchor(80%)’.

- For sizing properties, change the specified axis in ‘anchor-size()’ functions to maintain the same relative relationship to the new direction that they had to the old.

> EXAMPLE 14 ¶
> For example, if "top" and "left" are being swapped, then ‘width: anchor-size(width)’ will become ‘height: anchor-size(height)’.

- For the self-alignment properties, if *directions* are opposing, change the specified <self-position> (or ‘left’/‘right’ keywords), if any, to maintain the same relative relationship to the new direction that they had to the old.

> EXAMPLE 15 ¶
> For example, if "top" and "bottom" are being swapped, then ‘align-self: start’ will become ‘align-self: end’.
>
> However, ‘align-self: center’ will remain unchanged, as it has the same relationship to both directions.
>
> Similarly, ‘align-self: first baseline’ will remain unchanged, as it’s a <baseline-position> rather than a <self-position>.

- For ‘position-area’, change the value so that the selected rows/columns of the position-area grid maintain the same relative relationship to the new direction that they had to the old.

> EXAMPLE 16 ¶
> For example, if "top" and "left" are being swapped, then ‘position-area: left span-bottom’ will become ‘position-area: top span-right’.

5. Return *styles*.

§ 6.6. Conditional Hiding: the ‘position-visibility’ property

Name:	***'position-visibility'***
Value:	always ¦ [anchors-valid ¦¦ anchors-visible || no-overflow]
Initial:	anchors-visible
Applies to:	absolutely positioned boxes
Inherited:	no
Percentages:	n/a
Computed value:	as specified
Canonical order:	per grammar
Animation type:	discrete

There are some conditions in which it might not make sense to display an absolutely positioned box. This property allows such boxes to be made conditionally visible, depending on some commonly needed layout conditions.

'always'
: This property has no effect. (The box is displayed without regard for its anchors or its overflowing status.)

'anchors-valid'
: If any of the box's required anchor references do not resolve to a target anchor element, the box's 'visibility' property computes to 'force-hidden'.

> ISSUE 8 What is a ***required anchor reference***? 'anchor()' functions that don't have a fallback value; the default anchor *sometimes*? Need more detail here. ¶

> ISSUE 9 *Any* anchors are missing, or *all* anchors are missing? I can see use-cases for either, potentially. Do we want to make a decision here, or make it controllable somehow? ¶

'anchors-visible'
: If the box has a default anchor box but that anchor box is invisible or clipped by intervening boxes, the box's 'visibility' property computes to 'force-hidden'.

'no-overflow'

If the box overflows its inset-modified containing block even after applying 'position-try', the box's 'visibility' property computes to 'force-hidden'.

An anchor box *anchor* is ***clipped by intervening boxes*** relative to a positioned box *abspos* relying on it if *anchor*'s ink overflow rectangle is fully clipped by a box which is an ancestor of *anchor* but a descendant of *abspos*'s containing block. Clipping in this case refers only to the same clipping effects that are (by default) checked by `IntersectionObserver`, i.e. clipping due to 'clip-path', 'overflow', or other effects (such as paint containment) that clip to the overflow clip edge. If *anchor* has non-zero area, it must also have a non-zero intersection area to be considered not fully clipped.

Whether or not *anchor* is clipped by intervening boxes must be checked after updating content relevancy for a document (see 'content-visibility' in [css-contain-2]) and running any `ResizeObserver`, but before running any `IntersectionObserver`. It may also be checked at other times to improve responsiveness.

NOTE: This means that if an abspos is next to its anchor in the DOM, for example, it'll remain visible even if its default anchor is scrolled off, since it's clipped by the same scroller anyway.

ISSUE 10 Make sure this definition of clipped is consistent with View Transitions, which wants a similar concept.

NOTE: This ensures that in a "chained anchor" situation, if the first abspos is hidden due to this property (due to its anchor being scrolled off), then another abspos using it as an anchor will also be hidden, rather than *also* floating in a nonsensical location.

§ 7. Accessibility Implications

CSS Anchor Positioning does not create, delete, or alter any accessibility bindings between elements. Authors must use appropriate markup features to control such bindings.

Because it can be used in many different ways for many different use cases, CSS Anchor Positioning does not automatically establish any semantic relationship between a positioned box and any of its anchors. For example, the *visual* anchoring relationship in a design might be between an element and its *semantic* anchor, or it might connect the element to an ancestor, sibling, or descendant of the semantic anchor, depending on the desired visual effect. Siimilarly, a design might opt out of a visual anchoring relationship even while there is a semantic one, or vice versa.

Authors must not rely on the visual connections implied by CSS positioning to link elements together semantically. Without appropriate markup, the elements linked visually have *no* meaningful DOM relationship—which if there *is* a meaningful relationship, can make them difficult or impossible to use in non-visual user agents, like screen readers, or in non-graphical navigation modes, such as tab navigation.

Many features on the web platform, both existing and upcoming, allow establishing semantic connections explicitly, so that non-visual user agents can also benefit. For example, the Popover API in HTML automatically links the invoker button to the popover element, including automatically adjusting tabbing order; it *also* establishes the invoker button as the implicit anchor element for the popover, making it easy to use Anchor Positioning as well.

> ISSUE 11 Add a popover example. ¶

In more general cases, ARIA features such as the `aria-details` or `aria-describedby` attributes on an anchor element can create connections in a slightly more manual fashion. In concert with the `role` attribute on the positioned element, it allows non-visual user agents to tell their users about the relationship between the elements and let them automatically navigate between them.

