NSRulerView

Inherits From:	NSView : NSResponder : NSObject
Conforms To:	NSCoding (NSResponder) NSObject (NSObject)
Declared In:	AppKit/NSRulerView.h

Purpose
An NSRulerView displays a ruler and markers above or to the side of an NSScrollView's document view. Views within the NSScrollView can become clients of the ruler view, having it display markers for their elements, and receiving messages from the ruler view when the user manipulates the markers.

Principal Attributes
- Displays markers that represent elements of the client view
- Displays in arbitrary units
- Provides for an accessory view containing extra controls

Creation

– setHasHorizontalRuler: (NSScrollView)	
– setHasVerticalRuler: (NSScrollView)	
– initWithScrollView:orientation:	Designated initializer.

Commonly Used Methods

– setClientView:	Changes the ruler's client view.
– setMarkers:	Sets the markers displayed by the ruler view.
– setAccessoryView:	Sets the accessory view.
– trackMarker:withMouseEvent:	Allows the user to add a new marker.

Class Description

An NSRulerView resides in an NSScrollView, displaying a labeled ruler and markers for its client, the document view of the NSScrollView or a subview of the document view. The client view can add and remove markers representing its contents, such as graphic elements, margins, and text tabs. The NSRulerView tracks user manipulation of the markers and informs the client view of those actions. NSRulerView handles both horizontal and vertical rulers, which are tiled in the scroll view above and to the side of the content view, respectively. NSRulerViews are sometimes called simply *ruler views* or even *rulers*.

A ruler view comprises three regions. First is the *ruler area*, where the ruler's baseline, hash marks, and labels are drawn. The ruler area is largely static, but it scales its hash marks to document view's coordinate system, and can display the hash marks in arbitrary units. The second region is the *marker area*, where ruler markers (NSRulerMarker objects) representing elements of the client view are displayed. This is a more dynamic area, changing with the selection and state of the client view. The final region is the *accessory view area*, which is by default not present but appears when you add an accessory view to the ruler view. An accessory view can contain additional controls for manipulating the ruler's client view, such as alignment buttons or a set of markers that can be dragged onto the ruler.

A ruler view interacts with its client view in several ways. On appropriating the ruler view, the client view usually sets it up according to its needs. The client view can also dynamically update the ruler view's markers as its layout changes. In its turn, the ruler view informs the client view of actions the user takes on the ruler markers, allowing the client view to approve or limit the actions and to update its state based on the results of the actions.

The appearance of a ruler is based on both the document view and the client view. The document view, as the backdrop within the scroll view, serves as the canvas on which any client views are laid. Because of the document view's anchoring role, a ruler always draws its hash marks and labels relative to the document view's coordinate system. A vertical ruler also checks whether the document view is flipped and acts accordingly. However, the ruler view treats subviews of the document view as items laid out within the coordinate system defined by the document view, and so doesn't change its hash marks when a client view other than the document view is moved or scaled. For the client view's convenience it does, however, express marker locations in the client view's coordinate system. A few other operations that ruler views perform are defined in terms of the ruler's own coordinate system. The discussion of a feature or method makes clear which coordinate system applies. For reference, this table summarizes all of the coordinate systems involved in using ruler views, and the operations based on them:

Coordinate System	**Used for**
Client view	Marker locations
Document view	Calculating hash marks based on measurement units and scaling, origin offset for zero marks
Ruler view	Temporary rulerlines, component layout
Marker image	Image origin (which locates the image relative to the marker location)

Measurement Units

A new ruler view automatically uses the user's preferred measurement units for drawing hash marks and labels, as stored in the user defaults system under the key "NSMeasurementUnit". If your application allows the user to change his preferred measurement units, you can change them at run time using

setMeasurementUnits:, which takes the name of the units to use, such as “Inches” or “Centimeters”, and causes the ruler view to use that unit definition in spacing its hash marks and labels.

NSRulerView supports the units Inches, Centimeters, Points, and Picas by default. If your application uses other measurement units, your application should define and register them before creating any ruler views. To do, use the class method **registerUnitWithName:abbreviation:unitsToPointsConversionFactor: stepUpCycle:stepDownCycle:**. Your application can register these wherever it’s most convenient, such as in the NSApplication delegate method **applicationDidFinishLaunching:**. This code fragment registers a new unit called Grummets, with the abbreviation gt:

```
NSArray *upArray;
NSArray *downArray;

upArray = [NSArray arrayWithObjects:[NSNumber numberWithFloat:2.0], nil];
downArray = [NSArray arrayWithObjects:[NSNumber numberWithFloat:0.5],
    [NSNumber numberWithFloat:0.2], nil];
[NSRulerView registerUnitWithName:@"Grummets"
    abbreviation:NSLocalizedString(@"gt", @"Grummets abbreviation string")
    unitToPointsConversionFactor:100.0
    stepUpCycle:upArray stepDownCycle:downArray];
```

A Grummet is 100.0 PostScript units (points) in length, so a ruler view using it draws a major hash mark every 100.0 points when its document view is unscaled. If the document view is scaled, the ruler view spaces its hash marks accordingly.

The step-up and step-down cycles control how hash marks are drawn for fractions and multiples of units. NSRulerView attempts to place hash marks so that they’re neither too crowded nor too sparse based on the current scale of the document view. It does so by drawing smaller hash marks for fractions of units where possible, and by removing hash marks for whole units where necessary.

The step-down cycle determines the fractional units checked by the ruler view. For example, with the unit Grummets defined above, the step down cycle is 0.5, then 0.2. With this cycle, the ruler view first checks to see if there’s room for marks every half Grummet, placing them if there is. Then, it checks every fifth of the remaining space, or a tenth of a full Grummet, placing further hash marks there if there’s room. Then it returns to the first step in the cycle to further subdivide the ruler, and so on.

The step-up cycle determines how many full unit marks get dropped when there isn’t room for each one. The example uses a single-step cycle of 2.0, which means that each second Grommet’s hash mark is displayed if there isn’t room for every one, then every fourth if there still isn’t room, and so on.

Preparing a Ruler View for Use

Adding a ruler view to a scroll view can be as simple as invoking NSScrollView’s **setHasHorizontalRuler:** and **setHasVerticalRuler:** methods. These create instances of the default ruler view class, which you can change using the NSScrollView class method **setRulerViewClass:**. You can also set ruler views directly on a per-instance basis using **setHorizontalRulerView:** and

setVerticalRulerView:. Once you've added rulers to a scroll view, you can hide and reveal them using **setRulersVisible:**.

Beyond creating the rulers, you need take only a few steps to set them up properly for use by the views contained within the scroll view: locating the zero marks of the rulers, and reserving room for accessory views. You normally perform these steps only once, when setting up the NSScrollView with rulers. However, if you allow the user to reset document attributes such as margins, you should change the zero mark locations as well. Also, if you reuse the scroll view by swapping in a new document view you may need to set up the rulers again with different settings.

The first step is to determine where you want the zero marks of the rulers to be located relative to the bounds origin of the document view. The zero marks are coincident with the bounds origin by default, but you can change this with the method **setOriginOffset:**. This method takes an offset specified in the document view's coordinate system. If you need to set the origin offset based on a point in a subview of the document view, such as a text view that's inset on a page, use **convertPoint:fromView:** to realize it in the document view's coordinate system. This code fragment places the zero marks at the bounds origin of a client view, which lies somewhere inside the document view:

```
zero = [docView convertPoint:[clientView bounds].origin fromView:clientView];
[horizRuler setOriginOffset:zero.x - [docView bounds].origin.x];
```

After placing the zero marks, you should set up your rulers so that they don't change in size as the user works within the document view. For example, if two different subviews of the document view use different accessory views, the ruler view enlarges itself as necessary each time you change the accessory view. Such changes are at best unsightly and at worst confusing to the user. To avoid this problem, calculate ahead of time the sizes of the largest accessory view and the largest markers, and set the ruler view's required thickness for these elements using **setReservedThicknessForAccessoryView:** and **setReservedThicknessForMarkers:**. For example, if you have two accessory views for the horizontal ruler, one 16.0 PostScript units high and the other 24.0, invoke **setReservedThicknessForAccessoryView:** with an argument of 24.0.

Changing the Client

Once the ruler view is fully set up, the scroll view's document view, or any subview of the document view, can become its client by sending it a **setClientView:** message. This method notifies the prior client that it's losing the ruler view using the **rulerView:willSetClientView:** method, removes all of the ruler view's markers, and sets the new client view. A client view normally appropriates the ruler when it becomes first responder and keeps it until some other view appropriates it. After appropriating the ruler view, the client needs to set up its layout and markers.

Adjusting the Layout

If the client has a custom accessory view, it sets that using **setAccessoryView:**. Clients without accessory views should avoid removing the ruler view's accessory view when appropriating the ruler, as this can cause unsightly screen flicker as the ruler is redrawn. It's better in this case for a client view that has an accessory view to implement **rulerView:willSetClientView:**, disabling the controls in the accessory view so that they're not active when other clients are using the ruler. Then, when the client view with the accessory view appropriates the ruler, it should set its accessory view again in case another client swapped the accessory view out, and reenable the controls.