However, authors should not overuse such features either, since overburdening the page with extra, unnecessary semantic connections can also make the page difficult to comprehend.

> ISSUE 12 Suggestions for ways to improve this section, especially author guidance and examples of best practices for common use cases, is welcome. [Issue #10311] ¶

§ 8. DOM Interfaces

§ 8.1. The CSSPositionTryRule interface

The `CSSPositionTryRule` interface represents the '@position-try' rule:

```
[Exposed=Window]
interface CSSPositionTryRule : CSSRule {
  readonly attribute CSSOMString name;
  [SameObject, PutForwards=cssText] readonly attribute CSSPositionTryDescriptors
};

[Exposed=Window]
interface CSSPositionTryDescriptors : CSSStyleDeclaration {
```

```
attribute CSSOMString margin;
attribute CSSOMString marginTop;
attribute CSSOMString marginRight;
attribute CSSOMString marginBottom;
attribute CSSOMString marginLeft;
attribute CSSOMString marginBlock;
attribute CSSOMString marginBlockStart;
attribute CSSOMString marginBlockEnd;
attribute CSSOMString marginInline;
attribute CSSOMString marginInlineStart;
attribute CSSOMString marginInlineEnd;
attribute CSSOMString margin-top;
attribute CSSOMString margin-right;
attribute CSSOMString margin-bottom;
attribute CSSOMString margin-left;
attribute CSSOMString margin-block;
attribute CSSOMString margin-block-start;
attribute CSSOMString margin-block-end;
attribute CSSOMString margin-inline;
attribute CSSOMString margin-inline-start;
attribute CSSOMString margin-inline-end;
attribute CSSOMString inset;
attribute CSSOMString insetBlock;
attribute CSSOMString insetBlockStart;
attribute CSSOMString insetBlockEnd;
attribute CSSOMString insetInline;
attribute CSSOMString insetInlineStart;
attribute CSSOMString insetInlineEnd;
attribute CSSOMString top;
attribute CSSOMString left;
attribute CSSOMString right;
attribute CSSOMString bottom;
attribute CSSOMString inset-block;
attribute CSSOMString inset-block-start;
attribute CSSOMString inset-block-end;
attribute CSSOMString inset-inline;
attribute CSSOMString inset-inline-start;
attribute CSSOMString inset-inline-end;
attribute CSSOMString width;
attribute CSSOMString minWidth;
attribute CSSOMString maxWidth;
attribute CSSOMString height;
attribute CSSOMString minHeight;
```

```
  attribute CSSOMString maxHeight;
  attribute CSSOMString blockSize;
  attribute CSSOMString minBlockSize;
  attribute CSSOMString maxBlockSize;
  attribute CSSOMString inlineSize;
  attribute CSSOMString minInlineSize;
  attribute CSSOMString maxInlineSize;
  attribute CSSOMString min-width;
  attribute CSSOMString max-width;
  attribute CSSOMString min-height;
  attribute CSSOMString max-height;
  attribute CSSOMString block-size;
  attribute CSSOMString min-block-size;
  attribute CSSOMString max-block-size;
  attribute CSSOMString inline-size;
  attribute CSSOMString min-inline-size;
  attribute CSSOMString max-inline-size;
  attribute CSSOMString placeSelf;
  attribute CSSOMString alignSelf;
  attribute CSSOMString justifySelf;
  attribute CSSOMString place-self;
  attribute CSSOMString align-self;
  attribute CSSOMString justify-self;
  attribute CSSOMString positionAnchor;
  attribute CSSOMString position-anchor;
  attribute CSSOMString positionArea;
  attribute CSSOMString position-area;
};
```

Its ***name*** attribute represents the name declared in the rule's prelude.

Its ***style*** attribute represents the properties declared in the rule's body, in the specified order. On getting, it must return a `CSSPositionTryDescriptors` object for the '@position-try' at-rule, with the following properties:

computed flag
: Unset

readonly flag
: Unset

declarations
: The declared descriptors in the rule, in specified order.

parent CSS rule

The context object

owner node

Null

9. Appendix: Style & Layout Interleaving

Style & layout interleaving is a technique where a style update can occur on a subtree during the layout process, resulting in retroactive updates to elements' computed styles.

ISSUE 13 This is not the correct spec for this concept, it should probably go in Cascade, but I need a sketch of it to refer to.

NOTE: Style & layout interleaving is already used with container queries and container query lengths. A length like '10cqw' is resolved into a computed length using layout information about the query container's size, which can thus trigger transitions when the container changes size between layouts.

The accepted @position-try properties are also interleaved when resolving fallback (see 'position-try').

ISSUE 14 Obviously this needs way more details filled in, but for now "act like you already do for container queries" suffices. That behavior is also undefined, but at least it's interoperable (to some extent?).

10. Security Considerations

No Security issues have been raised against this document.

11. Privacy Considerations

No Privacy issues have been raised against this document.