Setting Ruler Markers

Aside from the layout of the ruler view itself, the client can also add markers to indicate the positions of its graphic elements, such as tabs and margins in text or the bounding boxes of drawn shapes or images. Each marker is an NSRulerMarker object, which displays a graphic image on the ruler at its given location, and can be associated with an object that identifies the attribute indicated by the marker. You initialize an NSRulerMarker using its **initWithRulerView:markerLocation:image:imageOrigin:** method, which takes as arguments the NSRulerView where the marker will be displayed, its location on the ruler in the client view's coordinate system, the image to display, and the point within the image that lies on the ruler's baseline. Once you've created the markers, you can use NSRulerView's **addMarker:** or **setMarkers:** methods to put them on the ruler. This code fragment, for example, sets up markers denoting the left and right edges of the selected object's frame rectangle:

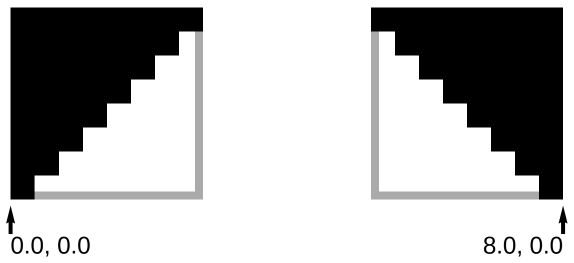
```
NSRulerMarker *leftMarker;
NSRulerMarker *rightMarker;

leftMarker = [[NSRulerMarker alloc] initWithRulerView:horizRuler
    markerLocation:NSMinX([selectedItem frame]) image:leftImage
    imageOrigin:NSMakePoint(0.0, 0.0)];

rightMarker = [[NSRulerMarker alloc] initWithRulerView:horizRuler
    markerLocation:NSMaxX([selectedItem frame]) image:rightImage
    imageOrigin:NSMakePoint(8.0, 0.0)];

[horizRuler setMarkers:[NSArray arrayWithObjects:leftMarker, rightMarker, nil]];
```

The images used for this example are 8 pixels square, and lie just inside of their relevant positions. The figure below shows the left and right marker images, enlarged and with gray bounding boxes. Thus, the left marker's image must be placed with its lower left corner, or (0.0, 0.0), at the marker location, while the lower right corner of the right marker, at (8.0, 0.0), is used. The image origin is always expressed in the coordinate system of the image itself, just as an NSCursor's hot spot is.



A new NSRulerMarker allows the user to drag it around on its ruler, but not to remove it. You can change these defaults by sending it **setMovable:** and **setRemovable:** messages. For example, you might make markers representing tabs in text removable to allow the user to edit the paragraph settings.

Markers bear one additional attribute, which allows you to distinguish among multiple markers, specifically markers that share the same image. This is the *represented object*, set with NSRulerMarker's **setRepresentedObject:** method. A represented object can simply be a string identifying a generic attribute, such as "Left Margin" or "Right Margin". It can also be an object stored in the client view or in the selection; for example, the OPENSTEP text system records tab stops as NSTextTab objects, which include the tab location and its alignment. When the user manipulates a tab marker, the client can simply retrieve its represented object to get the tab being affected.

Updating the Ruler View

A single client view may contain many selectable items, such as graphic shapes or paragraphs of text with different ruler settings. When the selection changes, the client must reset the ruler view's markers based on the new selection. This kind of updating is fairly straightforward and can be performed as described above for situations where the client view itself changes.

Another kind of updating is needed when you wish to support dynamic updating of ruler markers as the user manipulates the elements of the client view. For example, when the user moves a shape, you want the ruler markers to relocate when the user finishes moving it. Any method that changes relevant attributes of the selection should update the ruler markers, whether by replacing them wholesale or by checking each one present and updating its location.

You can even put such updating code within a modal loop that handles dragging items around in the client view, so that the markers track the position of the selected item. This can be a fairly heavyweight operation to perform while also handling movement of the selected item, however. In support of a lighter weight means of showing this information, NSRulerView allows you to draw temporary *rulerlines* that can be drawn and erased very quickly. One method, **moveRulerlineFromLocation:toLocation:**, controls the drawing of rulerlines. It takes two locations expressed in the NSRulerView's coordinate system, erasing the rulerline at the old location and redrawing it at the new. To create a new rulerline, specify –1.0 as the old location; to erase one completely, specify –1.0 as the new location. Although you're responsible for keeping track of the locations to erase and redraw, this isn't as cumbersome or inefficient as sifting through or replacing the markers themselves.

User Manipulation of Markers

While a ruler's client view must perform the work of determining marker locations and placing them on the ruler, the ruler itself handles all the work of tracking user manipulations of the markers, sending messages to the client view that inform it of the changes before they begin, as they occur, and after they finish. The client view can use these messages to update its own state. The following sections describe the individual processes of moving, removing, and adding markers, along with a special method for handling mouse events in the ruler area.