12. Changes

Significant changes since the 22 December 2025 Working Draft:

- Clarify positoin fallback interaction with transitions and animation. (Issue 13048)
- Clarify that intersection must be non-zero (when anchor box is non-zero) for 'anchors-visible'. (Issue 13176)

Significant changes since the 7 October 2025 Working Draft:

- Add 'flip-x' and 'flip-y' to 'position-try-fallbacks'. (Issue 12869)
- Define 'anchor-center' to also use the scrollable containing block so that it doesn't trigger overflow alignment when positioned outside the local containing block. (Issue 12952)
- Resolve 'auto' margins to zero when 'position-area' or 'anchor-center' is in effect, due to the ill-considered HTML UA default stylesheet rules for popovers. Also drop the 'dialog' alignment value which was the previous attempt to address this problem. (Issue 10258)
- Add 'none' value to 'position-anchor' and make it the initial value to avoid switching all absolutely positioned boxes that have an implicit anchor element to using the scrollable containing block. Note the initial value may change again, as the discussion is still open... (Issue 13067)
- Clarify that 'flip-block', 'flip-inline', and 'flip-start' use the writing mode of the containing block. (Issue 12869, Issue 13076)
- Clarify that 'auto' 'inset' values are treated as zero when finding the inset-modified containing block size for comparing position options in 'position-try-order'. (Issue 12942)
- Add 'clip-path' to the list of clipping effects considered for 'anchors-visible' and clarify the timing of its checks. (Issue 12732)
- Fix error where base styles were accidentally left out of the position options list. (Issue 12890)
- Clarify timing of 'anchors-visible' checks. (Issue 12732)
- Clarify that the 'normal' alignment resolution based on the 'position-area' value affects the used value (which then keys into how overflow is handled per [CSS-ALIGN-3]).
- Fix algorithm error requiring matching tree roots for anchor name matching, since it is sometimes possible to match across shadow tree boundaries. (Issue 12941)
- Clarify that 'auto' 'inset' values are treated as zero when finding the inset-modified containing block size for 'position-try-order'. (Issue 12942)
- Reorganize prose in § 6 Overflow Management and § 2 Determining the Anchor for better readability. (Issue 12818, Issue 11022)
- Improve guidance in § 7 Accessibility Implications and clarify UA requirements. (Issue 10311)

- Improve examples.

See also Previous Changes.

§ Conformance

§ Document conventions

Conformance requirements are expressed with a combination of descriptive assertions and RFC 2119 terminology. The key words “MUST”, “MUST NOT”, “REQUIRED”, “SHALL”, “SHALL NOT”, “SHOULD”, “SHOULD NOT”, “RECOMMENDED”, “MAY”, and “OPTIONAL” in the normative parts of this document are to be interpreted as described in RFC 2119. However, for readability, these words do not appear in all uppercase letters in this specification.

All of the text of this specification is normative except sections explicitly marked as non-normative, examples, and notes. [RFC2119]

Examples in this specification are introduced with the words “for example” or are set apart from the normative text with `class="example"`, like this:

EXAMPLE 17 ¶

This is an example of an informative example.

Informative notes begin with the word “Note” and are set apart from the normative text with `class="note"`, like this:

Note, this is an informative note.

Advisements are normative sections styled to evoke special attention and are set apart from other normative text with `<strong class="advisement">`, like this:

UAs MUST provide an accessible alternative.

▼ TESTS

Tests relating to the content of this specification may be documented in “Tests” blocks like this one. Any such block is non-normative.

§ Conformance classes

Conformance to this specification is defined for three conformance classes:

style sheet
A CSS style sheet.

renderer
A UA that interprets the semantics of a style sheet and renders documents that use them.

authoring tool
A UA that writes a style sheet.

A style sheet is conformant to this specification if all of its statements that use syntax defined in this module are valid according to the generic CSS grammar and the individual grammars of each feature defined in this module.

A renderer is conformant to this specification if, in addition to interpreting the style sheet as defined by the appropriate specifications, it supports all the features defined by this specification by parsing them correctly and rendering the document accordingly. However, the inability of a UA to correctly render a document due to limitations of the device does not make the UA non-conformant. (For example, a UA is not required to render color on a monochrome monitor.)

An authoring tool is conformant to this specification if it writes style sheets that are syntactically correct according to the generic CSS grammar and the individual grammars of each feature in this module, and meet all other conformance requirements of style sheets as described in this module.

§ Partial implementations

So that authors can exploit the forward-compatible parsing rules to assign fallback values, CSS renderers **must** treat as invalid (and ignore as appropriate) any at-rules, properties, property values, keywords, and other syntactic constructs for which they have no usable level of support. In particular, user agents **must not** selectively ignore unsupported component values and honor supported values in a single multi-value property declaration: if any value is considered invalid (as unsupported values

must be), CSS requires that the entire declaration be ignored.

§ Implementations of Unstable and Proprietary Features

To avoid clashes with future stable CSS features, the CSSWG recommends following best practices for the implementation of unstable features and proprietary extensions to CSS.

§ Non-experimental implementations

Once a specification reaches the Candidate Recommendation stage, non-experimental implementations are possible, and implementors should release an unprefixed implementation of any CR-level feature they can demonstrate to be correctly implemented according to spec.

To establish and maintain the interoperability of CSS across implementations, the CSS Working Group requests that non-experimental CSS renderers submit an implementation report (and, if necessary, the testcases used for that implementation report) to the W3C before releasing an unprefixed implementation of any CSS features. Testcases submitted to W3C are subject to review and correction by the CSS Working Group.

Further information on submitting testcases and implementation reports can be found from on the CSS Working Group's website at http://www.w3.org/Style/CSS/Test/. Questions should be directed to the public-css-testsuite@w3.org mailing list.

§ Index

§ Terms defined by this specification

acceptable anchor element, in § 2.3

accepted @position-try properties, in § 6.4

align-items, in § 4.2

align-self

- (property), in § 4.2
- attribute for CSSPositionTryDescriptors, in § 8.1

alignSelf, in § 8.1

all, in § 2.2

always, in § 6.6

anchor, in § 2

anchor(), in § 3.2

anchor box, in § 2

anchor-center, in § 4.2

anchor element, in § 2.1

anchor functions, in § 1

<anchor-name>
(type), in § 3.2
value for position-anchor, in § 2.4
anchor name, in § 2.1
anchor-name, in § 2.1
Anchor positioning, in § 1
anchor recalculation point, in § 3.3
anchor reference, in § 1
anchor-scope, in § 2.2
<anchor-side>, in § 3.2
<anchor-size>, in § 5.1
anchor-size(), in § 5
anchor specifier, in § 2.3
anchors-valid, in § 6.6
anchors-visible, in § 6.6
apply a position option, in § 6.5.2
auto, in § 2.4
base style, in § 6.1
block, in § 5.1
block-end, in § 3.1.2
block-size, in § 8.1
blockSize, in § 8.1
block-start, in § 3.1.2
bottom
attribute for CSSPositionTryDescriptors, in § 8.1
value for anchor(), in § 3.2
value for position-area, <position-area>, in § 3.1.2
center
value for anchor(), in § 3.2
value for position-area, <position-area>, in § 3.1.2
clipped by intervening boxes, in § 6.6
compensate for scroll, in § 3.3
CSSPositionTryDescriptors, in § 8.1
CSSPositionTryRule, in § 8.1
<dashed-ident>
value for anchor(), in § 3.2
value for anchor-scope, in § 2.2
value for position-try-fallbacks, in § 6.1
<dashed-ident>#, in § 2.1
<dashed-ident> || <try-tactic>, in § 6.1
default anchor box, in § 2.4
default anchor element, in § 2.4
default scroll shift, in § 3.3
determine position fallback styles, in § 6.5
end
value for anchor(), in § 3.2
value for position-area, <position-area>, in § 3.1.2
execute a try-tactic, in § 6.5.2
fallback base styles, in § 6.1
fallback-sensitive changes, in § 6.5.1
flip-block, in § 6.1
flip-inline, in § 6.1
flip-start, in § 6.1
flip-x, in § 6.1
flip-y, in § 6.1

height
attribute for CSSPositionTryDescriptors, in § 8.1
value for anchor-size(), in § 5.1
implicit anchor element, in § 2.4.1
inline, in § 5.1
inline-end, in § 3.1.2
inline-size, in § 8.1
inlineSize, in § 8.1
inline-start, in § 3.1.2
inset, in § 8.1
inset-block, in § 8.1
insetBlock, in § 8.1
inset-block-end, in § 8.1
insetBlockEnd, in § 8.1
inset-block-start, in § 8.1
insetBlockStart, in § 8.1
inset-inline, in § 8.1
insetInline, in § 8.1
inset-inline-end, in § 8.1
insetInlineEnd, in § 8.1
inset-inline-start, in § 8.1
insetInlineStart, in § 8.1
inside, in § 3.2
interleave, in § 9
justify-items, in § 4.2
justify-self
(property), in § 4.2
attribute for CSSPositionTryDescriptors, in § 8.1
justifySelf, in § 8.1
last successful position option, in § 6.5.1.1
left
attribute for CSSPositionTryDescriptors, in § 8.1
value for anchor(), in § 3.2
value for position-area, <position-area>, in § 3.1.2
margin, in § 8.1
margin-block, in § 8.1
marginBlock, in § 8.1
margin-block-end, in § 8.1
marginBlockEnd, in § 8.1
margin-block-start, in § 8.1
marginBlockStart, in § 8.1
margin-bottom, in § 8.1
marginBottom, in § 8.1
margin-inline, in § 8.1
marginInline, in § 8.1
margin-inline-end, in § 8.1
marginInlineEnd, in § 8.1
margin-inline-start, in § 8.1
marginInlineStart, in § 8.1
margin-left, in § 8.1
marginLeft, in § 8.1
margin-right, in § 8.1
marginRight, in § 8.1
margin-top, in § 8.1
marginTop, in § 8.1
max-block-size, in § 8.1
maxBlockSize, in § 8.1
max-height, in § 8.1
maxHeight, in § 8.1

max-inline-size, in § 8.1

maxInlineSize, in § 8.1

max-width, in § 8.1

maxWidth, in § 8.1

min-block-size, in § 8.1

minBlockSize, in § 8.1

min-height, in § 8.1

minHeight, in § 8.1

min-inline-size, in § 8.1

minInlineSize, in § 8.1

min-width, in § 8.1

minWidth, in § 8.1

most-block-size, in § 6.2

most-height, in § 6.2

most-inline-size, in § 6.2

most-width, in § 6.2

name, in § 8.1

none

value for anchor-name, in § 2.1

value for anchor-scope, in § 2.2

value for position-anchor, in § 2.4

value for position-area, in § 3.1

value for position-try-fallbacks, in § 6.1

no-overflow, in § 6.6

normal, in § 6.2

outside, in § 3.2

<percentage>, in § 3.2

place-self, in § 8.1

placeSelf, in § 8.1

position-anchor

(property), in § 2.4

attribute for CSSPositionTryDescriptors, in § 8.1

positionAnchor, in § 8.1

<position-area>

(type), in § 3.1.2

value for position-area, in § 3.1

value for position-try-fallbacks, in § 6.1

position-area

(property), in § 3.1

attribute for CSSPositionTryDescriptors, in § 8.1

positionArea, in § 8.1

position-area grid, in § 3.1.1

Position Fallback Origin, in § 6.4

position option, in § 6.4

position options list, in § 6.1

@position-try, in § 6.4

position-try, in § 6.3

position-try-fallbacks, in § 6.1

position-try-order, in § 6.2

position-visibility, in § 6.6

record the last successful position option, in § 6.5.1.1

remembered scroll offset, in § 3.3

required anchor reference, in § 6.6

resolvable anchor function, in § 3.2.1

resolvable anchor-size function, in § 5.1.1

right

attribute for CSSPositionTryDescriptors, in § 8.1

value for anchor(), in § 3.2

value for position-area, <position-area>, in § 3.1.2

self-block, in § 5.1

self-block-end, in § 3.1.2

self-block-start, in § 3.1.2

self-end

- value for anchor(), in § 3.2
- value for position-area, <position-area>, in § 3.1.2