Moving Markers

When the user presses the mouse button over a ruler marker, NSRulerView sends the marker a **trackMouse:adding:** message. If the marker isn't movable this method does nothing and immediately returns NO. If it is movable, then it sends the client a series of messages allowing it to determine how the user can move the marker around on the ruler.

First of these messages is **rulerView:shouldMoveMarker:**, which allows the client view to prevent an otherwise movable marker from being moved. Normally, whether a marker can be moved should be set on the marker itself, but there are situations, such as where items can be locked in place, where this is more properly tracked by the client view instead. If the client view returns YES, allowing the movement, then it receives a series of **rulerView:willMoveMarker:toLocation:** messages as the user drags the marker around. Each message identifies the marker being moved and its proposed new location in the client view's coordinate system. The client view can return an altered location to restrict the marker's movement, or update its display to reflect the new location. Finally, when the user releases the mouse button, the client receives a **rulerView:didMoveMarker:**, on which it can update its state and clean up any information it may have used while tracking the marker's movements.

Removing Markers

Removal of markers is handled by a similar set of messages. However, these are always sent during a movement operation, as the user must first be dragging a marker within the ruler to be able to drag it off the ruler. If a marker isn't set to be removable, the user simply can't drag it off. If the marker is removable, then when the user drags the mouse far enough away from the ruler's baseline, it sends the client view a **rulerView:shouldRemoveMarker:** message, allowing the client to approve or veto the removal. No messages are necessary for new locations, of course, but if the user returns the marker to the ruler then it resumes sending **rulerView:willMoveMarker:toLocation:** messages as before. If the user releases the mouse with the marker dragged away from the ruler, the marker sends the client view a **rulerView:didRemoveMarker:** message, so the user can delete the item or attribute represented by the marker.

Adding Markers

User addition of a marker must be initiated by the application, of course, since there is no marker yet for the ruler to track. The first step in adding a marker, then, is to create one, using NSRulerMarker's **initWithRulerView:markerLocation:image:imageOrigin:** method. Once the new marker is created, you instruct the ruler view to handle dragging it onto itself by sending it a **trackMarker:withMouseEvent:** message. One means of doing this is to use the mouse event from the client view method **rulerView:handleMouseDown:**, as described below under "Handling Mouse Events in the Ruler Area." Another is to create a custom view object—which typically resides in the ruler's accessory view—that displays prototype markers, and that handles a mouse-down event by creating a new marker for the ruler and invoking **trackMarker:withMouseEvent:** with the new marker and that mouse-down event.

Once you've initiated the addition process, things proceed in the same manner as for moving a marker. The ruler view sends the new marker a **trackMouse:adding:** message, with YES as the second argument to indicate that the marker isn't merely being moved. The marker being added then sends the client view a **rulerView:shouldAddMarker:** message, and if the client approves then it repeatedly sends **rulerView:willAddMarker:atLocation:** messages as the user moves the marker around on the ruler. The user can drag the marker away to avoid adding it, or release the mouse button over the ruler, in which case the client receives a **rulerView:didAddMarker:** message.

As with moving a marker, you should consider enabling and disabling in a more immediate fashion than by the client view method if possible. If the user shouldn't be able to drag a marker from the accessory view, for example, the view containing the prototype marker should disable itself and indicate this in its appearance, rather than allowing the user to drag a marker out only to discover that the ruler won't accept it.

Handling Mouse Events in the Ruler Area

In addition to handling user manipulation of markers, a ruler informs its client view when the user presses the mouse button while the mouse is inside the ruler area (where hash marks are drawn), by sending it a **rulerView:handleMouseDown:** message. This allows the client view to take some special action, such as adding a new marker to the ruler, as described above. This approach works well when it's quite clear what kind of marker will be created. The client view can also use this message as a cue to change its display in some way; for example to add or remove a guideline that assists the user in laying out and aligning items in the view.

Method Types

Creating instances	– initWithScrollView:orientation:
Altering measurement units	+ registerUnitWithName:abbreviation: unitToPointsConversionFactor: stepUpCycle:stepDownCycle: – setMeasurementUnits: – measurementUnits
Setting the client view	– setClientView: – clientView
Setting an accessory view	– setAccessoryView: – accessoryView
Setting the zero mark position	– setOriginOffset: – originOffset
Adding and removing markers	– setMarkers: – markers – addMarker: – removeMarker: – trackMarker:withMouseEvent:

Drawing temporary rulerlines	– moveRulerlineFromLocation:toLocation:
Drawing	– drawHashMarksAndLabelsInRect: – drawMarkersInRect: – invalidateHashMarks
Ruler layout	– setScrollView: – scrollView – setOrientation: – orientation – setReservedThicknessForAccessoryView: – reservedThicknessForAccessoryView – setReservedThicknessForMarkers: – reservedThicknessForMarkers – setRuleThickness: – ruleThickness – requiredThickness – baselineLocation – isFlipped