self-inline, in § 5.1

self-inline-end, in § 3.1.2

self-inline-start, in § 3.1.2

self-start

- value for anchor(), in § 3.2
- value for position-area, <position-area>, in § 3.1.2

self-x-end, in § 3.1.2

self-x-start, in § 3.1.2

self-y-end, in § 3.1.2

self-y-start, in § 3.1.2

span-all, in § 3.1.2

span-block-end, in § 3.1.2

span-block-start, in § 3.1.2

span-bottom, in § 3.1.2

span-end, in § 3.1.2

span-inline-end, in § 3.1.2

span-inline-start, in § 3.1.2

span-left, in § 3.1.2

span-right, in § 3.1.2

span-self-block-end, in § 3.1.2

span-self-block-start, in § 3.1.2

span-self-end, in § 3.1.2

span-self-inline-end, in § 3.1.2

span-self-inline-start, in § 3.1.2

span-self-start, in § 3.1.2

span-self-x-end, in § 3.1.2

span-self-x-start, in § 3.1.2

span-self-y-end, in § 3.1.2

span-self-y-start, in § 3.1.2

span-start, in § 3.1.2

span-top, in § 3.1.2

span-x-end, in § 3.1.2

span-x-start, in § 3.1.2

span-y-end, in § 3.1.2

span-y-start, in § 3.1.2

start

- value for anchor(), in § 3.2
- value for position-area, <position-area>, in § 3.1.2

style, in § 8.1

style & layout interleave, in § 9

target anchor element, in § 2.3

top

- attribute for CSSPositionTryDescriptors, in § 8.1
- value for anchor(), in § 3.2
- value for position-area, <position-area>, in § 3.1.2

<try-size>, in § 6.2

<try-tactic>

- type for position-try-fallbacks, in § 6.1
- value for position-try-fallbacks, in § 6.1

unresolvable anchor function, in § 3.2.1

unresolvable anchor-size function, in § 5.1.1

width

attribute for CSSPositionTryDescriptors, in § 8.1

value for anchor-size(), in § 5.1

x-end, in § 3.1.2

x-start, in § 3.1.2

y-end, in § 3.1.2

y-start, in § 3.1.2

§ Terms defined by reference

[ARIA-1.3] defines the following terms:

role

[CSS-ALIGN-3] defines the following terms:

<baseline-position>

<self-position>

center

end

normal

self-alignment

self-alignment properties

[CSS-BOX-4] defines the following terms:

border box

margin box

margin edge

margin properties

margin-bottom

margin-left

margin-right

margin-top

[CSS-BREAK-4] defines the following terms:

box fragment

fragment

fragmentation context

[CSS-CASCADE-5] defines the following terms:

Animation Origin

author origin

computed value

longhand property

property

reset-only sub-property

revert

revert-layer

used value

user origin

[CSS-CASCADE-6] defines the following terms:

cascade

cascade origin

[CSS-CONDITIONAL-5] defines the following terms:

container query

container query length

[CSS-CONTAIN-2] defines the following terms:

- containment
- content-visibility
- layout containment
- paint containment
- relevant to the user
- skipped contents
- skipping its contents
- style containment
- update content relevancy for a document

[CSS-DISPLAY-3] defines the following terms:

- element
- invisible

[CSS-DISPLAY-4] defines the following terms:

- containing block
- display
- force-hidden
- principal box
- visibility

[CSS-ENV-1] defines the following terms:

- env()
- environment variable
- safe-area-inset-top

[CSS-LOGICAL-1] defines the following terms:

- inset properties
- margin-block-end
- margin-block-start
- margin-inline-start

[CSS-MASKING-1] defines the following terms:

- clip-path

[CSS-OVERFLOW-3] defines the following terms:

- initial scroll position
- ink overflow rectangle
- overflow
- scroll container
- scroll offset
- scrollable overflow area

[CSS-OVERFLOW-4] defines the following terms:

- overflow clip edge

[CSS-POSITION-3] defines the following terms:

- absolute position
- absolute-position containing block
- absolutely position
- absolutely positioned box
- absolutely positioned element
- auto
- bottom
- inset
- inset-block-end
- inset-block-start
- inset-modified containing block
- left
- original containing block
- position
- right
- top

[CSS-POSITION-4] defines the following terms:

- in a lower top layer
- local containing block
- scrollable containing block
- top layer

[CSS-PSEUDO-4] defines the following terms:

- tree-abiding pseudo-elements

[CSS-SCOPING-1] defines the following terms:

- flat tree
- flattened element tree
- loosely matched
- strictly matched
- tree-scoped name
- tree-scoped reference

[CSS-SHADOW-PARTS-1] defines the following terms:

- ::part()

[CSS-SIZING-3] defines the following terms:

- height
- max-height
- max-width
- min-height
- min-width
- sizing property
- width

[CSS-SIZING-4] defines the following terms:

- last remembered size

[CSS-SYNTAX-3] defines the following terms:

- <declaration-list>

[CSS-TRANSFORMS-1] defines the following terms:

- transform

[CSS-TRANSITIONS-1] defines the following terms:

- transitions

[CSS-VALUES-4] defines the following terms:

- #
- &&
- ,
- <dashed-ident>
- <length-percentage>
- <length>
- <percentage>
- ?
- computed length
- CSS-wide keywords
- math function
- {A,B}
- |
- ||

[CSS-VALUES-5] defines the following terms:

- arbitrary substitution function
- invalid at computed-value time

[CSS-VARIABLES-1] defines the following terms:

- substitute a var()

[CSS-VIEWPORT-1] defines the following terms:

- zoom

[CSS-WRITING-MODES-4] defines the following terms:

- block axis
- block-axis
- block-end
- block-start
- end
- flow-relative
- horizontal axis
- horizontal-axis
- horizontal-tb
- inline axis
- inline-axis
- inline-end
- inline-start
- physical
- start
- vertical axis
- vertical-axis
- writing mode

[CSSOM-1] defines the following terms:

- CSSOMString
- CSSRule
- CSSStyleDeclaration
- computed flag
- declarations
- owner node
- parent CSS rule
- readonly flag
- specified order

[DOM] defines the following terms:

- shadow tree

[HTML] defines the following terms:

- input
- li

[INFRA] defines the following terms:

- continue
- for each

[INTERSECTION-OBSERVER] defines the following terms:

- IntersectionObserver

[MOTION-1] defines the following terms:

- offset-path

[RESIZE-OBSERVER-1] defines the following terms:

- ResizeObserver

[SELECTORS-4] defines the following terms:

- originating element
- pseudo-element

[STREAMS] defines the following terms:

- transform

[WEB-ANIMATIONS-1] defines the following terms:

- animation

[WEBIDL] defines the following terms:

- Exposed
- PutForwards
- SameObject

§ References

§ Normative References

[CSS-ALIGN-3]
Elika Etemad; Tab Atkins Jr.. *CSS Box Alignment Module Level 3*. URL: https://drafts.csswg.org/css-align/

[CSS-BOX-4]
Elika Etemad. *CSS Box Model Module Level 4*. URL: https://drafts.csswg.org/css-box-4/

[CSS-BREAK-4]
Rossen Atanassov; Elika Etemad. *CSS Fragmentation Module Level 4*. URL: https://drafts.csswg.org/css-break-4/

[CSS-CASCADE-5]
Elika Etemad; Miriam Suzanne; Tab Atkins Jr.. *CSS Cascading and Inheritance Level 5*. URL: https://drafts.csswg.org/css-cascade-5/

[CSS-CASCADE-6]
Elika Etemad; Miriam Suzanne; Tab Atkins Jr.. *CSS Cascading and Inheritance Level 6*. URL: https://drafts.csswg.org/css-cascade-6/

[CSS-CONTAIN-2]
Tab Atkins Jr.; Florian Rivoal; Vladimir Levin. *CSS Containment Module Level 2*. URL: https://drafts.csswg.org/css-contain-2/

[CSS-DISPLAY-3]
Elika Etemad; Tab Atkins Jr.. *CSS Display Module Level 3*. URL: https://drafts.csswg.org/css-display/

[CSS-DISPLAY-4]
Elika Etemad; Tab Atkins Jr.. *CSS Display Module Level 4*. URL: https://drafts.csswg.org/css-display-4/

[CSS-ENV-1]
CSS Environment Variables Module Level 1. URL: https://drafts.csswg.org/css-env-1/

[CSS-LOGICAL-1]
Elika Etemad; Rossen Atanassov. *CSS Logical Properties and Values Module Level 1*. URL: https://drafts.csswg.org/css-logical-1/

[CSS-MASKING-1]
Dirk Schulze; Brian Birtles; Tab Atkins Jr.. *CSS Masking Module Level 1*. URL: https://drafts.csswg.org/css-masking-1/

[CSS-OVERFLOW-3]
Elika Etemad; Florian Rivoal. *CSS Overflow Module Level 3*. URL: https://drafts.csswg.org/css-overflow-3/

[CSS-OVERFLOW-4]
David Baron; Florian Rivoal; Elika Etemad. *CSS Overflow Module Level 4*. URL: https://drafts.csswg.org/css-overflow-4/

[CSS-POSITION-3]
Elika Etemad; Tab Atkins Jr.. *CSS Positioned Layout Module Level 3*. URL: https://drafts.csswg.org/css-position-3/

[CSS-POSITION-4]
Elika Etemad; Tab Atkins Jr.. *CSS Positioned Layout Module Level 4*. URL: https://drafts.csswg.org/css-position-4/

[CSS-PSEUDO-4]
Elika Etemad; Alan Stearns. *CSS Pseudo-Elements Module Level 4*. URL: https://drafts.csswg.org/css-pseudo-4/

[CSS-SCOPING-1]
Tab Atkins Jr.; Elika Etemad. *CSS Scoping Module Level 1*. URL: https://drafts.csswg.org/css-scoping/

[CSS-SIZING-3]
Tab Atkins Jr.; Elika Etemad. *CSS Box Sizing Module Level 3*. URL: https://drafts.csswg.org/css-sizing-3/