Class Methods

registerUnitWithName:abbreviation:unitToPointsConversionFactor: stepUpCycle:stepDownCycle:

+ (void)**registerUnitWithName:**(NSString *)*unitName*
abbreviation:(NSString *)*abbreviation*
unitToPointsConversionFactor:(float)*conversionFactor*
stepUpCycle:(NSArray *)*stepUpCycle*
stepDownCycle:(NSArray *)*stepDownCycle*

Registers a new unit of measurement with the NSRulerView class, making it available to all instances of NSRulerView. *unitName* is the name of the unit in English, in plural form and capitalized by convention; “Inches”, for example. The unit name is used as a key to identify the measurement units, and so shouldn’t be localized. *abbreviation* is a localized short form of the unit name, such as “in” for Inches. *conversionFactor* is the number of PostScript points in the specified unit; there are 72.0 points per inch, for example. *stepUpCycle* and *stepDownCycle* are arrays of NSNumbers that specify how hash marks are calculated, as explained in the class description under “Preparing a Ruler View for Use.” All numbers in *stepUpCycle* should be greater than 1.0, those in *stepDownCycle* should be less than 1.0.

NSRulerView supports these units by default:

Unit Name	**Abbreviation**	**Points/Unit**	**Step-up Cycle**	**Step-down Cycle**
Inches	in	72.0	2.0	0.5
Centimeters	cm	28.35	2.0	0.5, 0.2
Points	pt	1.0	10.0	0.5
Picas	pc	12.0	10.0	0.5

See also: – **setMeasurementUnits:**

Instance Methods

accessoryView

– (NSView *)**accessoryView**

Returns the receiver’s accessory view, if it has one.

See also: – **setAccessoryView:**, – **reservedThicknessForAccessoryView:**

addMarker:

– (void)**addMarker:**(NSRulerMarker *)*aMarker*

Adds *aMarker* to the receiver, without consulting the client view for approval. Raises NSInternalInconsistencyException if the receiver has no client view.

See also: – **setMarkers:**, – **removeMarker:**, – **markers**, – **trackMarker:withMouseEvent:**

baselineLocation

– (float)**baselineLocation**

Returns the location of the receiver’s baseline, in its own coordinate system. This is a *y* position for horizontal rulers and an *x* position for vertical ones.

See also: – **ruleThickness**

clientView

– (NSView *)**clientView**

Returns the receiver’s client view, if it has one.

See also: – **setClientView:**

drawHashMarksAndLabelsInRect:

– (void)**drawHashMarksAndLabelsInRect:**(NSRect)*aRect*

Draws the receiver's hash marks and labels in *aRect*, which is expressed in the receiver's coordinate system. This method is invoked by **drawRect:**—you should never need to invoke it directly. You can define custom measurement units using the class method **registerUnitWithName:...**. Override this method if you want to customize the appearance of the hash marks themselves.

See also: – **invalidateHashMarks**, – **drawMarkersInRect:**

drawMarkersInRect:

– (void)**drawMarkersInRect:**(NSRect)*aRect*

Draws the receiver's markers in *aRect*, which is expressed in the receiver's coordinate system. This method is invoked by **drawRect:**; you should never need to invoke it directly, but you might want to override it if you want to do something different when drawing markers.

See also: – **reservedThicknessForMarkers**, – **drawHashMarksAndLabelsInRect:**

initWithScrollView:orientation:

– (id)**initWithScrollView:**(NSScrollView *)*aScrollView* **orientation:**(NSRulerOrientation)*orientation*

Initializes a newly allocated NSRulerView to have *orientation* (NSHorizontalRuler or NSVerticalRuler) within *aScrollView*. The new ruler view displays the user's preferred measurement units, and has no client, markers, or accessory view. Unlike most subclasses of NSView, no initial frame rectangle is given for NSRulerView; its containing NSScrollView adjusts its frame rectangle as needed.

This is the designated initializer for the NSRulerView class. Returns **self**.

invalidateHashMarks

– (void)**invalidateHashMarks**

Forces recalculation of the hash mark spacing for the next time the receiver is displayed. You should never need to invoke this method directly, but might need to override it if you override **drawHashMarksAndLabelsInRect:**.

See also: – **drawHashMarksAndLabelsInRect:**

isFlipped

– (BOOL)**isFlipped**

Returns YES if the NSRulerView's coordinate system is flipped, NO otherwise. A vertical ruler takes into account whether the coordinate system of the NSScrollView's document view—*not* the receiver's client view—is flipped. A horizontal ruler is always flipped.

markers

– (NSArray *)**markers**

Returns the receiver's NSRulerMarkers. The markers aren't guaranteed to be sorted in any particular order.