[CSS-SYNTAX-3]
Tab Atkins Jr.; Simon Sapin. *CSS Syntax Module Level 3*. URL: https://drafts.csswg.org/css-syntax/

[CSS-TRANSFORMS-1]
Simon Fraser; et al. *CSS Transforms Module Level 1*. URL: https://drafts.csswg.org/css-transforms/

[CSS-TRANSITIONS-1]
David Baron; et al. *CSS Transitions*. URL: https://drafts.csswg.org/css-transitions/

[CSS-VALUES-3]
Tab Atkins Jr.; Elika Etemad. *CSS Values and Units Module Level 3*. URL: https://drafts.csswg.org/css-values-3/

[CSS-VALUES-4]
Tab Atkins Jr.; Elika Etemad. *CSS Values and Units Module Level 4*. URL: https://drafts.csswg.org/css-values-4/

[CSS-VALUES-5]
Tab Atkins Jr.; Elika Etemad; Miriam Suzanne. *CSS Values and Units Module Level 5*. URL: https://drafts.csswg.org/css-values-5/

[CSS-VARIABLES-1]
Tab Atkins Jr.. *CSS Custom Properties for Cascading Variables Module Level 1*. URL: https://drafts.csswg.org/css-variables/

[CSS-VIEWPORT-1]
Florian Rivoal; Emilio Cobos Álvarez. *CSS Viewport Module Level 1*. URL: https://

drafts.csswg.org/css-viewport/

[CSS-WRITING-MODES-4]

Elika Etemad; Koji Ishii. *CSS Writing Modes Level 4*. URL: https://drafts.csswg.org/css-writing-modes-4/

[CSS2]

Bert Bos; et al. *Cascading Style Sheets Level 2 Revision 1 (CSS 2.1) Specification*. URL: https://drafts.csswg.org/css2/

[CSSOM-1]

Daniel Glazman; Emilio Cobos Álvarez. *CSS Object Model (CSSOM)*. URL: https://drafts.csswg.org/cssom/

[HTML]

Anne van Kesteren; et al. *HTML Standard*. Living Standard. URL: https://html.spec.whatwg.org/multipage/

[INFRA]

Anne van Kesteren; Domenic Denicola. *Infra Standard*. Living Standard. URL: https://infra.spec.whatwg.org/

[INTERSECTION-OBSERVER]

Stefan Zager; Emilio Cobos Álvarez; Traian Captan. *Intersection Observer*. URL: https://w3c.github.io/IntersectionObserver/

[MOTION-1]

Tab Atkins Jr.; Dirk Schulze; Jihye Hong. *Motion Path Module Level 1*. URL: https://drafts.csswg.org/motion-1/

[RESIZE-OBSERVER-1]

Aleks Totic; Greg Whitworth. *Resize Observer*. URL: https://drafts.csswg.org/resize-observer/

[RFC2119]

S. Bradner. *Key words for use in RFCs to Indicate Requirement Levels*. March 1997. Best Current Practice. URL: https://datatracker.ietf.org/doc/html/rfc2119

[SELECTORS-4]

Elika Etemad; Tab Atkins Jr.. *Selectors Level 4*. URL: https://drafts.csswg.org/selectors/

[STREAMS]

Adam Rice; et al. *Streams Standard*. Living Standard. URL: https://streams.spec.whatwg.org/

[WEB-ANIMATIONS-1]

Brian Birtles; et al. *Web Animations*. URL: https://drafts.csswg.org/web-animations-1/

[WEBIDL]

Edgar Chen; Timothy Gu. *Web IDL Standard*. Living Standard. URL: https://webidl.spec.whatwg.org/

§ Informative References

[CSS-ANIMATIONS-1]
David Baron; et al. *CSS Animations Level 1*. URL: https://drafts.csswg.org/css-animations/

[CSS-CONDITIONAL-5]
Chris Lilley; et al. *CSS Conditional Rules Module Level 5*. URL: https://drafts.csswg.org/css-conditional-5/

[CSS-SHADOW-PARTS-1]
Tab Atkins Jr.; Fergal Daly. *CSS Shadow Parts Module Level 1*. URL: https://drafts.csswg.org/css-shadow-parts/

[CSS-SIZING-4]
Tab Atkins Jr.; Elika Etemad; Jen Simmons. *CSS Box Sizing Module Level 4*. URL: https://drafts.csswg.org/css-sizing-4/

[DOM]
Anne van Kesteren. *DOM Standard*. Living Standard. URL: https://dom.spec.whatwg.org/

§ Property Index

Name	Value	Initial	Applies to	Inh.	%ages	Anim-ation type	Canonical order	Com-puted value	New values
'align-items'									anchor-center
'align-self'									anchor-center
'anchor-name'	none | <dashed-ident>#	none	all elements that generate a principal box	no	n/a	discrete	per grammar	as specified	
'anchor-scope'	none | all | <dashed-ident>#	none	all elements	no	n/a	discrete	per grammar	as specified	
'justify-items'									anchor-center
'justify-self'									anchor-center

Name	Value	Initial	Applies to	Inh.	%ages	Anim-ation type	Canonical order	Com-puted value	New values
'position-anchor'	none | auto | <anchor-name>	none	absolutely positioned boxes	no	n/a	discrete	per grammar	as specified	
'position-area'	none | <position-area>	none	positioned boxes with a default anchor box	no	n/a	TBD	per grammar	the keyword none or a pair of keywords, see	
'position-try'	<'position-try-order'>? <'position-try-fallbacks'>	see individual properties	see individual properties	see individual properties	see individual properties	see individual properties	per grammar	see individual properties	
'position-try-fallbacks'	none | [[<dashed-ident> || <try-tactic>] | <position-area>]#	none	absolutely positioned boxes	no	n/a	discrete	per grammar	as specified	
'position-try-order'	normal | <try-size>	normal	absolutely positioned boxes	no	n/a	discrete	per grammar	as specified	
'position-visibility'	always | [anchors-valid || anchors-visible || no-overflow]	anchors-visible	absolutely positioned boxes	no	n/a	discrete	per grammar	as specified	