See also: – **setMarkers:**, – **addMarker:**, – **removeMarker:**, – **markerLocation** (NSRulerMarker)

measurementUnits

– (NSString *)**measurementUnits**

Returns the full name of the measurement units in effect for the receiver.

See also: – **setMeasurementUnits:**, + **registerUnitWithName:...**

moveRulerlineFromLocation:toLocation:

– (void)**moveRulerlineFromLocation:**(float)*oldLoc* **toLocation:**(float)*newLoc*

Draws temporary lines in the ruler area. If *oldLoc* is zero or greater, erases the rulerline at that location; if *newLoc* is zero or greater, draws a new rulerline at that location. *oldLoc* and *newLoc* are expressed in the coordinate system of the NSRulerView, *not* of the client or document view, and are *x* coordinates for horizontal rulers and *y* coordinates for vertical rulers. Use NSView's **convert...** methods to convert coordinates from the client or document view's coordinate system to that of the NSRulerView.

This method is useful for drawing highlight lines in the ruler to show the position or extent of an object while it's being dragged in the client view. The sender is responsible for keeping track of the number and positions of temporary lines—the NSRulerView only does the drawing.

orientation

– (NSRulerOrientation)**orientation**

Returns the orientation of the NSRulerView, either NSHorizontalRuler or NSVerticalRuler.

See also: – **setOrientation:**

originOffset

– (float)**originOffset**

Returns the distance from the receiver's zero hash mark to the bounds origin of the NSScrollView's document view (*not* the receiver's client view), in the document view's coordinate system.

See also: – **setOriginOffset:**

removeMarker:

– (void)**removeMarker:**(NSRulerMarker *)*aMarker*

Removes *aMarker* from the receiver, without consulting the client view for approval.

See also: – **setMarkers:**, – **addMarker:**

requiredThickness

– (float)**requiredThickness**

Returns the thickness needed for proper tiling of the receiver within an NSScrollView. This is the height of a horizontal ruler and the width of a vertical ruler. The required thickness is the sum of the thicknesses of the ruler area, the marker area, and the accessory view.

See also: – **rulerThickness**, – **reservedThicknessForMarkers**, – **reservedThicknessForAccessoryView**

reservedThicknessForAccessoryView

– (float)**reservedThicknessForAccessoryView**

Returns the thickness reserved to contain the receiver's accessory view, its height or width depending on the receiver's orientation. This is automatically enlarged as necessary to the accessory view's thickness (but never automatically reduced). To prevent retiling of a ruler view's scroll view, you should set its maximal thickness upon creating using **setReservedThicknessForAccessoryView:**.

reservedThicknessForMarkers

– (float)**reservedThicknessForMarkers**

Returns the thickness reserved to contain the images of the receiver's ruler markers, the height or width depending on the receiver's orientation. This is automatically enlarged as necessary to accommodate the thickest ruler marker image (but never automatically reduced). To prevent retiling of a ruler view's scroll view, you should set its maximal thickness upon creating using **setReservedThicknessForMarkers:**.

See also: – **thicknessRequiredInRulerView** (NSRulerMarker)

ruleThickness

– (float)**ruleThickness**

Returns the thickness of the receiver's ruler area (the area where hash marks and labels are drawn), its height or width depending on the receiver's orientation.

See also: – **setRuleThickness:**

scrollView

– (NSScrollView *)**scrollView**

Returns the NSScrollView object that contains the receiver.

See also: – **setScrollView:**, – **setHorizontalRuler:** (NSScrollView), – **setVerticalRuler:** (NSScrollView)

setAccessoryView:

– (void)**setAccessoryView:**(NSView *)*aView*

Sets the receiver's accessory view to *aView*. Raises an NSInternalInconsistencyException if *aView* is non-**nil** and the receiver has no client view.

See also: – **accessoryView**, – **reservedThicknessForAccessoryView**

setClientView:

– (void)**setClientView:**(NSView *)*aView*

Sets the receiver's client view to *aView*, without retaining it, and removes its ruler markers, after informing the prior client of the change using **rulerView:willSetClientView:**. *aView* must be either the document view of the NSScrollView that contains the receiver, or a subview of the document view.

See also: – **clientView**

setMarkers:

– (void)**setMarkers:**(NSArray *)*markers*

Sets the receiver's ruler markers to *markers*, removing any existing ruler markers and not consulting with the client view about the new markers. *markers* can be **nil** or empty to remove all ruler markers. Raises an NSInternalInconsistencyException if *markers* is non-**nil** and the receiver has no client view.

See also: – **addMarker:**, – **removeMarker:**

setMeasurementUnits:

– (void)**setMeasurementUnits:**(NSString *)*unitName*

Sets the measurement units used by the ruler to *unitName*. *unitName* must have been registered with the NSRulerView class object prior to invoking this method. See the description of the class method **registerUnitWithName:...** for a list of predefined units.

See also: – **measurementUnits**

setOrientation:

– (void)**setOrientation:**(NSRulerOrientation)*orientation*

Sets the orientation of the receiver to *orientation*, which may be NSHorizontalRuler or NSVerticalRuler. You should never need to invoke this method directly—it's automatically invoked by the containing NSScrollView.

See also: – **orientation**

setOriginOffset:

– (void)**setOriginOffset:**(float)*offset*

Sets the distance to the zero hash mark from the bounds origin of the NSScrollView's document view (*not* of the receiver's client view), in the document view's coordinate system. The default offset is 0.0, meaning that the ruler origin coincides with the bounds origin of the document view.

See also: – **originOffset**

setReservedThicknessForAccessoryView:

– (void)**setReservedThicknessForAccessoryView:**(float)*thickness*

Sets the room available for the NSRulerView's accessory view to *thickness*. If the ruler is horizontal, *thickness* is the height of the accessory view; otherwise, it's the width. NSRulerViews by default reserve no space for an accessory view.

An NSRulerView automatically increases the reserved thickness as necessary to that of the accessory view. When the accessory view is thinner than the reserved space, it's centered in that space. If you plan to use several accessory views of different sizes, you should set the reserved thickness beforehand to that of the thickest accessory view, in order to avoid retiling of the NSScrollView.

See also: – **reservedThicknessForAccessoryView**, – **setAccessoryView:**,
– **setReservedThicknessForMarkers:**

setReservedThicknessForMarkers:

– (void)**setReservedThicknessForMarkers:**(float)*thickness*

Sets the room available for ruler markers to *thickness*. The default thickness reserved for markers is 15.0 PostScript units for a horizontal ruler and 0.0 PostScript units for a vertical ruler (under the assumption that vertical rulers rarely contain markers). If you don't expect to have any markers on the ruler, you can set the reserved thickness to 0.0.

An NSRulerView automatically increases the reserved thickness as necessary to that of its thickest marker. If you plan to use markers of varying sizes, you should set the reserved thickness beforehand to that of the thickest one in order to avoid retiling of the NSScrollView.

See also: – **reservedThicknessForMarkers**, – **setMarkers:**,
– **setReservedThicknessForAccessoryView:**,
– **thicknessRequiredInRulerView** (NSRulerMarker)

setRuleThickness:

– (void)**setRuleThickness:**(float)*thickness*

Sets to *thickness* the thickness of the area where ruler hash marks and labels are drawn. This value is the height of the ruler area for a horizontal ruler or the width of the ruler area for a vertical ruler. Rulers are by default 16.0 PostScript units thick. You should rarely need to change this layout attribute, but subclasses might do so to accommodate custom drawing.

See also: – **ruleThickness**

setScrollView:

– (void)**setScrollView:**(NSScrollView *)*scrollView*

Sets the NSScrollView that owns the receiver to *scrollView*, without retaining it. This method is generally invoked only by the ruler's scroll view; you should rarely need to invoke it directly.

See also: – **scrollView**, – **setHorizontalRuler:** (NSScrollView), – **setVerticalRuler:** (NSScrollView)

trackMarker:withMouseEvent:

– (BOOL)**trackMarker:**(NSRulerMarker *)*aMarker* **withMouseEvent:**(NSEvent *)*theEvent*

Tracks the mouse to add *aMarker* based on the initial mouse-down or mouse-dragged event *theEvent*. Returns YES if the receiver adds *aMarker*, NO if it doesn't. This method works by sending **trackMouse:adding:** to *aMarker* with *theEvent* and YES as arguments.

An application typically invokes this method in one of two cases. In the simpler case, the client view can implement **rulerView:handleMouseDown:** to invoke this method when the user presses the mouse button in the NSRulerView's ruler area. This technique is appropriate when it's clear what kind of marker will be added by clicking in the ruler area. The second, more general, case involves the application providing a palette of different kinds of markers that can be dragged onto the ruler, from the ruler's accessory view or from some other place. With this technique the palette tracks the mouse until it enters the ruler view, at which time it hands over control to the ruler view by invoking **trackMarker:withMouseEvent:**.

See also: – **addMarker:**, – **setMarkers:**

Methods Implemented by the NSRulerView's Client

rulerView:didAddMarker:

– (void)**rulerView:**(NSRulerView *)*aRulerView* **didAddMarker:**(NSRulerMarker *)*aMarker*

Informs the client that *aRulerView* allowed the user to add *aMarker*. The client can take whatever action it needs based on this message, such as adding a new tab stop to the selected paragraph or creating a layout guideline.