§ IDL Index

```
[Exposed=Window]
interface CSSPositionTryRule : CSSRule {
  readonly attribute CSSOMString name;
```

```
  [SameObject, PutForwards=cssText] readonly attribute CSSPositionTryDescriptors
};

[Exposed=Window]
interface CSSPositionTryDescriptors : CSSStyleDeclaration {
  attribute CSSOMString margin;
  attribute CSSOMString marginTop;
  attribute CSSOMString marginRight;
  attribute CSSOMString marginBottom;
  attribute CSSOMString marginLeft;
  attribute CSSOMString marginBlock;
  attribute CSSOMString marginBlockStart;
  attribute CSSOMString marginBlockEnd;
  attribute CSSOMString marginInline;
  attribute CSSOMString marginInlineStart;
  attribute CSSOMString marginInlineEnd;
  attribute CSSOMString margin-top;
  attribute CSSOMString margin-right;
  attribute CSSOMString margin-bottom;
  attribute CSSOMString margin-left;
  attribute CSSOMString margin-block;
  attribute CSSOMString margin-block-start;
  attribute CSSOMString margin-block-end;
  attribute CSSOMString margin-inline;
  attribute CSSOMString margin-inline-start;
  attribute CSSOMString margin-inline-end;
  attribute CSSOMString inset;
  attribute CSSOMString insetBlock;
  attribute CSSOMString insetBlockStart;
  attribute CSSOMString insetBlockEnd;
  attribute CSSOMString insetInline;
  attribute CSSOMString insetInlineStart;
  attribute CSSOMString insetInlineEnd;
  attribute CSSOMString top;
  attribute CSSOMString left;
  attribute CSSOMString right;
  attribute CSSOMString bottom;
  attribute CSSOMString inset-block;
  attribute CSSOMString inset-block-start;
  attribute CSSOMString inset-block-end;
  attribute CSSOMString inset-inline;
  attribute CSSOMString inset-inline-start;
  attribute CSSOMString inset-inline-end;
```

```
  attribute CSSOMString width;
  attribute CSSOMString minWidth;
  attribute CSSOMString maxWidth;
  attribute CSSOMString height;
  attribute CSSOMString minHeight;
  attribute CSSOMString maxHeight;
  attribute CSSOMString blockSize;
  attribute CSSOMString minBlockSize;
  attribute CSSOMString maxBlockSize;
  attribute CSSOMString inlineSize;
  attribute CSSOMString minInlineSize;
  attribute CSSOMString maxInlineSize;
  attribute CSSOMString min-width;
  attribute CSSOMString max-width;
  attribute CSSOMString min-height;
  attribute CSSOMString max-height;
  attribute CSSOMString block-size;
  attribute CSSOMString min-block-size;
  attribute CSSOMString max-block-size;
  attribute CSSOMString inline-size;
  attribute CSSOMString min-inline-size;
  attribute CSSOMString max-inline-size;
  attribute CSSOMString placeSelf;
  attribute CSSOMString alignSelf;
  attribute CSSOMString justifySelf;
  attribute CSSOMString place-self;
  attribute CSSOMString align-self;
  attribute CSSOMString justify-self;
  attribute CSSOMString positionAnchor;
  attribute CSSOMString position-anchor;
  attribute CSSOMString positionArea;
  attribute CSSOMString position-area;
};
```

§ Issues Index

ISSUE 1 We might want to change the initial value to be slightly more magical, auto-choosing between ‘none’ and ‘auto’ based on ‘position-area’ being used or not. [Issue #13067]

ISSUE 2 Add a better example; this one can be accomplished easily with 'anchor-center'. [Issue #10776]

ISSUE 3 Transforms have the same issue as scrolling, so Anchor Positioning similarly doesn't pay attention to them normally. Can we go ahead and incorporate the effects of transforms here?

ISSUE 4 Define the precise timing of the snapshot: updated each frame, before style recalc.

ISSUE 5 Similar to remembered scroll offset, can we pay attention to transforms on the default anchor element?

ISSUE 6 Add a picture!

ISSUE 7 The following sections attempt to clarify the interaction with transitions and animations. [Issue #13048]

ISSUE 8 What is a required anchor reference? 'anchor()' functions that don't have a fallback value; the default anchor *sometimes*? Need more detail here.

ISSUE 9 *Any* anchors are missing, or *all* anchors are missing? I can see use-cases for either, potentially. Do we want to make a decision here, or make it controllable somehow?

ISSUE 10 Make sure this definition of clipped is consistent with View Transitions, which wants a similar concept.

ISSUE 11 Add a popover example.

ISSUE 12 Suggestions for ways to improve this section, especially author guidance and examples of best practices for common use cases, is welcome. [Issue #10311]

ISSUE 13 This is not the correct spec for this concept, it should probably go in Cascade, but I need a sketch of it to refer to.

ISSUE 14 Obviously this needs way more details filled in, but for now "act like you already do for container queries" suffices. That behavior is also undefined, but at least it's interoperable (to some extent?).