See also: – **representedObject** (NSRulerMarker), – **markerLocation** (NSRulerMarker)

rulerView:didMoveMarker:

– (void)**rulerView:**(NSRulerView *)*aRulerView* **didMoveMarker:**(NSRulerMarker *)*aMarker*

Informs the client that *aRulerView* allowed the user to move *aMarker*. The client can take whatever action it needs based on this message, such as updating the location of a tab stop in the selected paragraph, moving a layout guideline, or resizing a graphic element.

See also: – **representedObject** (NSRulerMarker), – **markerLocation** (NSRulerMarker)

rulerView:didRemoveMarker:

– (void)**rulerView:**(NSRulerView *)*aRulerView* **didRemoveMarker:**(NSRulerMarker *)*aMarker*

Informs the client that *aRulerView* allowed the user to remove *aMarker*. The client can take whatever action it needs based on this message, such as deleting a tab stop from the paragraph style or removing a layout guideline.

See also: – **representedObject** (NSRulerMarker)

rulerView:handleMouseDown:

– (void)**rulerView:**(NSRulerView *)*aRulerView* **handleMouseDown:**(NSEvent *)*theEvent*

Informs the client that the user has pressed the mouse button while the cursor is in the ruler area of *aRulerView*. *theEvent* is the mouse-down event that triggered the message. The client view can implement this method to perform an action such as adding a new marker using **trackMarker:withMouseEvent:** or adding layout guidelines.

rulerView:shouldAddMarker:

– (BOOL)**rulerView:**(NSRulerView *)*aRulerView* **shouldAddMarker:**(NSRulerMarker *)*aMarker*

Requests permission for *aRulerView* to add *aMarker*, an NSRulerMarker being dragged onto the ruler by the user. If the client returns YES then the ruler view accepts the new marker and begins tracking its movement; if the client returns NO then the ruler view refuses the new marker.

See also: – **rulerView:willAddMarker:atLocation:**

rulerView:shouldMoveMarker:

– (BOOL)**rulerView:**(NSRulerView *)*aRulerView* **shouldMoveMarker:**(NSRulerMarker *)*aMarker*

Requests permission for *aRulerView* to move *aMarker*. If the client returns YES then the ruler view allows the user to move the marker; if the client returns NO then the marker doesn't move.

The user's ability to move a marker is typically set on the marker itself, using NSRulerMarker's **setMovable:** method. You should use this client view method only when the marker's movability can vary depending on a variable condition (for example, if graphic items can be locked down to prevent them from being inadvertently moved).

See also: – **rulerView:willMoveMarker:toLocation:**

rulerView:shouldRemoveMarker:

– (BOOL)**rulerView:**(NSRulerView *)*aRulerView*
    shouldRemoveMarker:(NSRulerMarker *)*aMarker*

Requests permission for *aRulerView* to remove *aMarker*. If the client returns YES then the ruler view allows the user to remove the marker; if the client returns NO then the marker is kept pinned to the ruler's baseline. This message is sent as many times as needed while the user drags the marker.

The user's ability to remove a marker is typically set on the marker itself, using NSRulerMarker's **setRemovable:** method. You should use this client view method only when the marker's removability can vary while the user drags it (for example, if the user must press the Shift key to remove a marker).

rulerView:willAddMarker:atLocation:

– (float)**rulerView:**(NSRulerView *)*aRulerView*
    willAddMarker:(NSRulerMarker *)*aMarker*
    atLocation:(float)*location*

Informs the client that *aRulerView* will add the new NSRulerMarker, *aMarker*. *location* is the marker's tentative new location, expressed in the client view's coordinate system. The value returned by the client view is actually used; the client can simply return *location* unchanged, or adjust it as needed. For example, it may snap the location to a grid. This message is sent repeatedly to the client as the user drags the marker.

See also: – **rulerView:willMoveMarker:toLocation:**

rulerView:willMoveMarker:toLocation:

– (float)**rulerView:**(NSRulerView *)*aRulerView*
    willMoveMarker:(NSRulerMarker *)*aMarker*
    toLocation:(float)*location*

Informs the client that *aRulerView* will move *aMarker*, an NSRulerMarker already on the ruler view. *location* is the marker's tentative new location, expressed in the client view's coordinate system. The value returned by the client view is actually used; the client can simply return *location* unchanged, or adjust it as needed. For example, it may snap the location to a grid. This message is sent repeatedly to the client as the user drags the marker.

See also: – **rulerView:willAddMarker:atLocation:**

rulerView:willSetClientView:

– (void)**rulerView:**(NSRulerView *)*aRulerView*
  willSetClientView:(NSView *)*newClient*

Informs the client view that *aRulerView* is about to be appropriated by *newClient*. The client view can use this opportunity to clear any cached information related to the ruler.